Учреждение образования
«Брестский государственный технический университет»
Факультет электронно-информационных систем
Кафедра интеллектуальных информационных технологий

СОГЛАСОВАНО
Заведующий кафедрой

_____ В.А. Головко
« 24 » 12 2024 г.

СОГЛАСОВАНО
Декан факультета

_____ А.Н. Парфиевич
« 24 » 12 2024 г.

ЭЛЕКТРОННЫЙ УЧЕБНО-МЕТОДИЧЕСКИЙ КОМПЛЕКС ПО УЧЕБНОЙ ДИСЦИПЛИНЕ

«ОПЕРАЦИОННЫЕ СИСТЕМЫ»

для специальности:

6-05-0612-01 Программная инженерия

СОСТАВИТЕЛЬ:
Ю. И. Давидюк, старший преподаватель кафедры ИИТ

Рассмотрено и утверждено на заседании Научно-методического совета университета 27.12.2024 г., протокол №2

рег. № УМК 24/25 - 15

ПОЯСНИТЕЛЬНАЯ ЗАПИСКА

Актуальность изучения дисциплины обусловлена применением вычислительной техники практически на всех уровнях систем автоматизированной обработки информации в производственной и общественной деятельности. Независимо от масштаба таких систем, начиная от отдельных рабочих мест, локальных, корпоративных и глобальных вычислительных сетей, возникает ряд задач инсталляции и настройки операционной среды на условия применения прикладного программного обеспечения, обеспечения связи процессов в распределенных гетерогенных системах, функционирующих на основе различных стандартов и протоколов.

Изучение системного программного обеспечения необходимо для решения комплексных задач по использованию и распределению ресурсов вычислительных систем, управлению их конфигурацией, производительностью и безопасностью. Знания, полученные при изучении предмета, позволят грамотно использовать функциональные возможности и сервисы современных операционных систем при разработке прикладного программного обеспечения, решении задач системного администрирования.

Цель дисциплины: изучение теоретических основ и приобретение практических навыков проектирования, реализации и сопровождения системных программных средств современных электронных вычислительных машин (ЭВМ), администрирования локальных и распределенных вычислительных систем.

Задачи дисциплины:

– формирование представления о системном программном обеспечении как примере использования и реализации системного подхода в чистом виде;

– приобретение знаний о возможностях, методах, моделях и средствах поддержки современных промышленных информационных технологий;

– формирование навыков практической работы со средствами обеспечения жизненного цикла создания и эволюционного развития сложных программных систем;

– овладение методами использования механизмов операционных систем для автоматизации функций прикладных систем.

Электронный учебно-методический комплекс (ЭУМК) объединяет структурные элементы учебно-методического обеспечения образовательного процесса, и представляет собой сборник материалов теоретического и практического характера для организации работы обучающихся специальности 6-05-0612-01 «Программная инженерия» по изучению дисциплины «Операционные системы».

ЭУМК разработан на основании Положения об учебно-методическом комплексе на уровне высшего образования, утвержденного Постановлением Министерства образования Республики Беларусь от 26 июля 2011 г., № 167, и предназначен для реализации требований учебной программы по учебной дисциплине «Операционные системы» для специальности 6-6-05-0612-01 «Программная инженерия». ЭУМК разработан в полном соответствии с утвержденной учебной программой по учебной дисциплине компонента учреждения высшего образования «Операционные системы».

Цели ЭУМК:

– обеспечение качественного методического сопровождения процесса обучения;

– организация эффективной самостоятельной работы обучающихся.

Содержание и объем ЭУМК полностью соответствуют образовательному стандарту высшего образования специальности ОСВО 6-05-0612-01-2022 специальности 6-05-0612-01

«Программная инженерия», а также учебно-программной документации образовательных программ высшего образования. Материал представлен на требуемом методическом уровне и адаптирован к современным образовательным технологиям. Структура электронного учебно-методического комплекса по дисциплине «Операционные системы»:

Теоретический раздел ЭУМК содержит материалы для теоретического изучения учебной дисциплины и представлен конспектом лекций.

Практический раздел ЭУМК содержит материалы для проведения лабораторных учебных занятий в виде методических указаний для выполнения лабораторных работ.

Раздел контроля знаний ЭУМК содержит примерный перечень вопросов, выносимых на зачет, позволяющих определить соответствие результатов учебной деятельности обучающихся требованиям образовательных стандартов высшего образования и учебно-программной документации образовательных программ высшего образования.

Вспомогательный раздел включает учебную программу по дисциплине «Операционные системы».

Рекомендации по организации работы с ЭУМК

– лекции проводятся с использованием представленных в ЭУМК теоретических материалов, часть материала представляется с использованием персонального компьютера и мультимедийного проектора; при подготовке к зачету, выполнению и защите лабораторных работ обучающиеся могут использовать конспект лекций;

– лабораторные занятия проводятся с использованием представленных в ЭУМК методических указаний,

– зачет может проводиться как в письменной форме, так и в форме тестирования. Вопросы к зачету приведены в разделе контроля знаний.

ЭУМК способствует успешному усвоению обучающимися учебного материала, дает возможность планировать и осуществлять самостоятельную работу обучающихся, обеспечивает рациональное распределение учебного времени по темам учебной дисциплины и совершенствование методики проведения занятий.

ПЕРЕЧЕНЬ МАТЕРИАЛОВ В КОМПЛЕКСЕ

ТЕОРЕТИЧЕСКИЙ РАЗДЕЛ

1 ВВЕДЕНИЕ

1.1 ОПРЕДЕЛЕНИЕ ОПЕРАЦИОННОЙ СИСТЕМЫ

Из чего состоит любая вычислительная система? Во-первых, из того, что в англоязычных странах принято называть словом hardware, или техническое обеспечение: процессор, память, монитор, дисковые устройства и т.д., объединенные магистральным соединением, которое называется шиной.

Во-вторых, вычислительная система состоит из программного обеспечения. Все программное обеспечение принято делить на две части: прикладное и системное. К прикладному программному обеспечению, как правило, относятся разнообразные банковские и прочие бизнес-программы, игры, текстовые процессоры и т. п. Под системным программным обеспечением обычно понимают программы, способствующие функционированию и разработке прикладных программ. Надо сказать, что деление на прикладное и системное программное обеспечение является отчасти условным и зависит от того, кто осуществляет такое деление. Так, обычный пользователь, неискушенный в программировании, может считать Microsoft Word системной программой, а, с точки зрения программиста, это – приложение. Компилятор языка Си для обычного программиста – системная программа, а для системного – прикладная. Несмотря на эту нечеткую грань, данную ситуацию можно отобразить в виде последовательности слоев (см. рис. 1.1), выделив отдельно наиболее общую часть системного программного обеспечения – операционную систему:

Рисунок 1.1 - Слои программного обеспечения компьютерной системы

Большинство пользователей имеет опыт эксплуатации операционных систем, но тем не менее они затруднятся дать этому понятию точное определение. Давайте кратко рассмотрим основные точки зрения.

Операционная система как виртуальная машина

При разработке ОС широко применяется абстрагирование, которое является важным методом упрощения и позволяет сконцентрироваться на взаимодействии высокоуровневых

компонентов системы, игнорируя детали их реализации. В этом смысле ОС представляет собой интерфейс между пользователем и компьютером.

Архитектура большинства компьютеров на уровне машинных команд очень неудобна для использования прикладными программами. Например, работа с диском предполагает знание внутреннего устройства его электронного компонента – контроллера для ввода команд вращения диска, поиска и форматирования дорожек, чтения и записи секторов и т. д. Ясно, что средний программист не в состоянии учитывать все особенности работы оборудования (в современной терминологии – заниматься разработкой драйверов устройств), а должен иметь простую высокоуровневую абстракцию, скажем представляя информационное пространство диска как набор файлов. Файл можно открывать для чтения или записи, использовать для получения или сброса информации, а потом закрывать. Это концептуально проще, чем заботиться о деталях перемещения головок дисков или организации работы мотора. Аналогичным образом, с помощью простых и ясных абстракций, скрываются от программиста все ненужные подробности организации прерываний, работы таймера, управления памятью и т. д. Более того, на современных вычислительных комплексах можно создать иллюзию неограниченного размера оперативной памяти и числа процессоров. Всем этим занимается операционная система. Таким образом, операционная система представляется пользователю виртуальной машиной, с которой проще иметь дело, чем непосредственно с оборудованием компьютера.

Операционная система как менеджер ресурсов

Операционная система предназначена для управления всеми частями весьма сложной архитектуры компьютера. Представим, к примеру, что произойдет, если несколько программ, работающих на одном компьютере, будут пытаться одновременно осуществлять вывод на принтер. Мы получили бы мешанину строчек и страниц, выведенных различными программами. Операционная система предотвращает такого рода хаос за счет буферизации информации, предназначенной для печати, на диске и организации очереди на печать. Для многопользовательских компьютеров необходимость управления ресурсами и их защиты еще более очевидна. Следовательно, операционная система, как менеджер ресурсов, осуществляет упорядоченное и контролируемое распределение процессоров, памяти и других ресурсов между различными программами.

Операционная система как защитник пользователей и программ

Если вычислительная система допускает совместную работу нескольких пользователей, то возникает проблема организации их безопасной деятельности. Необходимо обеспечить сохранность информации на диске, чтобы никто не мог удалить или повредить чужие файлы. Нельзя разрешить программам одних пользователей произвольно вмешиваться в работу программ других пользователей. Нужно пресекать попытки несанкционированного использования вычислительной системы. Всю эту деятельность осуществляет операционная система как организатор безопасной работы пользователей и их программ. С такой точки зрения операционная система представляется системой безопасности государства, на которую возложены полицейские и контрразведывательные функции.

Операционная система как постоянно функционирующее ядро

Наконец, можно дать и такое определение: операционная система – это программа, постоянно работающая на компьютере и взаимодействующая со всеми прикладными программами. Казалось бы, это абсолютно правильное определение, но, как мы увидим дальше, во многих современных операционных системах постоянно работает на компьютере лишь часть операционной системы, которую принято называть ее ядром.

Как мы видим, существует много точек зрения на то, что такое операционная система. Невозможно дать ей адекватное строгое определение. Нам проще сказать не что есть операционная система, а для чего она нужна и что она делает. Для выяснения этого вопроса рассмотрим историю развития вычислительных систем.

1.2 ТРЕБОВАНИЯ, ПРЕДЪЯВЛЯЕМЫЕ К СОВРЕМЕННЫМ ОС

Операционная система является сердцевиной сетевого программного обеспечения, она создает среду для выполнения приложений и во многом определяет, какими полезными для пользователя свойствами эти приложения будут обладать. В связи с этим рассмотрим требования, которым должна удовлетворять современная ОС.

Очевидно, что главным требованием, предъявляемым к операционной системе, является способность выполнения основных функций: эффективного управления ресурсами и обеспечения удобного интерфейса для пользователя и прикладных программ. Современная ОС, как правило, должна реализовывать мультипрограммную обработку, виртуальную память, свопинг, поддерживать многооконный интерфейс, а также выполнять многие другие, совершенно необходимые функции. Кроме этих функциональных требований к операционным системам предъявляются не менее важные рыночные требования. К этим требованиям относятся:

- Расширяемость. Код должен быть написан таким образом, чтобы можно было легко внести дополнения и изменения, если это потребуется, и не нарушить целостность системы.
- Переносимость. Код должен легко переноситься с процессора одного типа на процессор другого типа и с аппаратной платформы (которая включает наряду с типом процессора и способ организации всей аппаратуры компьютера) одного типа на аппаратную платформу другого типа.
- Надежность и отказоустойчивость. Система должна быть защищена как от внутренних, так и от внешних ошибок, сбоев и отказов. Ее действия должны быть всегда предсказуемыми, а приложения не должны быть в состоянии наносить вред ОС.
- Совместимость. ОС должна иметь средства для выполнения прикладных программ, написанных для других операционных систем. Кроме того, пользовательский интерфейс должен быть совместим с существующими системами и стандартами.
- Безопасность. ОС должна обладать средствами защиты ресурсов одних пользователей от других.
- Производительность. Система должна обладать настолько хорошим быстродействием и временем реакции, насколько это позволяет аппаратная платформа.

Рассмотрим более подробно некоторые из этих требований.

Расширяемость

В то время, как аппаратная часть компьютера устаревает за несколько лет, полезная жизнь операционных систем может измеряться десятилетиями. Примером может служить ОС UNIX. Поэтому операционные системы всегда эволюционно изменяются со временем, и эти изменения более значимы, чем изменения аппаратных средств. Изменения ОС обычно представляют собой приобретение ею новых свойств. Например, поддержка новых устройств, таких как CD-ROM, возможность связи с сетями нового типа, поддержка многообещающих технологий, таких как графический интерфейс пользователя или объектно-ориентированное программное окружение, использование более чем одного процессора. Сохранение целостности кода, какие бы изменения не вносились в операционную систему, является главной целью разработки.

Расширяемость может достигаться за счет модульной структуры ОС, при которой программы строятся из набора отдельных модулей, взаимодействующих только через функциональный интерфейс. Новые компоненты могут быть добавлены в операционную

систему модульным путем, они выполняют свою работу, используя интерфейсы, поддерживаемые существующими компонентами.

Использование объектов для представления системных ресурсов также улучшает расширяемость системы. Объекты - это абстрактные типы данных, над которыми можно производить только те действия, которые предусмотрены специальным набором объектных функций. Объекты позволяют единообразно управлять системными ресурсами. Добавление новых объектов не разрушает существующие объекты и не требует изменений существующего кода.

Прекрасные возможности для расширения предоставляет подход к структурированию ОС по типу клиент-сервер с использованием микроядерной технологии. В соответствии с этим подходом ОС строится как совокупность привилегированной управляющей программы и набора непривилегированных услуг-серверов. Основная часть ОС может оставаться неизменной в то время, как могут быть добавлены новые серверы или улучшены старые.

Средства вызова удаленных процедур (RPC) также дают возможность расширить функциональные возможности ОС. Новые программные процедуры могут быть добавлены в любую машину сети и немедленно поступить в распоряжение прикладных программ на других машинах сети.

Некоторые ОС для улучшения расширяемости поддерживают загружаемые драйверы, которые могут быть добавлены в систему во время ее работы. Новые файловые системы, устройства и сети могут поддерживаться путем написания драйвера устройства, драйвера файловой системы или транспортного драйвера и загрузки его в систему.

Переносимость

Требование переносимости кода тесно связано с расширяемостью. Расширяемость позволяет улучшать операционную систему, в то время как переносимость дает возможность перемещать всю систему на машину, базирующуюся на другом процессоре или аппаратной платформе, делая при этом по возможности небольшие изменения в коде. Хотя ОС часто описываются либо как переносимые, либо как непереносимые, переносимость - это не бинарное состояние. Вопрос не в том, может ли быть система перенесена, а в том, насколько легко можно это сделать. Написание переносимой ОС аналогично написанию любого переносимого кода - нужно следовать некоторым правилам.

Во-первых, большая часть кода должна быть написана на языке, который имеется на всех машинах, куда вы хотите переносить систему. Обычно это означает, что код должен быть написан на языке высокого уровня, предпочтительно стандартизованном, например, на языке С. Программа, написанная на ассемблере, не является переносимой, если только вы не собираетесь переносить ее на машину, обладающую командной совместимостью с вашей.

Во-вторых, следует учесть, в какое физическое окружение программа должна быть перенесена. Различная аппаратура требует различных решений при создании ОС. Например, ОС, построенная на 32-битовых адресах, не может быть перенесена на машину с 16-битовыми адресами (разве что с огромными трудностями).

В-третьих, важно минимизировать или, если возможно, исключить те части кода, которые непосредственно взаимодействуют с аппаратными средствами. Зависимость от аппаратуры может иметь много форм. Некоторые очевидные формы зависимости включают прямое манипулирование регистрами и другими аппаратными средствами.

В-четвертых, если аппаратно зависимый код не может быть полностью исключен, то он должен быть изолирован в нескольких хорошо локализуемых модулях. Аппаратно-зависимый код не должен быть распределен по всей системе. Например, можно спрятать

аппаратно-зависимую структуру в программно-задаваемые данные абстрактного типа. Другие модули системы будут работать с этими данными, а не с аппаратурой, используя набор некоторых функций. Когда ОС переносится, то изменяются только эти данные и функции, которые ими манипулируют.

Для легкого переноса ОС при ее разработке должны быть соблюдены следующие требования:

- Переносимый язык высокого уровня. Большинство переносимых ОС написано на языке С (стандарт ANSI X3.159-1989). Разработчики выбирают С потому, что он стандартизован, и потому, что С-компиляторы широко доступны. Ассемблер используется только для тех частей системы, которые должны непосредственно взаимодействовать с аппаратурой (например, обработчик прерываний) или для частей, которые требуют максимальной скорости (например, целочисленная арифметика повышенной точности). Однако непереносимый код должен быть тщательно изолирован внутри тех компонентов, где он используется.
- Изоляция процессора. Некоторые низкоуровневые части ОС должны иметь доступ к процессорно-зависимым структурам данных и регистрам. Однако код, который делает это, должен содержаться в небольших модулях, которые могут быть заменены аналогичными модулями для других процессоров.
- Изоляция платформы. Зависимость от платформы заключается в различиях между рабочими станциями разных производителей, построенными на одном и том же процессоре (например, MIPS R4000). Должен быть введен программный уровень, абстрагирующий аппаратуру (кэши, контроллеры прерываний ввода-вывода и т. п.) вместе со слоем низкоуровневых программ таким образом, чтобы высокоуровневый код не нуждался в изменении при переносе с одной платформы на другую.

Совместимость

Одним из аспектов совместимости является способность ОС выполнять программы, написанные для других ОС или для более ранних версий данной операционной системы, а также для другой аппаратной платформы.

Необходимо разделять вопросы двоичной совместимости и совместимости на уровне исходных текстов приложений. Двоичная совместимость достигается в том случае, когда можно взять исполняемую программу и запустить ее на выполнение на другой ОС. Для этого необходимы: совместимость на уровне команд процессора, совместимость на уровне системных вызовов и даже на уровне библиотечных вызовов, если они являются динамически связываемыми.

Совместимость на уровне исходных текстов требует наличия соответствующего компилятора в составе программного обеспечения, а также совместимости на уровне библиотек и системных вызовов. При этом необходима перекомпиляция имеющихся исходных текстов в новый выполняемый модуль.

Совместимость на уровне исходных текстов важна в основном для разработчиков приложений, в распоряжении которых эти исходные тексты всегда имеются. Но для конечных пользователей практическое значение имеет только двоичная совместимость, так как только в этом случае они могут использовать один и тот же коммерческий продукт, поставляемый в виде двоичного исполняемого кода, в различных операционных средах и на различных машинах.

Обладает ли новая ОС двоичной совместимостью или совместимостью исходных текстов с существующими системами, зависит от многих факторов. Самый главный из них -

архитектура процессора, на котором работает новая ОС. Если процессор, на который переносится ОС, использует тот же набор команд (возможно с некоторыми добавлениями) и тот же диапазон адресов, тогда двоичная совместимость может быть достигнута достаточно просто.

Гораздо сложнее достичь двоичной совместимости между процессорами, основанными на разных архитектурах. Для того, чтобы один компьютер выполнял программы другого (например, DOS-программу на Mac), этот компьютер должен работать с машинными командами, которые ему изначально непонятны. Например, процессор типа 680x0 на Mac должен исполнять двоичный код, предназначенный для процессора 80x86 в PC. Процессор 80x86 имеет свои собственные дешифратор команд, регистры и внутреннюю архитектуру. Процессор 680x0 не понимает двоичный код 80x86, поэтому он должен выбрать каждую команду, декодировать ее, чтобы определить, для чего она предназначена, а затем выполнить эквивалентную подпрограмму, написанную для 680x0. Так как к тому же у 680x0 нет в точности таких же регистров, флагов и внутреннего арифметико-логического устройства, как в 80x86, он должен имитировать все эти элементы с использованием своих регистров или памяти. И он должен тщательно воспроизводить результаты каждой команды, что требует специально написанных подпрограмм для 680x0, гарантирующих, что состояние эмулируемых регистров и флагов после выполнения каждой команды будет в точности таким же, как и на реальном 80x86.

Это простая, но очень медленная работа, так как микрокод внутри процессора 80x86 исполняется на значительно более быстродействующем уровне, чем эмулирующие его внешние команды 680x0. За время выполнения одной команды 80x86 на 680x0, реальный 80x86 может выполнить десятки команд. Следовательно, если процессор, производящий эмуляцию, не настолько быстр, чтобы компенсировать все потери при эмуляции, то программы, исполняющиеся под эмуляцией, будут очень медленными.

Выходом в таких случаях является использование так называемых прикладных сред. Учитывая, что основную часть программы, как правило, составляют вызовы библиотечных функций, прикладная среда имитирует библиотечные функции целиком, используя заранее написанную библиотеку функций аналогичного назначения, а остальные команды эмулирует каждую по отдельности.

Соответствие стандартам POSIX также является средством обеспечения совместимости программных и пользовательских интерфейсов. Во второй половине 80-х правительственные агентства США начали разрабатывать POSIX как стандарты на поставляемое оборудование при заключении правительственных контрактов в компьютерной области. POSIX - это "интерфейс переносимой ОС, базирующейся на UNIX". POSIX - собрание международных стандартов интерфейсов ОС в стиле UNIX. Использование стандарта POSIX (IEEE стандарт 1003.1 - 1988) позволяет создавать программы стиле UNIX, которые могут легко переноситься из одной системы в другую.

Безопасность

В дополнение к стандарту POSIX правительство США также определило требования компьютерной безопасности для приложений, используемых правительством. Многие из этих требований являются желаемыми свойствами для любой многопользовательской системы. Правила безопасности определяют такие свойства, как защита ресурсов одного пользователя от других и установление квот по ресурсам для предотвращения захвата одним пользователем всех системных ресурсов (таких как память).

Обеспечение защиты информации от несанкционированного доступа является обязательной функцией сетевых операционных систем. В большинстве популярных систем гарантируется степень безопасности данных, соответствующая уровню С2 в системе стандартов США.

Основы стандартов в области безопасности были заложены "Критериями оценки надежных компьютерных систем". Этот документ, изданный в США в 1983 году национальным центром компьютерной безопасности (NCSC - National Computer Security Center), часто называют Оранжевой Книгой.

В соответствии с требованиями Оранжевой книги безопасной считается такая система, которая "посредством специальных механизмов защиты контролирует доступ к информации таким образом, что только имеющие соответствующие полномочия лица или процессы, выполняющиеся от их имени, могут получить доступ на чтение, запись, создание или удаление информации".

Иерархия уровней безопасности, приведенная в Оранжевой Книге, помечает низший уровень безопасности как D, а высший - как А.

- В класс D попадают системы, оценка которых выявила их несоответствие требованиям всех других классов.
- Основными свойствами, характерными для С-систем, являются: наличие подсистемы учета событий, связанных с безопасностью, и избирательный контроль доступа. Уровень С делится на 2 подуровня: уровень С1, обеспечивающий защиту данных от ошибок пользователей, но не от действий злоумышленников, и более строгий уровень С2. На уровне С2 должны присутствовать средства секретного входа, обеспечивающие идентификацию пользователей путем ввода уникального имени и пароля перед тем, как им будет разрешен доступ к системе. Избирательный контроль доступа, требуемый на этом уровне позволяет владельцу ресурса определить, кто имеет доступ к ресурсу и что он может с ним делать. Владелец делает это путем предоставляемых прав доступа пользователю или группе пользователей. Средства учета и наблюдения (auditing) - обеспечивают возможность обнаружить и зафиксировать важные события, связанные с безопасностью, или любые попытки создать, получить доступ или удалить системные ресурсы. Защита памяти - заключается в том, что память инициализируется перед тем, как повторно используется. На этом уровне система не защищена от ошибок пользователя, но поведение его может быть проконтролировано по записям в журнале, оставленным средствами наблюдения и аудитинга.
- Системы уровня В основаны на помеченных данных и распределении пользователей по категориям, то есть реализуют мандатный контроль доступа. Каждому пользователю присваивается рейтинг защиты, и он может получать доступ к данным только в соответствии с этим рейтингом. Этот уровень в отличие от уровня С защищает систему от ошибочного поведения пользователя.
- Уровень А является самым высоким уровнем безопасности, он требует в дополнение ко всем требованиям уровня В выполнения формального, математически обоснованного доказательства соответствия системы требованиям безопасности.

Различные коммерческие структуры (например, банки) особо выделяют необходимость учетной службы, аналогичной той, что предлагают государственные рекомендации С2. Любая деятельность, связанная с безопасностью, может быть отслежена и тем самым учтена. Это как раз то, что требует С2 и то, что обычно нужно банкам. Однако, коммерческие пользователи, как правило, не хотят расплачиваться производительностью за повышенный уровень безопасности. А-уровень безопасности занимает своими

управляющими механизмами до 90% процессорного времени. Более безопасные системы не только снижают эффективность, но и существенно ограничивают число доступных прикладных пакетов, которые соответствующим образом могут выполняться в подобной системе. Например для ОС Solaris (версия UNIX) есть несколько тысяч приложений, а для ее аналога B-уровня - только сотня.

1.3 КЛАССИФИКАЦИЯ ОС

Существует несколько схем классификации операционных систем. Ниже приведена классификация по некоторым признакам с точки зрения пользователя.

Реализация многозадачности

По числу одновременно выполняемых задач операционные системы можно разделить на два класса:

- многозадачные (Unix, OS/2, Windows);
- однозадачные (например, MS-DOS).

Многозадачная ОС, решая проблемы распределения ресурсов и конкуренции, полностью реализует мультипрограммный режим в соответствии с требованиями раздела "Основные понятия, концепции ОС".

Многозадачный режим, который воплощает в себе идею разделения времени, называется вытесняющим (preemptive). Каждой программе выделяется квант процессорного времени, по истечении которого управление передается другой программе. Говорят, что первая программа будет вытеснена. В вытесняющем режиме работают пользовательские программы большинства коммерческих ОС.

В некоторых ОС (Windows 3.11, например) пользовательская программа может монополизировать процессор, то есть работать в невытесняющем режиме. Как правило, в большинстве систем не подлежит вытеснению код собственно ОС. Ответственные программы, в частности задачи реального времени, также не вытесняются. Более подробно об этом рассказано в лекции, посвященной планированию работы процессора.

По приведенным примерам можно судить о приблизительности классификации. Так, в ОС MS-DOS можно организовать запуск дочерней задачи и наличие в памяти двух и более задач одновременно. Однако эта ОС традиционно считается однозадачной, главным образом из-за отсутствия защитных механизмов и коммуникационных возможностей.

Поддержка многопользовательского режима

По числу одновременно работающих пользователей ОС можно разделить на:

- однопользовательские (MS-DOS, Windows 3.x);
- многопользовательские (Windows NT, Unix).

Наиболее существенное отличие между этими ОС заключается в наличии у многопользовательских систем механизмов защиты персональных данных каждого пользователя.

Особенности областей использования

Многозадачные ОС подразделяются на три типа в соответствии с использованными при их разработке критериями эффективности:

- системы пакетной обработки (например, ОС ЕС),
- системы разделения времени (UNIX, VMS),
- системы реального времени (QNX, RT/11).

Системы пакетной обработки предназначались для решения задач в основном вычислительного характера, не требующих быстрого получения результатов. Главной целью и критерием эффективности систем пакетной обработки является максимальная пропускная способность, то есть решение максимального числа задач в единицу времени. Для достижения этой цели в системах пакетной обработки используются следующая схема функционирования: в начале работы формируется пакет заданий, каждое задание содержит требование к системным ресурсам; из этого пакета заданий формируется мультипрограммная смесь, то есть множество одновременно выполняемых задач. Для одновременного выполнения выбираются задачи, предъявляющие отличающиеся требования к ресурсам, так, чтобы обеспечивалась сбалансированная загрузка всех устройств вычислительной машины; так, например, в мультипрограммной смеси желательно одновременное присутствие вычислительных задач и задач с интенсивным вводом-выводом. Таким образом, выбор нового задания из пакета заданий зависит от внутренней ситуации, складывающейся в системе, то есть выбирается "выгодное" задание. Следовательно, в таких ОС невозможно гарантировать выполнение того или иного задания в течение определенного периода времени. В системах пакетной обработки переключение процессора с выполнения одной задачи на выполнение другой происходит только в случае, если активная задача сама отказывается от процессора, например, из-за необходимости выполнить операцию ввода-вывода. Поэтому одна задача может надолго занять процессор, что делает невозможным выполнение интерактивных задач. Таким образом, взаимодействие пользователя с вычислительной машиной, на которой установлена система пакетной обработки, сводится к тому, что он приносит задание, отдает его диспетчеру-оператору, а в конце дня после выполнения всего пакета заданий получает результат. Очевидно, что такой порядок снижает эффективность работы пользователя.

Системы разделения времени призваны исправить основной недостаток систем пакетной обработки - изоляцию пользователя-программиста от процесса выполнения его задач. Каждому пользователю системы разделения времени предоставляется терминал, с которого он может вести диалог со своей программой. Так как в системах разделения времени каждой задаче выделяется только квант процессорного времени, ни одна задача не занимает процессор надолго, и время ответа оказывается приемлемым. Если квант выбран достаточно небольшим, то у всех пользователей, одновременно работающих на одной и той же машине, складывается впечатление, что каждый из них единолично использует машину. Ясно, что системы разделения времени обладают меньшей пропускной способностью, чем системы пакетной обработки, так как на выполнение принимается каждая запущенная пользователем задача, а не та, которая "выгодна" системе, и, кроме того, имеются накладные расходы вычислительной мощности на более частое переключение процессора с задачи на задачу. Критерием эффективности систем разделения времени является не максимальная пропускная способность, а удобство и эффективность работы пользователя.

Системы реального времени применяются для управления различными техническими объектами, такими, например, как станок, спутник, научная экспериментальная установка или технологическими процессами, такими, как гальваническая линия, доменный процесс и т.п. Во всех этих случаях существует предельно допустимое время, в течение которого должна быть выполнена та или иная программа, управляющая объектом, в противном случае может произойти авария: спутник выйдет из зоны видимости, экспериментальные данные, поступающие с датчиков, будут потеряны, толщина гальванического покрытия не будет соответствовать норме. Таким образом, критерием эффективности для систем реального времени является их способность выдерживать заранее заданные интервалы времени между

запуском программы и получением результата (управляющего воздействия). Это время называется временем реакции системы, а соответствующее свойство системы - реактивностью. Для этих систем мультипрограммная смесь представляет собой фиксированный набор заранее разработанных программ, а выбор программы на выполнение осуществляется исходя из текущего состояния объекта или в соответствии с расписанием плановых работ.

Некоторые операционные системы могут совмещать в себе свойства систем разных типов, например, часть задач может выполняться в режиме пакетной обработки, а часть - в режиме реального времени или в режиме разделения времени. В таких случаях режим пакетной обработки часто называют фоновым режимом.

2 ИСТОРИЯ ОС

2.1 ЭВОЛЮЦИЯ ВЫЧИСЛИТЕЛЬНЫХ СИСТЕМ

Мы будем рассматривать историю развития именно вычислительных, а не операционных систем, потому что hardware и программное обеспечение эволюционировали совместно, оказывая взаимное влияние друг на друга. Появление новых технических возможностей приводило к прорыву в области создания удобных, эффективных и безопасных программ, а свежие идеи в программной области стимулировали поиски новых технических решений. Именно эти критерии – удобство, эффективность и безопасность – играли роль факторов естественного отбора при эволюции вычислительных систем.

Первый период (1945 -1955). Ламповые машины. Операционных систем нет

Известно, что компьютер был изобретен английским математиком Чарльзом Бэбиджем в конце восемнадцатого века. Его "аналитическая машина" так и не смогла по-настоящему заработать, потому что технологии того времени не удовлетворяли требованиям по изготовлению деталей точной механики, которые были необходимы для вычислительной техники. Известно также, что этот компьютер не имел операционной системы.

Опуская историю механических и электромеханических устройств перейдём сразу к цифровым. Некоторый прогресс в создании цифровых вычислительных машин произошел после второй мировой войны. В середине 40-х были созданы первые ламповые вычислительные устройства и появился принцип программы, хранящейся в памяти машины (John Von Neumann, июнь 1945 г.).. В то время одна и та же группа людей участвовала и в проектировании, и в эксплуатации, и в программировании вычислительной машины. Это была скорее научно-исследовательская работа в области вычислительной техники, а не использование компьютеров в качестве инструмента решения каких-либо практических задач из других прикладных областей. Программирование осуществлялось исключительно на машинном языке. Об операционных системах не было и речи, все задачи организации вычислительного процесса решались вручную каждым программистом с пульта управления. Программа загружалась в память машины в лучшем случае с колоды перфокарт, а обычно с помощью панели переключателей.

Вычислительная система выполняла одновременно только одну операцию (ввод-вывод или собственно вычисления). Отладка программ велась с пульта управления с помощью изучения состояния памяти и регистров машины. В конце этого периода появляется первое системное программное обеспечение: в 1951–1952 гг. возникают прообразы первых компиляторов с символических языков (Fortran и др.), а в 1954 г. Nat Rochester разрабатывает Ассемблер для IBM-701. Другого системного программного обеспечения, кроме библиотек математических и служебных подпрограмм, не было.

Существенная часть времени уходила на подготовку запуска программы, а сами программы выполнялись строго последовательно. Такой режим работы называется последовательной обработкой данных. В целом первый период характеризуется крайне высокой стоимостью вычислительных систем, их малым количеством и низкой эффективностью использования.

Второй период (1955 - 1965). Компьютеры на основе транзисторов. Пакетные операционные системы

С середины 50-х годов начался следующий период в эволюции вычислительной техники, связанный с появлением новой технической базы – полупроводниковых элементов. Применение транзисторов вместо часто перегоравших электронных ламп привело к

повышению надежности компьютеров. Теперь машины могут непрерывно работать достаточно долго, чтобы на них можно было возложить выполнение практически важных задач. Снижается потребление вычислительными машинами электроэнергии, совершенствуются системы охлаждения. Размеры компьютеров уменьшились. Снизилась стоимость эксплуатации и обслуживания вычислительной техники. Началось использование ЭВМ коммерческими фирмами. Одновременно наблюдается бурное развитие алгоритмических языков (LISP, COBOL, ALGOL-60, PL-1 и т.д.). Появляются первые настоящие компиляторы, редакторы связей, библиотеки математических и служебных подпрограмм. Упрощается процесс программирования. Пропадает необходимость взваливать на одних и тех же людей весь процесс разработки и использования компьютеров. Именно в этот период происходит разделение персонала на программистов и операторов, специалистов по эксплуатации и разработчиков вычислительных машин.

Изменяется сам процесс прогона программ. Теперь пользователь приносит программу с входными данными в виде колоды перфокарт и указывает необходимые ресурсы. Такая колода получает название задания. Оператор загружает задание в память машины и запускает его на исполнение. Полученные выходные данные печатаются на принтере, и пользователь получает их обратно через некоторое (довольно продолжительное) время.

Смена запрошенных ресурсов вызывает приостановку выполнения программ, в результате процессор часто простаивает. Для повышения эффективности использования компьютера задания с похожими ресурсами начинают собирать вместе, создавая пакет заданий.

Появляются первые системы пакетной обработки, которые просто автоматизируют запуск одной программы из пакета за другой и тем самым увеличивают коэффициент загрузки процессора. При реализации систем пакетной обработки был разработан формализованный язык управления заданиями, с помощью которого программист сообщал системе и оператору, какую работу он хочет выполнить на вычислительной машине. Системы пакетной обработки стали прообразом современных операционных систем, они были первыми системными программами, предназначенными для управления вычислительным процессом.

Третий период (1965 - 1980). Компьютеры на основе интегральных микросхем. Первые многозадачные ОС

Следующий важный период развития вычислительных машин относится к 1965-1980 годам. В это время в технической базе произошел переход от отдельных полупроводниковых элементов типа транзисторов к интегральным микросхемам, что дало гораздо большие возможности новому, третьему поколению компьютеров.

Для этого периода характерно также создание семейств программно-совместимых машин. Первым семейством программно-совместимых машин, построенных на интегральных микросхемах, явилась серия машин IBM/360. Построенное в начале 60-х годов это семейство значительно превосходило машины второго поколения по критерию цена/производительность. Вскоре идея программно-совместимых машин стала общепризнанной.

Программная совместимость требовала и совместимости операционных систем. Такие операционные системы должны были бы работать и на больших, и на малых вычислительных системах, с большим и с малым количеством разнообразной периферии, в коммерческой области и в области научных исследований. Операционные системы, построенные с намерением удовлетворить всем этим противоречивым требованиям, оказались чрезвычайно сложными "монстрами". Они состояли из многих миллионов ассемблерных строк, написанных тысячами программистов, и содержали тысячи ошибок, вызывающих

нескончаемый поток исправлений. В каждой новой версии операционной системы исправлялись одни ошибки и вносились другие.

Однако, несмотря на необозримые размеры и множество проблем, OS/360 и другие ей подобные операционные системы машин третьего поколения действительно удовлетворяли большинству требований потребителей. Важнейшим достижением ОС данного поколения явилась реализация мультипрограммирования. Мультипрограммирование- это способ организации вычислительного процесса, при котором на одном процессоре попеременно выполняются несколько программ. Пока одна программа выполняет операцию ввода-вывода, процессор не простаивает, как это происходило при последовательном выполнении программ (однопрограммный режим), а выполняет другую программу (многопрограммный режим). При этом каждая программа загружается в свой участок оперативной памяти, называемый разделом.

Другое нововведение - спулинг (spooling). Спулинг в то время определялся как способ организации вычислительного процесса, в соответствии с которым задания считывались с перфокарт на диск в том темпе, в котором они появлялись в помещении вычислительного центра, а затем, когда очередное задание завершалось, новое задание с диска загружалось в освободившийся раздел.

Наряду с мультипрограммной реализацией систем пакетной обработки появился новый тип ОС - системы разделения времени. Вариант мультипрограммирования, применяемый в системах разделения времени, нацелен на создание для каждого отдельного пользователя иллюзии единоличного использования вычислительной машины.

Четвертый период (1980 - настоящее время) . Персональные компьютеры. Классические, сетевые и распределенные системы

Следующий период в эволюции операционных систем связан с появлением больших интегральных схем (БИС). В эти годы произошло резкое возрастание степени интеграции и удешевление микросхем. Компьютер стал доступен отдельному человеку, и наступила эра персональных компьютеров. С точки зрения архитектуры персональные компьютеры ничем не отличались от класса миникомпьютеров типа PDP-11, но вот цена у них существенно отличалась. Если миникомпьютер дал возможность иметь собственную вычислительную машину отделу предприятия или университету, то персональный компьютер сделал это возможным для отдельного человека.

Компьютеры стали широко использоваться неспециалистами, что потребовало разработки "дружественного" программного обеспечения, это положило конец кастовости программистов.

На рынке операционных систем доминировали две системы: MS-DOS и UNIX. Однопрограммная однопользовательская ОС MS-DOS широко использовалась для компьютеров, построенных на базе микропроцессоров Intel 8088, а затем 80286, 80386 и 80486. Мультипрограммная многопользовательская ОС UNIX доминировала в среде "не-интеловских" компьютеров, особенно построенных на базе высокопроизводительных RISC-процессоров.

В середине 80-х стали бурно развиваться сети персональных компьютеров, работающие под управлением сетевых или распределенных ОС.

В сетевых операционных системах пользователи могут получить доступ к ресурсам другого сетевого компьютера, только они должны знать об их наличии и уметь это сделать. Каждая машина в сети работает под управлением своей локальной операционной системы, отличающейся от операционной системы автономного компьютера наличием дополнительных

средств (программной поддержкой для сетевых интерфейсных устройств и доступа к удаленным ресурсам), но эти дополнения не меняют структуру операционной системы.

Распределенная система, напротив, внешне выглядит как обычная автономная система. Пользователь не знает и не должен знать, где его файлы хранятся – на локальной или удаленной машине – и где его программы выполняются. Он может вообще не знать, подключен ли его компьютер к сети. Внутреннее строение распределенной операционной системы имеет существенные отличия от автономных систем.

2.2 СПОСОБЫ ПОСТРОЕНИЯ ЯДРА ОС

Ядро — центральная часть операционной системы, обеспечивающая приложениям координированный доступ к ресурсам компьютера, таким как процессорное время, оперативная память, внешнее оборудование. Обычно предоставляет сервисы файловой системы. [определение из 1-й лекции]

Монолитное ядро

Монолитное ядро — классическая и на сегодняшний день наиболее распространённая архитектура ядер операционных систем. Монолитные ядра предоставляют богатый набор абстракций оборудования. Все части монолитного ядра работают в одном адресном пространстве.

Монолитные ядра имеют долгую историю развития и усовершенствования и на данный момент являются наиболее архитектурно зрелыми и пригодными к эксплуатации. Вместе с тем монолитность ядер усложняет их отладку, понимание кода ядра, добавление новых функций и возможностей, удаление "мёртвого", ненужного, унаследованного от предыдущих версий, кода. Разбухание кода монолитных ядер также повышает требования к объёму оперативной памяти, требуемому для функционирования ядра ОС. Это делает монолитные ядерные архитектуры малопригодными к эксплуатации в системах, сильно ограниченных по объёму ОЗУ, например, встраиваемых системах, производственных микроконтроллерах и т.д.

Альтернативой монолитным ядрам считаются архитектуры, основанные на ***микроядрах***.

Старые монолитные ядра требовали перекомпиляции при любом изменении состава оборудования. Большинство современных ядер позволяют динамически во время работы подгружать модули, выполняющие части функции ядра. Такие ядра называются модульными ядрами. Возможность динамической подгрузки модулей не нарушает монолитности архитектуры ядра, так как динамически подгружаемые модули загружаются в адресное пространство ядра и в дальнейшем работают как интегральная часть ядра. Не следует путать модульность ядра с гибридной или микроядерной архитектурой.

Достоинства: Скорость работы, упрощённая разработка модулей.

Недостатки: Поскольку всё ядро работает в одном адресном пространстве, сбой в одном из компонентов может нарушить работоспособность всей системы.

Примеры: Традиционные ядра UNIX, такие как BSD; Linux.

Модульное ядро

Модульное ядро — современная, усовершенствованная модификация архитектуры монолитных ядер операционных систем компьютеров.

В отличие от «классических» монолитных ядер, считающихся ныне устаревшими, модульные ядра, как правило, не требуют полной перекомпиляции ядра при изменении состава аппаратного обеспечения компьютера. Вместо этого модульные ядра предоставляют тот или иной механизм подгрузки модулей ядра, поддерживающих то или иное аппаратное обеспечение (например, драйверов). При этом подгрузка модулей может быть как динамической (выполняемой «на лету», без перезагрузки ОС, в работающей системе), так и статической (выполняемой при перезагрузке ОС после переконфигурирования системы на загрузку тех или иных модулей).

Все модули ядра работают в адресном пространстве ядра и могут пользоваться всеми функциями, предоставляемыми ядром. Поэтому модульные ядра продолжают оставаться монолитными.

Модульные ядра удобнее для разработки, чем традиционные монолитные ядра, не поддерживающие динамическую загрузку модулей, так как от разработчика не требуется многократная полная перекомпиляция ядра при работе над какой-либо его подсистемой или драйвером. Выявление, локализация, отладка и устранение ошибок при тестировании также облегчаются.

Модульные ядра предоставляют особый программный интерфейс (API) для связывания модулей с ядром, для обеспечения динамической подгрузки и выгрузки модулей. В свою очередь, не любая программа может быть сделана модулем ядра: на модули ядра накладываются определённые ограничения в части используемых функций (например, они не могут пользоваться функциями стандартной библиотеки C/C++ и должны использовать специальные аналоги, являющиеся функциями API ядра). Кроме того, модули ядра обязаны экспортировать определённые функции, нужные ядру для правильного подключения и распознавания модуля, для его корректной инициализации при загрузке и корректного завершения при выгрузке, для регистрации модуля в таблице модулей ядра и для обращения из ядра к сервисам, предоставляемым модулем.

Не все части ядра могут быть сделаны модулями. Некоторые части ядра всегда обязаны присутствовать в оперативной памяти и должны быть жёстко "вшиты" в ядро. Также не все модули допускают динамическую подгрузку (без перезагрузки ОС). Степень модульности ядер (количество и разнообразие кода, которое может быть вынесено в отдельные модули ядра и допускает динамическую подгрузку) различна в различных архитектурах модульных ядер. Ядра Linux в настоящее время имеют более модульную архитектуру, чем ядра *BSD (FreeBSD, NetBSD, OpenBSD).

Общей тенденцией развития современных модульных архитектур является всё большая модуляризация кода (повышение степени модульности ядер), улучшение механизмов динамической подгрузки и выгрузки, уменьшение или устранение необходимости в ручной подгрузке модулей или в переконфигурации ядра при изменениях аппаратуры путём введения тех или иных механизмов автоматического определения оборудования и автоматической подгрузки нужных модулей, универсализация кода ядра и введение в ядро абстрактных механизмов, предназначенных для совместного использования многими модулями (примером может служить VFS — «виртуальная файловая система», совместно используемая многими модулями файловых систем в ядре Linux).

Микроядро

Микроядро — это минимальная реализация основных функций ядра операционной системы компьютера.

Классические микроядра предоставляют лишь очень небольшой набор низкоуровневых примитивов, или системных вызовов, реализующих наиболее базовые сервисы операционной системы. Сюда относятся управление адресным пространством оперативной и виртуальной памяти, управление процессами и тредами, а также средства межпроцессной коммуникации.

Все остальные сервисы ОС, в классических монолитных ядрах ОС предоставляемые непосредственно ядром, в микроядерных архитектурах реализуются в адресном пространстве пользователя и называются серверами. Примерами таких серверов, выносимых в пространство пользователя в микроядерных архитектурах, являются сетевые сервисы, файловая система.

Такая конструкция позволяет сделать структуру ядра «истинно динамичной» и улучшить общее быстродействие системы (небольшое микроядро может уместится в монокристаллическом кэше процессора). Недостатком подобного подхода является плата за принудительное «переключение» процессов в ядре; этот факт собственно и объясняет трудности в проектировании и написании ядер подобной конструкции. Примеры операционных систем на основе микроядра: QNX, GNU Hurd, Minix3.

Экзоядро

Экзоядро — ядро операционной системы компьютеров, предоставляющее лишь функции для взаимодействия между процессами и безопасного выделения и освобождения ресурсов.

Экзо — приставка, обозначающая нечто внешнее, находящееся снаружи.

В традиционных операционных системах ядро предоставляет не только минимальный набор сервисов, обеспечивающих выполнение программ, но и большое количество высокоуровневых абстракций для использования разнородных ресурсов компьютера: оперативной памяти, жестких дисков, сетевых подключений. В отличие от них, ОС на основе экзоядра предоставляет лишь набор сервисов для взаимодействия между приложениями, а также необходимый минимум функций, связанных с защитой: выделение и высвобождение ресурсов, контроль прав доступа, и т. д. Экзоядро не занимается предоставлением абстракций для физических ресурсов — эти функции выносятся в библиотеку пользовательского уровня (так называемую libOS).

Гибридное ядро

Гибридные ядра это модифицированные микроядра, позволяющие для ускорения работы запускать «несущественные» части в пространстве ядра.

Имеют «гибридные» достоинства и недостатки.

Примеры: Windows NT, DragonFlyBSD.

2.3 ОСНОВНЫЕ ПРИНЦИПЫ ПОСТРОЕНИЯ ОС

Частотный принцип

Частотный принцип. Это наиболее общий принцип реализации системных программ. Основан на выделении в алгоритмах в обрабатывающих массивах действий данных по частоте использования. Для наиболее частых создаются условия быстрого выполнения. Например такие программы или данные хранятся в ОЗУ. Кроме того стараются оптимизировать код частых операций.

Принцип модульности

Отражая технологические и эксплуатационные свойства он наиболее эффективен, когда распространяется на ОС, прикладные программы и аппаратуру. Под модулем понимают функциональный элемент системы, законченный и выполненный в ее рамках и средства сопряжения с подобными элементами этой или другой системы. Для сопряжения используется интерфейс.

Принцип функциональной избирательности

Это сочетание 2-х типов. В ОС выделяется некоторая часть постоянных модулей, которые постоянно должны быть под рукой. Они составляют ядро ОС.

Инженерная задача – выбор решения при наличии противоречивых требований, основанных на оптимальном компромиссе. При формировании ядра возникает инженерная задача. В состав должны входить наиболее часто используемые модули, но с другой стороны количество занимаемой памяти не должно быть слишком большим. Поэтому, как правило, в ядро входят модули по управлению системой прерываний, диспетчеры памяти и времени ЦП и некоторые части планировщика. Как правило они постоянно находятся в ОЗУ- резиденты. Кроме них существуют транзитные модули, которые находятся на магнитных дисках, внешних устройствах и загружаются в ОЗУ по сере их необходимости, иногда перекрывая друг друга.

Принцип генерируемости

ОС должна быть настроена на конфигурацию конкретной ВС. Это происходит редко. Источником ОС является дистрибутивный том, транспортная магнитная лента и т.д. Это может быть простое копирование нужных модулей на системный ток (такая технология используется в ДОС) или использование специальной программы со своим входным языком. В результате генерации получается полная версия ОС, адаптированная к конкретной конфигурации ВС и запросам пользователей. Модульность упрощает генерацию.

Принцип функциональной избыточности

Это возможность выполнения одной и той же работы несколькими способами. Это надо для повышения надежности. Пример: Для ОС ЕС для пакетной обработки допускается при конфигурации: MFT, MVT, SVS. В RT-11 используется 5 видов мониторов а монитор – это управляющая программа ОС. В ОС DVCПАК существует 2 способа запуска задачи на чсет: пакетный режим и режим подчиненной задачи. В DOS вывести файл - type или copy…con

Принцип по умолчанию

Заключается в облегчении взаимодействия в системах при генерации и при работе. Его смысл основан на хранении в системе некоторых базовых описаний и характеристик. Определяющих прогнозируемые параметры, объем ОЗУ, время счета и т.д. Эту информацию ОС использует, если пользователь или оператор забудет или умышленно не конкретизирует. Число и значение этих параметров могут определятся при генерации или старте ОС и некоторые из них могут динамически изменяться в процессе работы.

Принцип перемещаемости

Смысл: работоспособность модулей не зависит от их расположения в ОЗУ, т. к. разные модули могут в разных случаях загружаться по разным адресам. Для этого их пишут в перемещаемом коде либо выполняют настройку адресов во время загрузки. Это необходимо, т.к. неизвестно куда будет загружаться модуль в очередной раз, по какому адресу. Этим свойством могут обладать и прикладные программы.

Принцип защиты

Его смысл в ограждении программ и данных пользователя от искажений и нежелательных взаимных влияний. Данные должны защищаться и при хранении и при выполнении. При работе обычно используется аппаратная защита – включение в код программ

привилегированных программ (команд), которые недоступны в режиме пользователя, но доступны ОС.

Один из эффективных способов – контекстная защита, которая основана на известности исполнительного адреса: если адрес не принадлежит контексту задачи, происходит программное прерывание – это аппаратная защита.

Наиболее распространены граничные регистры и защита по ключу.

Традиционным способом защиты является защита при помощи пароля. При правильно выбранном пароле он (способ) является простым и эффективным.

Пример: в Windows 95(Must Die!) ресурс в зависимости от пароля может быть доступен для чтения или чтения/записи. В UNIX 3 вида прав: Чтение, Запись, Выполнение.

Принцип независимости программ от внешних устройств

Желательно, что бы программы пользователя работали со внешними устройствами однотипно, независимо от их физических характеристик. Связь программ с физическими устройствами осуществляется не при трансляции, а при выполнении.

Достоинства:

Можно реализовать практически одинаковый интерфейс взаимодействия с пользовательской программой для всех устройств.

Код обслуживания устройств не дублируется в пользовательских программах.

Обычно код обслуживания ПУ оформляется в виде отдельных модулей (драйверов). В зависимости от ОС драйверы могут грузиться при старте ОС, либо в процессе работы. Наиболее последовательно этот принцип реализован в UNIX.

Принцип открытости и наращиваемости

Открытость - подразумевает доступность ОС для анализа и использования.

Пример: DOS – открытая, т. к. описан интерфейс взаимодействия с ней.

Наращиваемость позволяет вводить в ОС новые модули. Это возможно если ОС открыта.

3. ПРОЦЕССЫ
3.1 ПОНЯТИЕ ПРОЦЕССА

В первой лекции, поясняя понятие "операционная система" и описывая способы построения операционных систем, мы часто применяли слова "программа" и "задание". Мы говорили: вычислительная система исполняет одну или несколько программ, операционная система планирует задания, программы могут обмениваться данными и т. д. Мы использовали эти термины в некотором общеупотребительном, житейском смысле, предполагая, что все читатели одинаково представляют себе, что подразумевается под ними в каждом конкретном случае. При этом одни и те же слова обозначали и объекты в статическом состоянии, не обрабатывающиеся вычислительной системой (например, совокупность файлов на диске), и объекты в динамическом состоянии, находящиеся в процессе исполнения. Это было возможно, пока мы говорили об общих свойствах операционных систем, не вдаваясь в подробности их внутреннего устройства и поведения, или о работе вычислительных систем первого-второго поколений, которые не могли обрабатывать более одной программы или одного задания одновременно, по сути дела не имея операционных систем. Но теперь мы начинаем знакомиться с деталями функционирования современных компьютерных систем, и нам придется уточнить терминологию.

Рассмотрим следующий пример. Два студента запускают программу извлечения

Процесс (или по-другому, задача) - абстракция, описывающая выполняющуюся программу. Для операционной системы процесс представляет собой единицу работы, заявку на потребление системных ресурсов. Подсистема управления процессами планирует выполнение процессов, то есть распределяет процессорное время между несколькими одновременно существующими в системе процессами, а также занимается созданием и уничтожением процессов, обеспечивает процессы необходимыми системными ресурсами, поддерживает взаимодействие между процессами.

Понятие процесса характеризует некоторую совокупность набора исполняющихся команд, ассоциированных с ним ресурсов (выделенная для исполнения память или адресное пространство, стеки, используемые файлы и устройства ввода-вывода и т. д.) и текущего момента его выполнения (значения регистров, программного счетчика, состояние стека и значения переменных), находящуюся под управлением операционной системы.

Не существует взаимно-однозначного соответствия между процессами и программами, обрабатываемыми вычислительными системами. Как будет показано далее, в некоторых операционных системах для работы определенных программ может организовываться более одного процесса или один и тот же процесс может исполнять последовательно несколько различных программ. Более того, даже в случае обработки только одной программы в рамках одного процесса нельзя считать, что процесс представляет собой просто динамическое описание кода исполняемого файла, данных и выделенных для них ресурсов. Процесс находится под управлением операционной системы, поэтому в нем может выполняться часть кода ее ядра (не находящегося в исполняемом файле!), как в случаях, специально запланированных авторами программы (например, при использовании системных вызовов), так и в непредусмотренных ситуациях (например, при обработке внешних прерываний).

3.2 КЛАССИФИКАЦИЯ ПРОЦЕССОВ

По временным признакам

Процессы определяются рядом временных характеристик. В некоторый момент времени процесс может быть порожден (образован), а через некоторое время закончен. Интервал между этими моментами называют интервалом существования процесса.

В момент порождения последовательность и длительность пребывания процесса в каждом из своих состояний (трасса процесса) в общем случае непредсказуемы. Следовательно, непредсказуема и длительность интервала существования. Однако отдельные виды процессов требуют такого планирования, чтобы гарантировать окончание процесса до наступления некоторого конкретного момента времени. Процессы такого класса называют процессами реального времени. В другой класс входят процессы, время существования которых должно быть не более интервала времени допустимой реакции ЭВМ на запросы пользователя. Процессы такого класса называют интерактивными. Процессы, не вошедшие в эти классы, называют пакетными.

По генеалогическому признаку

В любой ОС по требованию существующего или существовавшего процесса проводится работа по порождению процессов. Процесс, задающий данное требование, называют порождающим, а создаваемый по требованию — порожденным. Если порожденный процесс на интервале своего существования в свою очередь выдает требование на порождение другого процесса, то он одновременно становится и порождающим.

При управлении процессами важно обеспечить воспроизводимость результатов работы каждого процесса, учитывать и управлять той ситуацией, которая складывалась при развитии процесса. Поэтому часто для оказывается важен не только результат счета, но и каким образом этот результат достигается. С этих позиций ОС сравнивает процессы по динамическим свойствам, используя понятие "трасса"—порядок и длительность пребывания процесса в допустимых состояниях на интервале существования.

По результативности

Два процесса, которые имеют одинаковый конечный результат обработки одних и тех же исходных данных по одной и той же или даже различным программам на одном и том же или на различных процессорах, называют эквивалентными. Трассы эквивалентных процессов в общем случае не совпадают. Если в каждом из эквивалентных процессов обработка данных происходит по одной и той же программе, но трассы при этом в общем случае не совпадают, то такие процессы называют тождественными. При совпадении трасс у тождественных процессов их называют равными. Во всех остальных случаях процессы всегда различны.

По динамическим признакам

Проблематичность управления процессами заключается в том, что в момент порождения процессов их трассы неизвестны. Кроме того, требуется учитывать, каким образом соотносятся во времени интервалы существования процессов. Если интервалы двух процессов не пересекаются во времени, то такие два процесса называют последовательными друг относительно друга. Если на рассматриваемом интервале времени существуют одновременно два процесса, то они на этом интервале являются параллельными друг относительно друга. Если на рассматриваемом интервале найдется хотя бы одна точка, в которой существует один процесс, но не существует другой, и хотя бы одна точка, в которой оба процесса существуют одновременно, то такие два процесса называют комбинированными.

По принадлежности к ЦП

В операционной системе принято различать процессы не только по времени, но и по месту их развития, т. е. на каком из процессоров исполняется программа процесса. Точкой отсчета принято считать центральный процессор (процессоры), на котором развиваются процессы, называемые программными или внутренними. Такое название указывает на возможность существования в системе процессов, называемых внешними. Это процессы, развитие которых происходит под контролем или управлением ОС на процессорах, отличных от центрального. Ими могут быть, например, процессы ввода — вывода, развивающиеся в канале. Деятельность любого пользователя ЭВМ, который в том или ином виде вводит посредством ОС информацию, требуемую для исполнения одной или нескольких программ, можно также рассматривать как внешний процесс.

По принадлежности к ОС

Программные процессы принято делить на системные и пользовательские. При развитии системного процесса исполняется программа из состава операционной системы. При развитии пользовательского процесса исполняется пользовательская (прикладная) программа.

По связности

Процессы независимо от их вида могут быть взаимосвязанными или изолированными друг от друга. Два процесса являются взаимосвязанными, если между ними поддерживаются с помощью системы управления процессами какого-либо рода связи: функциональные, пространственно-временные, управляющие, информационные и т. д. В противном случае они являются изолированными (точнее - процессами со слабыми связями, так как при отсутствии явных связей они могут быть связаны косвенно и определенным образом влиять на развитие друг друга).

При наличии между процессами управляющей связи устанавливается отношение вида "порождающий—порождаемый", рассмотренное выше. Если два взаимосвязанных процесса при развитии используют совместно некоторые ресурсы, но информационно между собой не связаны, т. е. не обмениваются информацией, то такие процессы называют информационно-независимыми. Связь между такими процессами может быть либо функциональная, либо пространственно-временная. При наличии информационных связей между двумя процессами их называют взаимодействующими, причем схемы, а следовательно, и механизмы установления таких связей могут быть различными. Особенность, во-первых, обусловлена динамикой процессов (т. е. являются ли взаимодействующие процессы последовательными, параллельными или комбинированными); во-вторых, выбранным способом связи (явным, с помощью явного обмена сообщениями между процессами, или неявным, с помощью разделяемых структур данных). Когда необходимо подчеркнуть связь между взаимосвязанными процессами по ресурсам, их называют конкурирующими.

3.3 ДИАГРАММА СОСТОЯНИЯ ПРОЦЕССА

В многозадачной (многопроцессной) системе процесс может находиться в одном из трех основных состояний:

ВЫПОЛНЕНИЕ - активное состояние процесса, во время которого процесс обладает всеми необходимыми ресурсами и непосредственно выполняется процессором;

ОЖИДАНИЕ - пассивное состояние процесса, процесс заблокирован, он не может выполняться по своим внутренним причинам, он ждет осуществления некоторого события,

например, завершения операции ввода-вывода, получения сообщения от другого процесса, освобождения какого-либо необходимого ему ресурса;

ГОТОВНОСТЬ - также пассивное состояние процесса, но в этом случае процесс заблокирован в связи с внешними по отношению к нему обстоятельствами: процесс имеет все требуемые для него ресурсы, он готов выполняться, однако процессор занят выполнением другого процесса.

В ходе жизненного цикла каждый процесс переходит из одного состояния в другое в соответствии с алгоритмом планирования процессов, реализуемым в данной операционной системе. Типичный граф состояний процесса показан на рисунке 3.1:

В состоянии ВЫПОЛНЕНИЕ в однопроцессорной системе может находиться только один процесс, а в каждом из состояний ОЖИДАНИЕ и ГОТОВНОСТЬ - несколько процессов, эти процессы образуют очереди соответственно ожидающих и готовых процессов. Жизненный цикл процесса начинается с состояния ГОТОВНОСТЬ, когда процесс готов к выполнению и ждет своей очереди. При активизации процесс переходит в состояние ВЫПОЛНЕНИЕ и находится в нем до тех пор, пока либо он сам освободит процессор, перейдя в состояние ОЖИДАНИЯ какого-нибудь события, либо будет насильно "вытеснен" из процессора, например, вследствие исчерпания отведенного данному процессу кванта процессорного времени. В последнем случае процесс возвращается в состояние ГОТОВНОСТЬ. В это же состояние процесс переходит из состояния ОЖИДАНИЕ, после того, как ожидаемое событие произойдет.

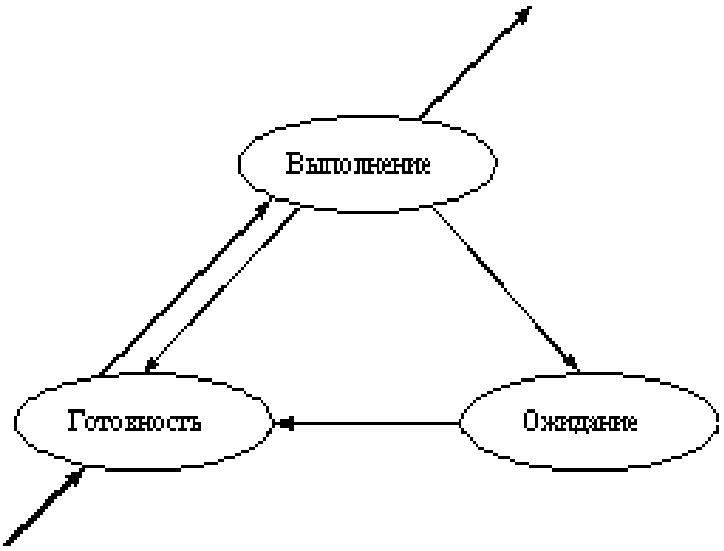


Рисунок 3.1 — Диаграмма состояний процесса

3.4 СТРУКТУРЫ ДАННЫХ, ОПИСЫВАЮЩИЕ ПРОЦЕСС

Таблица процессов

Таблица процессов содержит поля, которые должны быть всегда доступны ядру, а пространство процесса - поля, необходимость в которых возникает только у выполняющегося процесса. Поэтому ядро выделяет место для пространства процесса только при создании процесса: в нем нет необходимости, если записи в таблице процессов не соответствует конкретный процесс.

Пространство процесса

Каждый процесс имеет свое собственное пространство, однако ядро обращается к пространству выполняющегося процесса так, как если бы в системе оно было единственным.

Процесс имеет доступ к своему пространству, когда выполняется в режиме ядра, но не тогда, когда выполняется в режиме задачи. Поскольку ядро в каждый момент времени работает только с одним пространством процесса, используя для доступа виртуальный адрес, пространство процесса частично описывает контекст процесса, выполняющегося в системе. Когда ядро выбирает процесс для исполнения, оно ищет в физической памяти соответствующее процессу пространство и делает его доступным по виртуальному адресу.

Контекст процесса

На протяжении существования процесса его выполнение может быть многократно прервано и продолжено. Для того, чтобы возобновить выполнение процесса, необходимо восстановить состояние его операционной среды. Состояние операционной среды отображается состоянием регистров и программного счетчика, режимом работы процессора, указателями на открытые файлы, информацией о незавершенных операциях ввода-вывода, кодами ошибок выполняемых данным процессом системных вызовов и т.д. Эта информация называется контекстом процесса.

Дескриптор процесса

Кроме этого, операционной системе для реализации планирования процессов требуется дополнительная информация: идентификатор процесса, состояние процесса, данные о степени привилегированности процесса, место нахождения кодового сегмента и другая информация. В некоторых ОС (например, в ОС UNIX) информацию такого рода, используемую ОС для планирования процессов, называют дескриптором процесса.

Дескриптор процесса по сравнению с контекстом содержит более оперативную информацию, которая должна быть легко доступна подсистеме планирования процессов. Контекст процесса содержит менее актуальную информацию и используется операционной системой только после того, как принято решение о возобновлении прерванного процесса.

Очереди процессов

Очереди процессов представляют собой дескрипторы отдельных процессов, объединенные в списки. Таким образом, каждый дескриптор, кроме всего прочего, содержит по крайней мере один указатель на другой дескриптор, соседствующий с ним в очереди. Такая организация очередей позволяет легко их переупорядочивать, включать и исключать процессы, переводить процессы из одного состояния в другое.

4 ПОТОКИ
4.1 ПОНЯТИЕ ПОТОКА

Многопоточность — свойство операционной системы, заключающееся в том, что задача может выполняться в более чем 1 потоке, за счёт чего достигается более эффективное использование ресурсов вычислительной машины. Как правило, операционные системы, реализующие многозадачность, реализуют и многопоточность.

Сутью многопоточности является квазимногозадачность на уровне одного исполняемого процесса, то есть все потоки выполняются в адресном пространстве процесса. На момент выполнения процесса он имеет как минимум один (главный) поток.

К достоинствам многопоточности в программировании можно отнести следующее:

- Упрощение программы в некоторых случаях, за счет использования общего адресного пространства.
- Быстрота создания потока, по сравнению с процессом, примерно в 100 раз.
- Повышение производительности самой программы, т.к. есть возможность одновременно выполнять вычисления на процессоре и операцию ввода/вывода. Пример: текстовый редактор с тремя потоками может одновременно взаимодействовать с пользователем, форматировать текст и записывать на диск резервную копию.

Каждому процессу соответствует адресное пространство и одиночный поток исполняемых команд. В многопользовательских системах, при каждом обращении к одному и тому же сервису, приходится создавать новый процесс для обслуживания клиента. Это менее выгодно, чем создать квазипараллельный поток внутри этого процесса с одним адресным пространством (см. рисунок 4.1).

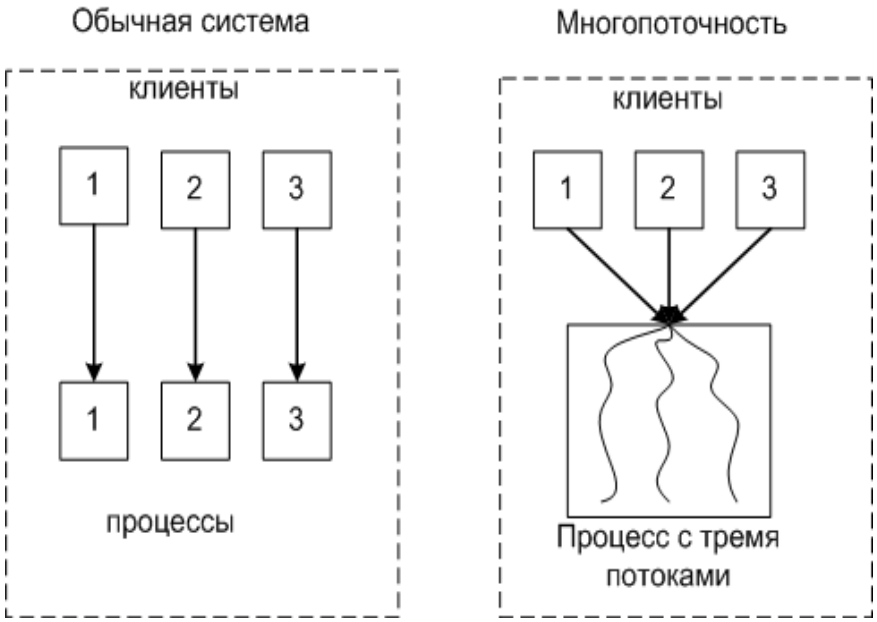


Рисунок 4.1 — Сравнение многопоточной системы с однопоточной

С каждым потоком связывается: счетчик выполнения команд, регистры для текущих переменных, стек, состояние.

Потоки делят между собой элементы своего процесса: адресное пространство, глобальные переменные, открытые файлы, таймеры, семафоры, статистическую информацию.

В остальном модель идентична модели процессов.

В Windows есть поддержка потоков на уровне ядра. В Linux есть новый системный вызов clone для создания потоков, отсутствующий во всех остальных версиях системы UNIX. В POSIX есть новый системный вызов pthread_create для создания потоков. В Windows есть новый системный вызов Createthread для создания потоков.

4.2 СПОСОБЫ РЕАЛИЗАЦИИ ПОТОКОВ

Поток в пространстве пользователя

Каждый процесс имеет таблицу потоков, аналогичную таблице процессов ядра. В этом случае ядро о потоках ничего не знает.

Преимущества:

- Такую многопоточность можно реализовать на ядре не поддерживающим многопоточность
- Более быстрое переключение, создание и завершение потоков
- Процесс может иметь собственный алгоритм планирования.

Недостатки:

- Отсутствие прерывания по таймеру внутри одного процесса
- При использовании блокирующего (процесс переводится в режим ожидания, например: чтение с клавиатуры, а данные не поступают) системного запроса все остальные потоки блокируются.
- Сложность реализации

Поток в пространстве ядра

Наряду с таблицей процессов в пространстве ядра имеется таблица потоков. Недостатки и преимущества противоположны предыдущему пункту.

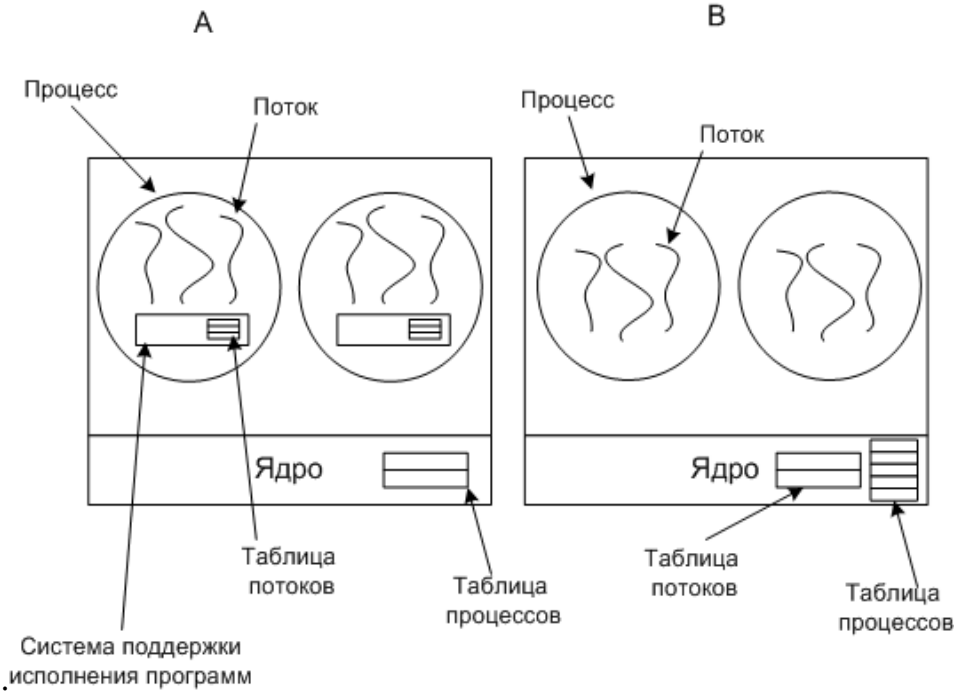
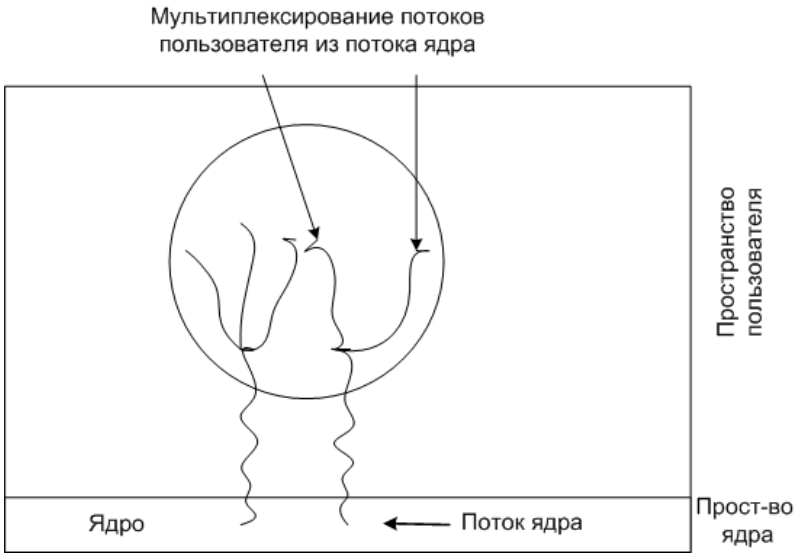


Рисунок 4.2 – Способы организаций потоков. А - потоки в пространстве пользователя. В - потоки в пространстве ядра

Смешанная реализация

Существует множество вариантов смешанной реализации. Основной: потоки работают в режиме пользователя, но при системных вызовах переключаются в режим ядра. Переключение в режим ядра и обратно является ресурсоемкой операцией и отрицательно сказывается на производительности системы (см. рисунок 4.3).



4.3 – Мультиплексирование потоков пользователя в потоках ядра

Поток ядра может содержать несколько несколько потоков пользователя.

4.3 ВЗАИМОДЕЙСТВИЕ ПОТОКОВ

В многопоточной среде часто возникают проблемы, связанные с использованием параллельно исполняемыми потоками одних и тех же данных или устройств. Для решения подобных проблем используются такие методы взаимодействия потоков, как взаимоисключения (мьютексы), семафоры, критические секции и события.

Мьютексы

Взаимоисключения (mutex, мьютекс) — это объект синхронизации, который устанавливается в особое сигнальное состояние, когда не занят каким-либо потоком. Только один поток владеет этим объектом в любой момент времени, отсюда и название таких объектов — одновременный доступ к общему ресурсу исключается. После всех необходимых действий мьютекс освобождается, предоставляя другим потокам доступ к общему ресурсу.

Семафоры

Семафоры представляют собой доступные ресурсы, которые могут быть приобретены несколькими потоками в одно и то же время, пока пул ресурсов не опустеет. Тогда дополнительные потоки должны ждать, пока требуемое количечтво ресурсов не будет снова доступно. Семафоры очень эффективны, поскольку они позволяют одновременный доступ к ресурсам.

Критические секции

Критические секции обеспечивают синхронизацию подобно мьютексам за исключением того, что объекты, представляющие критические секции, доступны в пределах одного процесса. События, мьютексы и семафоры также можно использовать в однопроцессном приложении, однако критические секции обеспечивают более быстрый и

более эффективный механизм взаимно-исключающей синхронизации. Подобно мьютексам объект, представляющий критическую секцию, может использоваться только одним потоком в данный момент времени, что делает их крайне полезными при разграничении доступа к общим ресурсам.

События

События полезны в тех случаях, когда необходимо послать сообщение потоку, сообщающее, что произошло определенное событие. Например, при асинхронных операциях ввода и вывода из одного устройства, система устанавливает событие в сигнальное состояние когда заканчивается какая-либо из этих операций. Один поток может использовать несколько различных событий в нескольких перекрывающихся операциях, а затем ожидать прихода сигнала от любого из них.

5. ПОДСИСТЕМА УПРАВЛЕНИЯ ПРОЦЕССАМИ
5.1 УРОВНИ ПЛАНИРОВАНИЯ ПРОЦЕССОВ

В первой лекции, рассматривая эволюцию компьютерных систем, мы говорили о двух видах планирования в вычислительных системах: планировании заданий и планировании использования процессора. Планирование заданий появилось в пакетных системах после того, как для хранения сформированных пакетов заданий начали использоваться магнитные диски. Магнитные диски, являясь устройствами прямого доступа, позволяют загружать задания в компьютер в произвольном порядке, а не только в том, в котором они были записаны на диск. Изменяя порядок загрузки заданий в вычислительную систему, можно повысить эффективность ее использования. Процедуру выбора очередного задания для загрузки в машину, т. е. для порождения соответствующего процесса, мы и назвали планированием заданий. Планирование использования процессора впервые возникает в мультипрограммных вычислительных системах, где в состоянии готовность могут одновременно находиться несколько процессов. Именно для процедуры выбора из них одного процесса, который получит процессор в свое распоряжение, т. е. будет переведен в состояние исполнение, мы использовали это словосочетание. Теперь, познакомившись с концепцией процессов в вычислительных системах, оба вида планирования мы будем рассматривать как различные уровни планирования процессов.

Планирование заданий используется в качестве долгосрочного планирования процессов. Оно отвечает за порождение новых процессов в системе, определяя ее степень мультипрограммирования, т. е. количество процессов, одновременно находящихся в ней. Если степень мультипрограммирования системы поддерживается постоянной, т. е. среднее количество процессов в компьютере не меняется, то новые процессы могут появляться только после завершения ранее загруженных. Поэтому долгосрочное планирование осуществляется достаточно редко, между появлением новых процессов могут проходить минуты и даже десятки минут. Решение о выборе для запуска того или иного процесса оказывает влияние на функционирование вычислительной системы на протяжении достаточно длительного времени. Отсюда и название этого уровня планирования – долгосрочное. В некоторых операционных системах долгосрочное планирование сведено к минимуму или отсутствует вовсе. Так, например, во многих интерактивных системах разделения времени порождение процесса происходит сразу после появления соответствующего запроса. Поддержание разумной степени мультипрограммирования осуществляется за счет ограничения количества пользователей, которые могут работать в системе, и особенностей человеческой психологии. Если между нажатием на клавишу и появлением символа на экране проходит 20–30 секунд, то многие пользователи предпочтут прекратить работу и продолжить ее, когда система будет менее загружена.

Планирование использования процессора применяется в качестве краткосрочного планирования процессов. Оно проводится, к примеру, при обращении исполняющегося процесса к устройствам ввода-вывода или просто по завершении определенного интервала времени. Поэтому краткосрочное планирование осуществляется, как правило, не реже одного раза в 100 миллисекунд. Выбор нового процесса для исполнения оказывает влияние на функционирование системы до наступления очередного аналогичного события, т. е. в течение короткого промежутка времени, чем и обусловлено название этого уровня планирования – краткосрочное.

В некоторых вычислительных системах бывает выгодно для повышения производительности временно удалить какой-либо частично выполнившийся процесс из оперативной памяти на диск, а позже вернуть его обратно для дальнейшего выполнения. Такая процедура в англоязычной литературе получила название swapping, что можно перевести на русский язык как "перекачка", хотя в специальной литературе оно употребляется без перевода – свопинг. Когда и какой из процессов нужно перекачать на диск и вернуть обратно, решается дополнительным промежуточным уровнем планирования процессов – среднесрочным.

Процесс планирования осуществляется частью операционной системы, называемой планировщиком.

Цели применения алгоритмов планирования

Для каждого уровня планирования процессов можно предложить много различных алгоритмов. Выбор конкретного алгоритма определяется классом задач, решаемых вычислительной системой (будут рассмотрены далее), и целями, которых мы хотим достичь, используя планирование. К числу таких целей можно отнести следующие:

Справедливость – гарантировать каждому заданию или процессу определенную часть времени использования процессора в компьютерной системе, стараясь не допустить возникновения ситуации, когда процесс одного пользователя постоянно занимает процессор, в то время как процесс другого пользователя фактически не начинал выполняться.

Эффективность – постараться занять процессор на все 100% рабочего времени, не позволяя ему простаивать в ожидании процессов, готовых к исполнению. В реальных вычислительных системах загрузка процессора колеблется от 40 до 90%.

Сокращение полного времени выполнения (turnaround time) – обеспечить минимальное время между стартом процесса или постановкой задания в очередь для загрузки и его завершением.

Сокращение времени ожидания (waiting time) – сократить время, которое проводят процессы в состоянии готовность и задания в очереди для загрузки.

Сокращение времени отклика (response time) – минимизировать время, которое требуется процессу в интерактивных системах для ответа на запрос пользователя.

Свойства алгоритмов

Независимо от поставленных целей планирования желательно также, чтобы алгоритмы обладали следующими свойствами.

Были предсказуемыми. Одно и то же задание должно выполняться приблизительно за одно и то же время. Применение алгоритма планирования не должно приводить, к примеру, к извлечению квадратного корня из 4 за сотые доли секунды при одном запуске и за несколько суток – при втором запуске.

Были связаны с минимальными накладными расходами. Если на каждые 100 миллисекунд, выделенные процессу для использования процессора, будет приходиться 200 миллисекунд на определение того, какой именно процесс получит процессор в свое распоряжение, и на переключение контекста, то такой алгоритм, очевидно, применять не стоит.

Равномерно загружали ресурсы вычислительной системы, отдавая предпочтение тем процессам, которые будут занимать малоиспользуемые ресурсы.

Были масштабируемыми, т. е. не сразу теряли работоспособность при увеличении нагрузки. Например, рост количества процессов в системе в два раза не должен приводить к увеличению полного времени выполнения процессов на порядок.

Многие из приведенных выше целей и свойств являются противоречивыми. Улучшая работу алгоритма с точки зрения одного критерия, мы ухудшаем ее с точки зрения другого. Приспосабливая алгоритм под один класс задач, мы тем самым дискриминируем задачи другого класса. "В одну телегу впрячь не можно коня и трепетную лань". Ничего не поделаешь. Такова жизнь.

5.2 СТРАТЕГИИ ПЛАНИРОВАНИЯ

Вытесняющее и невытесняющее планирование

Планировщик может принимать решения о выборе для исполнения нового процесса из числа находящихся в состоянии готовность в следующих четырех случаях.

- Когда процесс переводится из состояния исполнение в состояние закончил исполнение.
- Когда процесс переводится из состояния исполнение в состояние ожидание.
- Когда процесс переводится из состояния исполнение в состояние готовность (например, после прерывания от таймера).
- Когда процесс переводится из состояния ожидание в состояние готовность (завершилась операция ввода-вывода или произошло другое событие).

В случаях 1 и 2 процесс, находившийся в состоянии исполнение, не может дальше исполняться, и операционная система вынуждена осуществлять планирование выбирая новый процесс для выполнения. В случаях 3 и 4 планирование может как проводиться, так и не проводиться, планировщик не вынужден обязательно принимать решение о выборе процесса для выполнения, процесс, находившийся в состоянии исполнение может просто продолжить свою работу. Если в операционной системе планирование осуществляется только в вынужденных ситуациях, говорят, что имеет место невытесняющее (nonpreemptive) планирование. Если планировщик принимает и вынужденные, и невынужденные решения, говорят о вытесняющем (preemptive) планировании. Термин "вытесняющее планирование" возник потому, что исполняющийся процесс помимо своей воли может быть вытеснен из состояния исполнение другим процессом.

Невытесняющее планирование используется, например, в MS Windows 3.1 и ОС Apple Macintosh. При таком режиме планирования процесс занимает столько процессорного времени, сколько ему необходимо. При этом переключение процессов возникает только при желании самого исполняющегося процесса передать управление (для ожидания завершения операции ввода-вывода или по окончании работы). Этот метод планирования относительно просто реализуем и достаточно эффективен, так как позволяет выделить большую часть процессорного времени для работы самих процессов и до минимума сократить затраты на переключение контекста. Однако при невытесняющем планировании возникает проблема возможности полного захвата процессора одним процессом, который вследствие каких-либо причин (например, из-за ошибки в программе) зацикливается и не может передать управление другому процессу. В такой ситуации спасает только перезагрузка всей вычислительной системы.

Вытесняющее планирование обычно используется в системах разделения времени. В этом режиме планирования процесс может быть приостановлен в любой момент исполнения. Операционная система устанавливает специальный таймер для генерации сигнала прерывания по истечении некоторого интервала времени – кванта. После прерывания процессор

передается в распоряжение следующего процесса. Временные прерывания помогают гарантировать приемлемое время отклика процессов для пользователей, работающих в диалоговом режиме, и предотвращают "зависание" компьютерной системы из-за зацикливания какой-либо программы.

Алгоритмы, основанные на квантовании

В соответствии с алгоритмами, основанными на квантовании, смена активного процесса происходит, если:

- процесс завершился и покинул систему,
- произошла ошибка,
- процесс перешел в состояние ОЖИДАНИЕ,
- исчерпан квант процессорного времени, отведенный данному процессу.

Процесс, который исчерпал свой квант, переводится в состояние ГОТОВНОСТЬ и ожидает, когда ему будет предоставлен новый квант процессорного времени, а на выполнение в соответствии с определенным правилом выбирается новый процесс из очереди готовых. Таким образом, ни один процесс не занимает процессор надолго, поэтому квантование широко используется в системах разделения времени. Граф состояний процесса, изображенный на **Error! Reference source not found.**, соответствует алгоритму планирования, основанному на квантовании.

Кванты, выделяемые процессам, могут быть одинаковыми для всех процессов или различными. Кванты, выделяемые одному процессу, могут быть фиксированной величины или изменяться в разные периоды жизни процесса. Процессы, которые не полностью использовали выделенный им квант (например, из-за ухода на выполнение операций ввода-вывода), могут получить или не получить компенсацию в виде привилегий при последующем обслуживании. По разному может быть организована очередь готовых процессов: циклически, по правилу "первый пришел - первый обслужился" (FIFO) или по правилу "последний пришел - первый обслужился" (LIFO).

Алгоритмы, основанные на приоритетах

Другая группа алгоритмов использует понятие "приоритет" процесса. Приоритет- это число, характеризующее степень привилегированности процесса при использовании ресурсов вычислительной машины, в частности, процессорного времени: чем выше приоритет, тем выше привилегии.

Приоритет может выражаться целыми или дробными, положительным или отрицательным значением. Чем выше привилегии процесса, тем меньше времени он будет проводить в очередях. Приоритет может назначаться директивно администратором системы в зависимости от важности работы или внесенной платы, либо вычисляться самой ОС по определенным правилам, он может оставаться фиксированным на протяжении всей жизни процесса либо изменяться во времени в соответствии с некоторым законом. В последнем случае приоритеты называются динамическими.

Существует две разновидности приоритетных алгоритмов: алгоритмы, использующие относительные приоритеты, и алгоритмы, использующие абсолютные приоритеты.

В обоих случаях выбор процесса на выполнение из очереди готовых осуществляется одинаково: выбирается процесс, имеющий наивысший приоритет. По разному решается проблема определения момента смены активного процесса. В системах с относительными приоритетами активный процесс выполняется до тех пор, пока он сам не покинет процессор, перейдя в состояние ОЖИДАНИЕ (или же произойдет ошибка, или процесс завершится). В

системах с абсолютными приоритетами выполнение активного процесса прерывается еще при одном условии: если в очереди готовых процессов появился процесс, приоритет которого выше приоритета активного процесса. В этом случае прерванный процесс переходит в состояние готовности. На рисунке показаны графы состояний процесса для алгоритмов с относительными (а) и абсолютными (б) приоритетами.

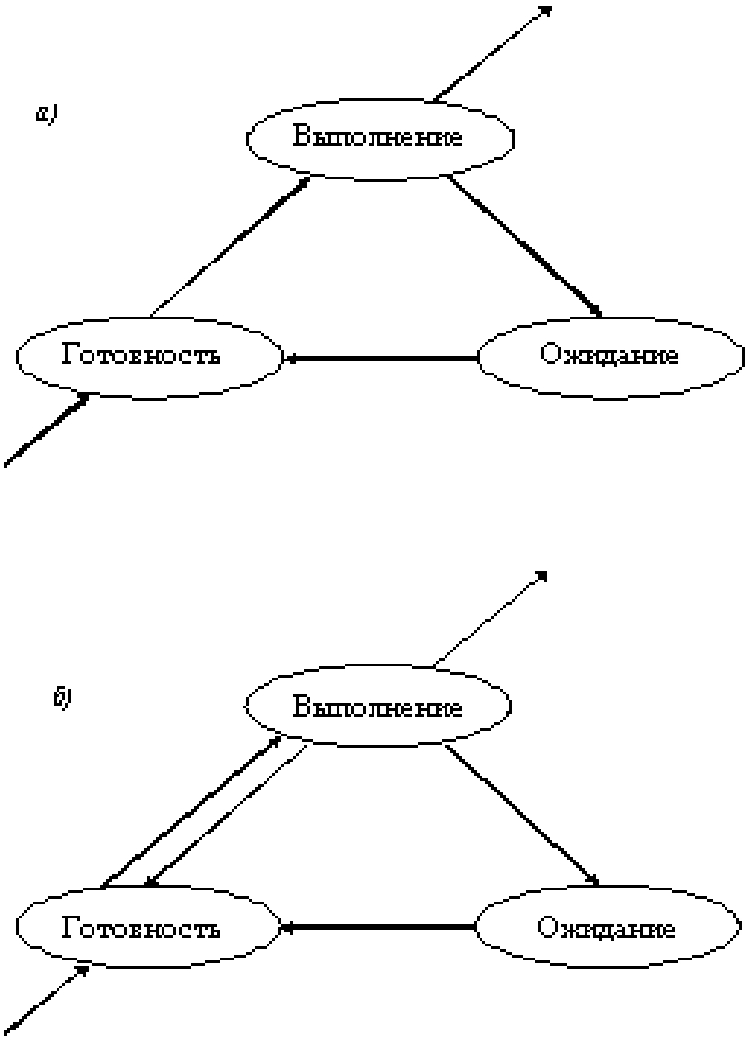


Рисунок 5.1 — Графы состояний процессов в система
(а) с относительными приоритетами; (б)с абсолютными приоритетами

Во многих операционных системах алгоритмы планирования построены с использованием как квантования, так и приоритетов. Например, в основе планирования лежит квантование, но величина кванта и/или порядок выбора процесса из очереди готовых определяется приоритетами процессов.

5.3 КАТЕГОРИИ АЛГОРИТМОВ ПЛАНИРОВАНИЯ

Необходимость алгоритма планирования зависит от задач, для которых будет использоваться операционная система.

Основные три среды:

- Системы пакетной обработки - могут использовать неприоритетный и приоритетный алгоритм (например: для расчетных программ).
- Интерактивные системы - могут использовать только приоритетный алгоритм, нельзя допустить чтобы один процесс занял надолго процессор (например: сервер общего доступа или персональный компьютер).
- Системы реального времени - могут использовать неприоритетный и приоритетный алгоритм (например: система управления автомобилем).

Задачи алгоритмов планирования

Для всех систем:

- Справедливость - каждому процессу справедливую долю процессорного времени.
- Контроль над выполнением принятой политики.
- Баланс - поддержка занятости всех частей системы (например: чтобы были заняты процессор и устройства ввода/вывода).

Системы пакетной обработки:

- Пропускная способность - количество задач в час
- Оборотное время - минимизация времени на ожидание обслуживания и обработку задач.
- Использование процесса - чтобы процессор всегда был занят.

Интерактивные системы:

- Время отклика - быстрая реакция на запросы.
- Соразмерность - выполнение ожиданий пользователя (например: пользователь не готов к долгой загрузке системы).

Системы реального времени:

- Окончание работы к сроку - предотвращение потери данных.
- Предсказуемость - предотвращение деградации качества в мультимедийных системах (например: потерь качества звука должно быть меньше чем видео).

5.4 ПЛАНИРОВАНИЕ В СИСТЕМАХ ПАКЕТНОЙ ОБРАБОТКИ ДАННЫХ

FCFS

Простейшим алгоритмом планирования является алгоритм, который принято обозначать аббревиатурой FCFS по первым буквам его английского названия – First-Come, First-Served (первым пришел, первым обслужен). Представим себе, что процессы, находящиеся в состоянии готовность, выстроены в очередь. Когда процесс переходит в состояние готовность, он, а точнее, ссылка на его PCB помещается в конец этой очереди. Выбор нового процесса для исполнения осуществляется из начала очереди с удалением оттуда ссылки на его PCB. Очередь подобного типа имеет в программировании специальное наименование – FIFO, сокращение от First In, First Out (первым вошел, первым вышел).

Такой алгоритм выбора процесса осуществляет невытесняющее планирование. Процесс, получивший в свое распоряжение процессор, занимает его до истечения текущего CPU burst. После этого для выполнения выбирается новый процесс из начала очереди (см. рисунок 5.1).

Таблица 5.1 — CPU burst для FCFS

Процесс	p0	p1	p2
Продолжительность очередного CPU burst	13	4	1

Преимуществом алгоритма FCFS является легкость его реализации, но в то же время он имеет и много недостатков. Рассмотрим следующий пример. Пусть в состоянии готовность находятся три процесса p0, p1 и p2, для которых известны времена их очередных CPU burst. Эти времена приведены в таблице 3.1. в некоторых условных единицах. Для простоты будем полагать, что вся деятельность процессов ограничивается использованием только одного промежутка CPU burst, что процессы не совершают операций ввода-вывода и что время переключения контекста так мало, что им можно пренебречь.

Если процессы расположены в очереди процессов, готовых к исполнению, в порядке p0, p1, p2, то картина их выполнения выглядит так, как показано на рисунке 5.2. Первым для выполнения выбирается процесс p0, который получает процессор на все время своего CPU burst, т. е. на 13 единиц времени. После его окончания в состояние исполнение переводится процесс p1, он занимает процессор на 4 единицы времени. И, наконец, возможность работать получает процесс p2. Время ожидания для процесса p0 составляет 0 единиц времени, для процесса p1 – 13 единиц, для процесса p2 – 13 + 4 = 17 единиц. Таким образом, среднее время ожидания в этом случае – (0 + 13 + 17)/3 = 10 единиц времени. Полное время выполнения для процесса p0 составляет 13 единиц времени, для процесса p1 – 13 + 4 = 17 единиц, для процесса p2 – 13 + 4 + 1 = 18 единиц. Среднее полное время выполнения оказывается равным (13 + 17 + + 18)/3 = 16 единицам времени.

Рисунок 5.2 — Выполнение процессов при порядке p0,p1,p2

Если те же самые процессы расположены в порядке p2, p1, p0, то картина их выполнения будет соответствовать рисунку 5.3. Время ожидания для процесса p0 равняется 5 единицам времени, для процесса p1 – 1 единице, для процесса p2 – 0 единиц. Среднее время ожидания составит (5 + 1 + 0)/3 = 2 единицы времени. Это в 5 (!) раз меньше, чем в предыдущем случае. Полное время выполнения для процесса p0 получается равным 18 единицам времени, для процесса p1 – 5 единицам, для процесса p2 – 1 единице. Среднее полное время выполнения составляет (18 + 5 + 1)/3 = 8 единиц времени, что почти в 2 раза меньше, чем при первой расстановке процессов.

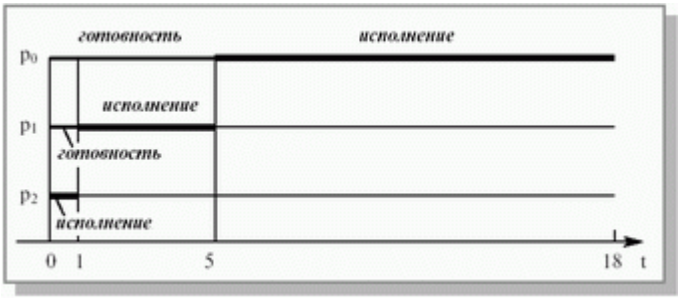


Рисунок 5.3 — Выполнение процессов при порядке p2, p1, p0

Поэтому алгоритм FCFS практически неприменим для систем разделения времени – слишком большим получается среднее время отклика в интерактивных процессах.

SJN

При рассмотрении алгоритмов FCFS и RR мы видели, насколько существенным для них является порядок расположения процессов в очереди процессов, готовых к исполнению. Если короткие задачи расположены в очереди ближе к ее началу, то общая производительность этих алгоритмов значительно возрастает. Если бы мы знали время следующих CPU burst для процессов, находящихся в состоянии готовность, то могли бы выбрать для исполнения не процесс из начала очереди, а процесс с минимальной длительностью CPU burst. Если же таких процессов два или больше, то для выбора одного из них можно использовать уже известный нам алгоритм FCFS. Квантование времени при этом не применяется. Описанный алгоритм получил название "кратчайшая работа первой" или Shortest Job Next (SJN).

SJN-алгоритм краткосрочного планирования может быть как вытесняющим, так и невытесняющим. При невытесняющем SJN-планировании процессор предоставляется избранному процессу на все необходимое ему время, независимо от событий, происходящих в вычислительной системе. При вытесняющем SJN-планировании учитывается появление новых процессов в очереди готовых к исполнению (из числа вновь родившихся или разблокированных) во время работы выбранного процесса. Если CPU burst нового процесса меньше, чем остаток CPU burst у исполняющегося, то исполняющийся процесс вытесняется новым.

Рассмотрим пример работы невытесняющего алгоритма SJN. Пусть в состоянии готовность находятся четыре процесса, p0, p1, p2 и p3, для которых известны времена их очередных CPU burst. Эти времена приведены в таблице 5.4. Как и прежде, будем полагать, что вся деятельность процессов ограничивается использованием только одного промежутка CPU burst, что процессы не совершают операций ввода-вывода и что временем переключения контекста можно пренебречь.

Таблица 5.4 — CPU burst для SJN

Процесс	p0	p1	p2	p3
Продолжительность очередного CPU burst	5	3	7	1

При использовании невытесняющего алгоритма SJN первым для исполнения будет выбран процесс p3, имеющий наименьшее значение продолжительности очередного CPU burst. После его завершения для исполнения выбирается процесс p1, затем p0 и, наконец, p2. Эта картина отражена в таблице 5.5.

Таблица 5.5 — Невытесняющего алгоритма SJN

Время	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
p0	Г	Г	Г	Г	И	И	И	И	И							
p1	Г	И	И	И												
p2	Г	Г	Г	Г	Г	Г	Г	Г	Г	И	И	И	И	И	И	И
p3	И															

Как мы видим, среднее время ожидания для алгоритма SJN составляет (4 + 1 + 9 +0)/4= = 3,5 единицы времени. Легко посчитать, что для алгоритма FCFS при порядке процессов p0, p1, p2, p3 эта величина будет равняться (0 + 5 + 8 + 15)/4 = 7 единицам времени, т. е. будет в два раза больше, чем для алгоритма SJN. Можно показать, что для заданного набора процессов (если в очереди не появляются новые процессы) алгоритм SJN является оптимальным с точки зрения минимизации среднего времени ожидания среди класса невытесняющих алгоритмов.

Для рассмотрения примера вытесняющего SJN планирования мы возьмем ряд процессов p0, p1, p2 и p3 с различными временами CPU burst и различными моментами их появления в очереди процессов, готовых к исполнению (см. табл. 5.6.).

Таблица 5.6 — CPU burst для SJN

Процесс	Время появления в очереди очередного CPU burst	Продолжительность
p0	0	6
p1	2	2
p2	6	7
p3	0	5

В начальный момент времени в состоянии готовность находятся только два процесса, p0 и p3. Меньшее время очередного CPU burst оказывается у процесса p3, поэтому он и выбирается для исполнения (см. таблицу 5.7.). По прошествии 2 единиц времени в систему поступает процесс p1. Время его CPU burst меньше, чем остаток CPU burst у процесса p3, который вытесняется из состояния исполнение и переводится в состояние готовность. По прошествии еще 2 единиц времени процесс p1 завершается, и для исполнения вновь выбирается процесс p3. В момент времени t = 6 в очереди процессов, готовых к исполнению, появляется процесс p2, но поскольку ему для работы нужно 7 единиц времени, а процессу p3 осталось трудиться всего 1 единицу времени, то процесс p3 остается в состоянии исполнение. После его завершения в момент времени t = 7 в очереди находятся процессы p0 и p2, из которых выбирается процесс p0. Наконец, последним получит возможность выполняться процесс p2.

Таблица 5.7 — Невытесняющего алгоритма SJN

Время	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
p0	Г	Г	Г	Г	Г	Г	Г	И	И	И	И	И	И							
p1			И	И																
p2							Г	Г	Г	Г	Г	Г	Г	И	И	И	И	И	И	И
p3	И	И	Г	Г	И	И	И													

Основную сложность при реализации алгоритма SJN представляет невозможность точного знания продолжительности очередного CPU burst для исполняющихся процессов. В пакетных системах количество процессорного времени, необходимое заданию для выполнения, указывает пользователь при формировании задания. Мы можем брать эту величину для осуществления долгосрочного SJN-планирования. Если пользователь укажет больше времени, чем ему нужно, он будет ждать результата дольше, чем мог бы, так как задание будет загружено в систему позже. Если же он укажет меньшее количество времени, задача может не досчитаться до конца. Таким образом, в пакетных системах решение задачи оценки времени использования процессора перекладывается на плечи пользователя. При краткосрочном планировании мы можем делать только прогноз длительности следующего CPU burst, исходя из предыстории работы процесса.

SRN

SRN – Shortest Remain Next. Данный алгоритм является вариацией предыдущего. В соответствии с этим алгоритмом планировщик выбирает всегда процесс с наименьшим оставшимся временем выполнения. При поступлении новой задачи её полное время выполнения сравнивается с оставшимся временем выполнения текущего процесса.

Алгоритм позволяет более быстро обслуживать короткие запросы. Недостаток тот же, что и у предыдущего алгоритма – нужно знать заранее, сколько ещё времени будет выполняться каждый конкретный процесс.

5.5 ПЛАНИРОВАНИЕ В ИНТЕРАКТИВНЫХ СИСТЕМАХ

RR

Модификацией алгоритма FCFS является алгоритм, получивший название Round Robin (Round Robin – это вид детской карусели в США) или сокращенно RR. По сути дела, это тот же самый алгоритм, только реализованный в режиме вытесняющего планирования. Можно представить себе все множество готовых процессов организованным циклически – процессы сидят на карусели. Карусель вращается так, что каждый процесс находится около процессора небольшой фиксированный квант времени, обычно 10 – 100 миллисекунд. Пока процесс находится рядом с процессором, он получает процессор в свое распоряжение и может исполняться (см. рисунок 5.2).

Рисунок 5.2 — Процессы на карусели

Реализуется такой алгоритм так же, как и предыдущий, с помощью организации процессов, находящихся в состоянии готовность, в очередь FIFO. Планировщик выбирает для очередного исполнения процесс, расположенный в начале очереди, и устанавливает таймер для генерации прерывания по истечении определенного кванта времени. При выполнении процесса возможны два варианта.

- Время непрерывного использования процессора, необходимое процессу (остаток текущего CPU burst), меньше или равно продолжительности кванта времени. Тогда процесс по своей воле освобождает процессор до истечения кванта времени, на исполнение поступает новый процесс из начала очереди, и таймер начинает отсчет кванта заново.
- Продолжительность остатка текущего CPU burst процесса больше, чем квант времени. Тогда по истечении этого кванта процесс прерывается таймером и помещается в конец очереди процессов, готовых к исполнению, а процессор выделяется для использования процессу, находящемуся в ее начале.

Рассмотрим предыдущий пример с порядком процессов p0, p1, p2 и величиной кванта времени равной 4. Выполнение этих процессов иллюстрируется таблицей 5.8. Обозначение "И" используется в ней для процесса, находящегося в состоянии исполнение, обозначение "Г" – для процессов в состоянии готовность, пустые ячейки соответствуют завершившимся процессам. Состояния процессов показаны на протяжении соответствующей единицы времени, т. е. колонка с номером 1 соответствует промежутку времени от 0 до 1.

Таблица 5.8 — Алгоритм для RR

Время	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
p0	И	И	И	И	Г	Г	Г	Г	Г	И	И	И	И	И	И	И	И	И
p1	Г	Г	Г	Г	И	И	И	И										
p2	Г	Г	Г	Г	Г	Г	Г	Г	И									

Первым для исполнения выбирается процесс p0. Продолжительность его CPU burst больше, чем величина кванта времени, и поэтому процесс исполняется до истечения кванта, т. е. в течение 4 единиц времени. После этого он помещается в конец очереди готовых к исполнению процессов, которая принимает вид p1, p2, p0. Следующим начинает выполняться процесс p1. Время его исполнения совпадает с величиной выделенного кванта, поэтому процесс работает до своего завершения. Теперь очередь процессов в состоянии готовность состоит из двух процессов, p2 и p0. Процессор выделяется процессу p2. Он завершается до истечения отпущенного ему процессорного времени, и очередные кванты отмеряются процессу p0 – единственному не закончившему к этому моменту свою работу. Время ожидания для процесса p0 (количество символов "Г" в соответствующей строке) составляет 5 единиц времени, для процесса p1 – 4 единицы времени, для процесса p2 – 8 единиц времени. Таким образом, среднее время ожидания для этого алгоритма получается равным (5 + 4 + 8)/3 = 5,6(6) единицы времени. Полное время выполнения для процесса p0 (количество непустых столбцов в соответствующей строке) составляет 18 единиц времени, для процесса p1 – 8 единиц, для процесса p2 – 9 единиц. Среднее полное время выполнения оказывается равным (18 + 8 + 9)/3 = 11,6(6) единицы времени.

Легко увидеть, что среднее время ожидания и среднее полное время выполнения для обратного порядка процессов не отличаются от соответствующих времен для алгоритма FCFS и составляют 2 и 6 единиц времени соответственно.

На производительность алгоритма RR сильно влияет величина кванта времени. Рассмотрим тот же самый пример с порядком процессов p0, p1, p2 для величины кванта времени, равной 1 (см. таблицу 5.9.). Время ожидания для процесса p0 составит 5 единиц времени, для процесса p1 – тоже 5 единиц, для процесса p2 – 2 единицы. В этом случае среднее время ожидания получается равным (5 + 5 + 2)/3 = 4 единицам времени. Среднее полное время исполнения составит (18 + 9 + 3)/3 = 10 единиц времени.

Таблица 5.9 — Алгоритм для RR

Время	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
p0	И	Г	Г	И	Г	И	Г	И	Г	И	И	И	И	И	И	И	И	И
p1	Г	И	Г	Г	И	Г	И	Г	И									
p2	Г	Г	И															

При очень больших величинах кванта времени, когда каждый процесс успевает завершить свой CPU burst до возникновения прерывания по времени, алгоритм RR вырождается в алгоритм FCFS. При очень малых величинах создается иллюзия того, что каждый из n процессов работает на собственном виртуальном процессоре с производительностью ~ 1/n от производительности реального процессора. Правда, это справедливо лишь при теоретическом анализе при условии пренебрежения временами переключения контекста процессов. В реальных условиях при слишком малой величине кванта времени и, соответственно, слишком частом переключении контекста накладные расходы на переключение резко снижают производительность системы.

Приоритетное планирование

При приоритетном планировании каждому процессу присваивается определенное числовое значение – приоритет, в соответствии с которым ему выделяется процессор. Процессы с одинаковыми приоритетами планируются в порядке FCFS. Для алгоритма SJN в качестве такого приоритета выступает оценка продолжительности следующего CPU burst. Чем меньше значение этой оценки, тем более высокий приоритет имеет процесс.

Алгоритмы назначения приоритетов процессов могут опираться как на внутренние параметры, связанные с происходящим внутри вычислительной системы, так и на внешние по отношению к ней. К внутренним параметрам относятся различные количественные и качественные характеристики процесса такие как: ограничения по времени использования процессора, требования к размеру памяти, число открытых файлов и используемых устройств ввода-вывода, отношение средних продолжительностей I/O burst к CPU burst и т. д. Алгоритмы SJN и гарантированного планирования используют внутренние параметры. В качестве внешних параметров могут выступать важность процесса для достижения каких-либо целей, стоимость оплаченного процессорного времени и другие политические факторы. Высокий внешний приоритет может быть присвоен задаче лектора или того, кто заплатил $100 за работу в течение одного часа.

Планирование с использованием приоритетов может быть как вытесняющим, так и невытесняющим. При вытесняющем планировании процесс с более высоким приоритетом, появившийся в очереди готовых процессов, вытесняет исполняющийся процесс с более

низким приоритетом. В случае невытесняющего планирования он просто становится в начало очереди готовых процессов. Давайте рассмотрим примеры использования различных режимов приоритетного планирования.

Пусть в очередь процессов, находящихся в состоянии готовность, поступают те же процессы, что и в примере для вытесняющего алгоритма SJN, только им дополнительно еще присвоены приоритеты (см. табл. 5.10). В вычислительных системах не существует определенного соглашения, какое значение приоритета – 1 или 4 считать более приоритетным. Во избежание путаницы, во всех наших примерах мы будем предполагать, что большее значение соответствует меньшему приоритету, т. е. наиболее приоритетным в нашем примере является процесс p3, а наименее приоритетным – процесс p0.

Как будут вести себя процессы при использовании невытесняющего приоритетного планирования? Первым для выполнения в момент времени t = 0 выбирается процесс p3, как обладающий наивысшим приоритетом. После его завершения в момент времени t = 5 в очереди процессов, готовых к исполнению, окажутся два процесса p0 и p1. Больший приоритет из них у процесса p1, он и начнет выполняться (см. табл. 5.11). Затем в момент времени t = 8 для исполнения будет избран процесс p2, и лишь потом – процесс p0.

Таблица 5.10 — Приоритеты для процессов

Процесс	Время появления в очереди	Продолжительность очередного CPU burst	Приоритет
p0	0	6	4
p1	2	2	3
p2	6	7	2
p3	0	5	1

Таблица 5.11 — Приоритетное невытесняющее планирование RR

Время	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
p0	Г	Г	Г	Г	Г	Г	Г	Г	Г	Г	Г	Г	Г	Г	И	И	И	И	И	И
p1			Г	Г	Г	И	И													
p2							Г	И	И	И	И	И	И	И						
p3	И	И	И	И	И															

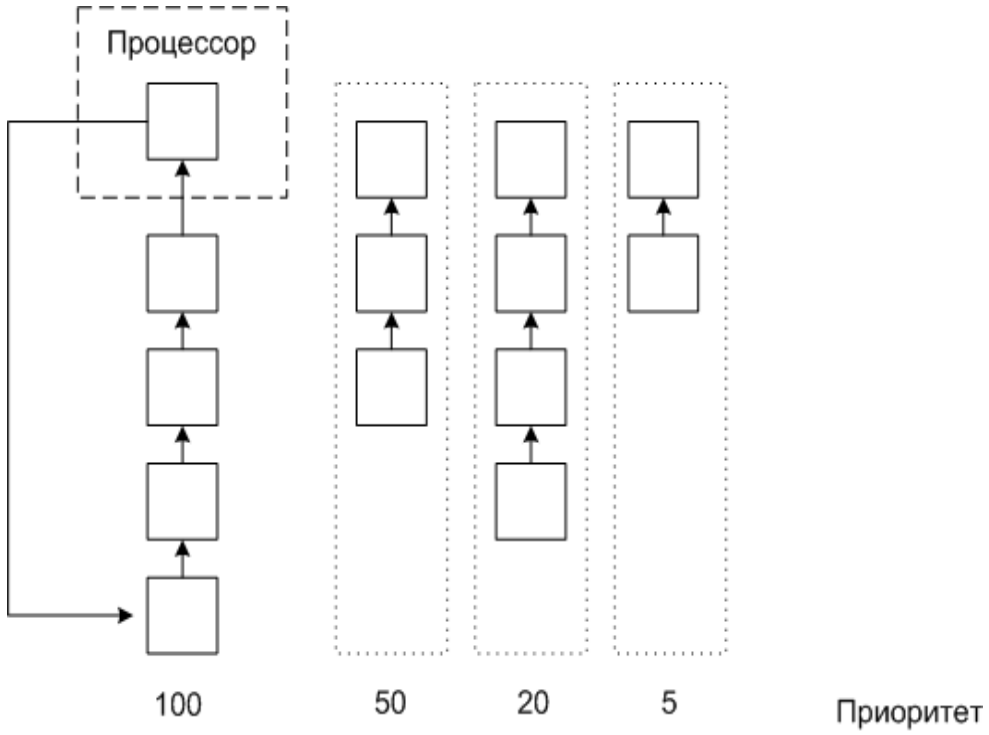
Иным будет предоставление процессора процессам в случае вытесняющего приоритетного планирования (см. табл. 5.12). Первым, как и в предыдущем случае, начнет исполняться процесс p3, а по его окончании – процесс p1. Однако в момент времени t = 6 он будет вытеснен процессом p2 и продолжит свое выполнение только в момент времени t = 13. Последним, как и раньше, будет исполняться процесс p0.

Таблица 5.12 — Приоритетное вытесняющее планирование RR

Время	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
p0	Г	Г	Г	Г	Г	Г	Г	Г	Г	Г	Г	Г	Г	Г	И	И	И	И	И	И
p1			Г	Г	Г	И	Г	Г	Г	Г	Г	Г	Г	И						
p2							И	И	И	И	И	И	И							
p3	И	И	И	И	И															

В рассмотренном выше примере приоритеты процессов с течением времени не изменялись. Такие приоритеты принято называть статическими. Механизмы статической приоритетности легко реализовать, и они сопряжены с относительно небольшими издержками на выбор наиболее приоритетного процесса. Однако статические приоритеты не реагируют на изменения ситуации в вычислительной системе, которые могут сделать желательной корректировку порядка исполнения процессов. Более гибкими являются динамические приоритеты процессов, изменяющие свои значения по ходу исполнения процессов. Начальное значение динамического приоритета, присвоенное процессу, действует в течение лишь короткого периода времени, после чего ему назначается новое, более подходящее значение. Изменение динамического приоритета процесса является единственной операцией над процессами, которую мы до сих пор не рассмотрели. Как правило, изменение приоритета процессов проводится согласованно с совершением каких-либо других операций: при рождении нового процесса, при разблокировке или блокировании процесса, по истечении определенного кванта времени или по завершении процесса. Примерами алгоритмов с динамическими приоритетами являются алгоритм SJN и алгоритм гарантированного планирования. Схемы с динамической приоритетностью гораздо сложнее в реализации и связаны с большими издержками по сравнению со статическими схемами. Однако их использование предполагает, что эти издержки оправдываются улучшением работы системы.

Часто процессы объединяют по приоритетам в группы, и используют приоритетное планирование среди групп, но внутри группы используют циклическое планирование (см. рисунок 5.3).



5.3 — Приоритетное планирование 4-х групп

Главная проблема приоритетного планирования заключается в том, что при ненадлежащем выборе механизма назначения и изменения приоритетов низкоприоритетные процессы могут не запускаться неопределенно долгое время. Обычно случается одно из двух.

Или они все же дожидаются своей очереди на исполнение (в девять часов утра в воскресенье, когда все приличные программисты ложатся спать). Или вычислительную систему приходится выключать, и они теряются (при остановке IBM 7094 в Массачусетском технологическом институте в 1973 году были найдены процессы, запущенные в 1967 году и ни разу с тех пор не исполнявшиеся).

Решение этой проблемы может быть достигнуто с помощью увеличения со временем значения приоритета процесса, находящегося в состоянии готовность. Пусть изначально процессам присваиваются приоритеты от 128 до 255. Каждый раз по истечении определенного промежутка времени значения приоритетов готовых процессов уменьшаются на 1. Процессу, побывавшему в состоянии исполнение, присваивается первоначальное значение приоритета. Даже такая грубая схема гарантирует, что любому процессу в разумные сроки будет предоставлено право на исполнение.

5.6 ПЛАНИРОВАНИЕ В СИСТЕМАХ РЕАЛЬНОГО ВРЕМЕНИ

Системы реального времени делятся на:

- жесткие (жесткие сроки для каждой задачи) - управление движением
- гибкие (нарушение временного графика не желательны, но допустимы) - управление видео и аудио

Внешние события, на которые система должна реагировать, делятся:

- периодические - потоковое видео и аудио
- непериодические (непредсказуемые) - сигнал о пожаре

Что бы систему реального времени можно было планировать, нужно чтобы выполнялось условие:

$$\sum_{i=1}^{m} \frac{T(i)}{P(i)} \leq 1,$$

где m - число периодическихсобытий
i - номер события
P(i) - период поступления события
T(i) - время, которое уходит на обработку события

Т.е. перегруженная система реального времени является не планируемой.

Планирование однородных процессов

В качестве однородных процессов можно рассмотреть видео сервер с несколькими видео потоками (несколько пользователей смотрят фильм).

Т.к. все процессы важны, можно использовать циклическое планирование.

Но так как количество пользователей и размеры кадров могут меняться, для реальных систем он не подходит.

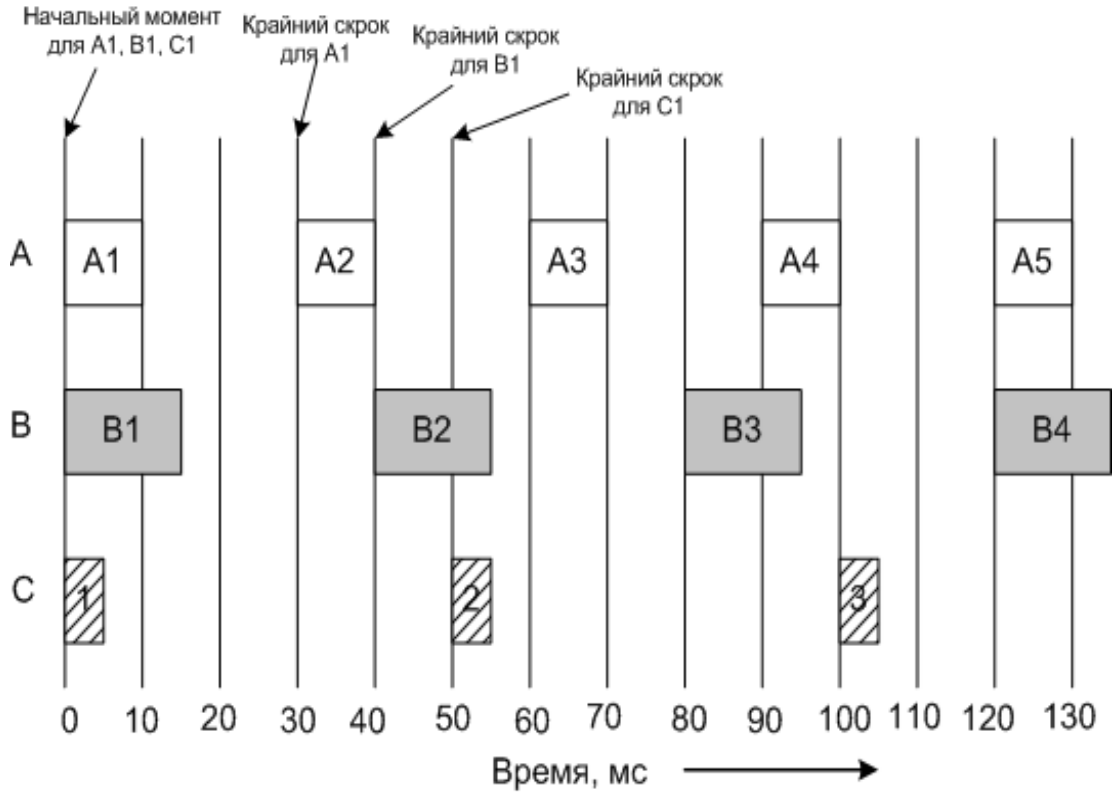
Общее планирование реального времени

Используется модель, когда каждый процесс борется за процессор со своим заданием и графиком его выполнения.

Планировщик должен знать:

- частоту, с которой должен работать каждый процесс
- объем работ, который ему предстоит выполнить
- ближайший срок выполнения очередной порции задания

Рассмотрим пример из трех процессов (см. рисунок 5.4).



5.4 — Три периодических процесса

- Процесс A запускается каждые 30мс, обработка кадра 10мс
- Процесс B частота 25 кадров, т.е. каждые 40мс, обработка кадра 15мс
- Процесс C частота 20 кадров, т.е. каждые 50мс, обработка кадра 5мс

Проверяем, можно ли планировать эти процессы.

$10/30+15/40+5/50=0.808<1$

Условие выполняется, планировать можно.

Будем планировать эти процессы статическим(приоритет заранее назначается каждому процессу) и динамическим методами.

Статический алгоритм планирования RMS (Rate Monotonic Scheduling)

Процессы должны удовлетворять условиям:

- Процесс должен быть завершен за время его периода
- Один процесс не должен зависеть от другого
- Каждому процессу требуется одинаковое процессорное время на каждом интервале
- У непериодических процессов нет жестких сроков
- Прерывание процесса происходит мгновенно

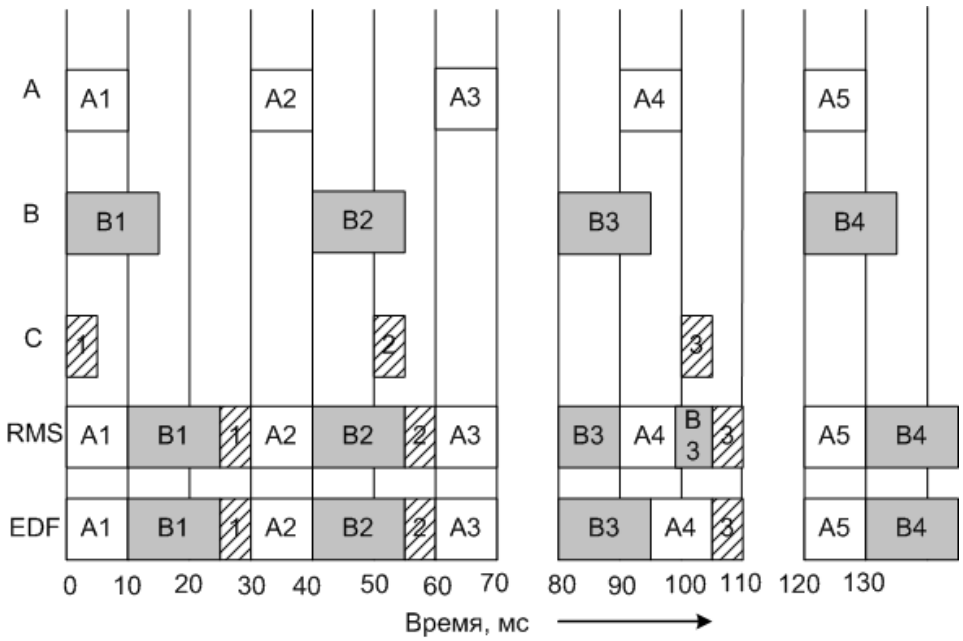
Приоритет в этом алгоритме пропорционален частоте.

Процессу A он равен 33 (частота кадров)

Процессу B он равен 25

Процессу C он равен 20

Процессы выполняются по приоритету (см. рисунок 5.5).



5.5 — Статический алгоритм планирования RMS (Rate Monotonic Scheduling)

Динамический алгоритм планирования EDF (Earliest Deadline First)

Наибольший приоритет выставляется процессу, у которого осталось наименьшее время выполнения.

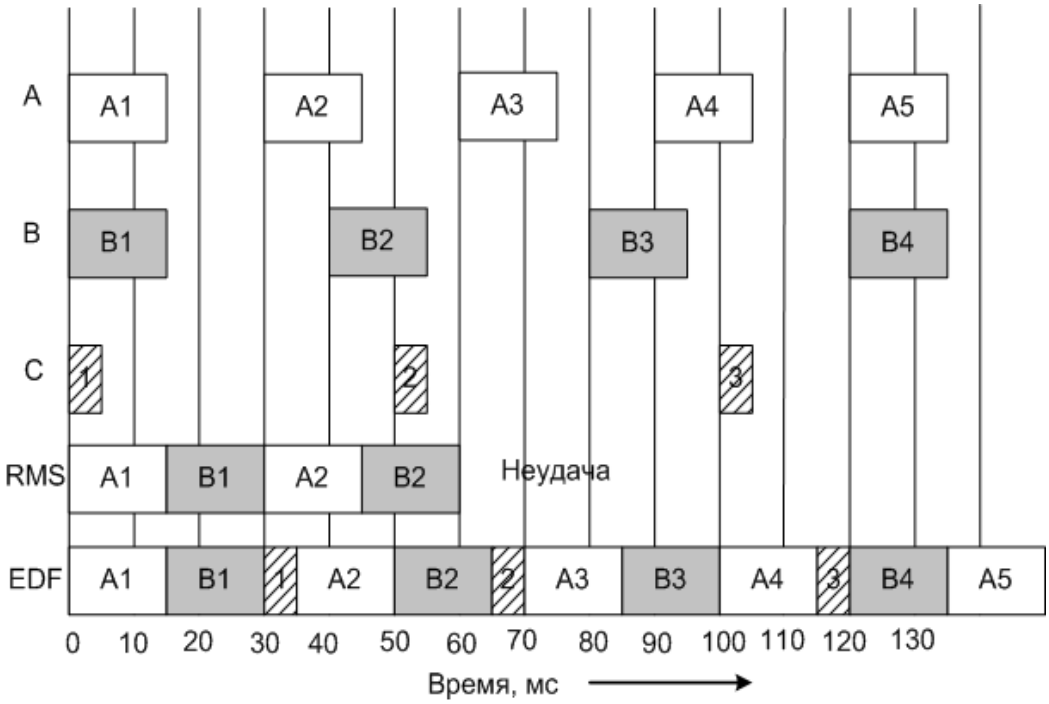
При больших загрузках системы EDF имеет преимущества.

Рассмотрим пример, когда процессу А требуется для обработки кадра - 15мс.

Проверяем, можно ли планировать эти процессы.

15/30+15/40+5/50=0.975<1

Загрузка системы 97.5% (см. рисунок 5.6).



5.6 — Динамический алгоритм планирования EDF (Earliest Deadline First), алгоритм планирования RMS терпит неудачу.

5.7 КАЧЕСТВО ДИСПЕТЧЕРИЗАЦИИ И ГАРАНТИИ ОБСЛУЖИВАНИЯ

Одна из проблем, которая возникает при выборе подходящей дисциплины обслуживания — это гарантия обслуживания. Дело и том, что при некоторых дисциплинах, например при использовании дисциплины абсолютных приоритетов, низкоприоритетные процессы оказываются обделенными многими ресурсами и, прежде всего, процессорным временем. Возникает реальная дискриминации низкоприоритетных .задач, и ряд таких процессов, имеющих к тому же большие потребности в ресурсах, могут очень длительное время откладываться или. в конце концов, вообще могут быть не выполнены. Известны случаи, когда вследствие высокой загрузки вычислительной системы отдельные процессы так и не были выполнены, несмотря на то, что прошло несколько лет (!) с момента их планирования.

Планирование с учетом жестких временных ограничений легко реализовать, организуя очередь готовых к выполнению процессов в порядке возрастания их временных ограничений. Основным недостатком такою простого упорядочения является то, что процесс (за счет других процессов) может быть обслужен быстрее, чем это ему реально необходимо. Для того чтобы избежать этого, проще всего процессорное время выделять все-таки квантами. Гарантировать обслуживание можно следующими тремя способами:

- выделять минимальную долю процессорного времени некоторому классу процессов, если по крайней мере один из них готов к исполнению. Например, можно отводить 20 % от каждых 10 мс процессам реального времени, Л0 % от каждых 2 с — интерактивным процессам и 10 % or каждых 5 мин — пакетным (фоновым) процессам;
- выделять минимальную долю процессорного времени некоторому конкретному процессу, если он готов к выполнению;
- выделять столько процессорного времени некоторому процессу, чтобы он мог выполнять своя вычисления к сроку.

Для сравнения алгоритмов диспетчеризации обычно используются следующие критерии:

- Использование (загрузка) центрального процессора (CPU utilisation). В большинстве персональных систем средняя загрузка процессора не превышает 2-3 %, доходя в моменты выполнения сложных вычислений и до 100 %. В реальных системах, где компьютеры выполняют очень много работы, например, в серверах, загрузка процессора колеблется в пределах 15—10 % для легко загруженного процессора II до 90-100 % — для сильно загруженного процессора.
- Пропускная способность (CPU throughput). Пропускная способность процессора может измеряться количеством щюцессов, которые выполняются к единицу премсни.
- Время оборота (turnaround time). Для некоторых процессов важным критерием яачяется полное время выполнения, то есть интервал от момента появления процесса во входной очереди до момента его завершения. Это время названо временем оборота и включает время ожидания во входной очереди, время ожидания и очереди готовых процессов, иремя ожидания в очередях к оборудованию, время выполнения в процессоре и время ввода/вы иода.
- Время ожидания (waiting Lime). Под временем ожидания понимается суммарное время нахождения процесса в очереди готовых процессов
- Время отклика (response time). Для интерактивных программ важным показателем является время отклика. Очевидно, что простейшая стратегия краткосрочного

планирования должна быть направлена на максимизацию средних значений пропускной способности, времени ожидания и времени отклика.

Правильное планирование процессов сильно влияет на производительность всей системы. Можно выделить следующие главные причины, приводящие к уменьшению производительности системы:

- Накладные расходы на переключение процессора. Они определяются не только переключениями контекстов задач, но и (при переключении на процессы другого приложения) перемещениями страниц виртуально й памяти, а также необходимостью обновления данных в кэше (колы и данные одной задачи, находящиеся » кэше, не нужны другой задаче и будут заменены, что приводит к дополни гельпым задержкам).
- Переключение на другой процесс в тот момент, когда текущий процесс выполняет критическую секцию, а другие процессы активно ожидают входа в свою критическую секцию (см. главу в). В атом случае потери будут особо велики (хотя вероятность прерывания пы пол иония коротких критических секций мала).

В случае использования мультипроцессорных систем применяются следующие методы повышения производительности системы:

- совместное планирование, при котором все потоки одного приложения (неблокнроваиние) одновременно выбираются для выполнения процессорами и одновременно снимаются с них (для сокращения переключений контекста);
- планирование, при котором находящиеся в критической секции задачи не прерываются, а активно ожидающие вхола в критическую секцию задачи не выбираются до тех пор, пока вход в секцию не освободится;
- планирование с учетом так называемых «советов*, программы (по время ее выполнения). Например, в известной своими новациями ОС Mach имелись два класса таких советов (hints) — указания (разной степени категоричности) о снятии текущего процесса с процессора, а также указания о том процессе, который должен быть выбран взамен текущего.

5.8 ПЛАНИРОВАНИЕ ПОТОКОВ

Если процессы делятся на несколько потоков, то реализуется два уровня паралеллизма: на уровне потоков и на уровне процессов. Планирование в таких системах существенно зависит от того, поддерживаются ли потоки на уровне пользователя, на уровне ядра или и те и другие.

Потоки на уровне пользователя

Для начала рассмотрим потоки на уровне пользователя. Поскольку ядро не знает о существовании потоков, оно выполняет обычное планирование, выбирая процесс А и предоставляя ему квант времени. Планировщик потоков внутри процесса выбирает поток, например А1. Поскольку в случае потоков прерывания по таймеру нет, выбранный поток будет работать столько, сколько пожелает. Если он займет весь квант процесса Л, ядро передаст управление другому процессу.

Когда управление снова перейдет к процессу А, поток А1 возобновит работу. Он будет продолжать потреблять все процессорное время, предоставляемое процессу А, пока не закончит свою работу. Впрочем, асоциальное поведение потока А1 на другие процессы не

повлияет. Они будут продолжать получать долю процессорного времени, которую планировщик считает справедливой, независимо от того, что происходит внутри процесса А.

Теперь представим, что потокам процесса А нужно всего лишь 5 мс из отведенного кванта в 50 мс. Соответственно, каждый из них будет выполнять свою небольшую работу и возвращать процессор планировщику потоков. Это приведет к следующей цепочке: А1, А2, А3, А1, А2, А3, А1, А2, А3, А1, прежде чем управление будет передано процессу В.

В качестве алгоритма планирования для системы поддержки исполнения программ можно взять любой из уже рассмотренных нами. Наиболее часто используются алгоритмы циклического и приоритетного планирования. Единственной проблемой является отсутствие таймера, который прерывал бы затянувшуюся работу потока.

Потоки на уровне ядра

Теперь рассмотрим потоки на уровне ядра. В этой ситуации ядро выбирает следующий поток. При этом ядро не обязано принимать во внимание, какой поток принадлежит какому процессу, хотя у него есть такая возможность. Потоку предоставляется квант времени и по истечении этого кванта управление передается другому потоку. В случае кванта в 50 мс и потоков, блокирующихся через 5 мс, цепочка длиной в 30 мс может выглядеть так: А1, В1, А2, В2.А3, В3, что было невозможно в случае потоков на уровне пользователя.

Основное различие между реализацией потоков на уровне пользователя и реализацией их на уровне ядра состоит в производительности. Для переключения потоков на уровне пользователя требуется выполнение всего нескольких машинных команд. Для переключения потоков на уровне ядра нужно выполнить полное переключение контекста с заменой карты памяти и аннулированием кэша, что выполняется на несколько порядков медленнее. С другой стороны, при реализации потоков на уровне пользователя блокировка потока на устройстве ввода-вывода блокирует весь процесс, чего не случается с потоками на уровне ядра.

Поскольку ядро знает, что на переключение от потока процесса Л к потоку процесса В будет затрачено больше ресурсов, чем на передачу управления следующему потоку процесса А (из-за карты памяти и кэша), эта информация может учитываться при принятии решения планирования. Например, при наличии двух одинаково важных потоков, один из которых принадлежит тому же процессу, что и только что блокированный поток, а второй — другому процессу, предпочтение будет отдано первому потоку.

Еще одним важным фактором является возможность совместного использования потоков на уровне пользователя и специализированного планировщика потоков. Рассмотрим, например, web-сервер на рис. 2.7. Пусть один рабочий поток только что заблокирован, а диспетчер и два оставшихся рабочих потока находятся в состоянии готовности. Который из них будет запушен? Система поддержки исполнения программ, которая обладает информацией о задаче каждого потока, выберет следующим диспетчера, чтобы он запустил следующий рабочий поток. Подобная стратегия увеличивает степень параллелизма в среде, где рабочие потоки часто блокируются на обращениях к диску. В случае потоков на уровне ядра оно не знает, чем занимается каждый поток (хотя у них могут быть разные приоритеты). В целом специализированные планировщики потоков лучше управляют приложениями, чем ядро.

6 ПРЕРЫВАНИЯ И СИСТЕМНЫЕ ВЫЗОВЫ
6.1 НАЗНАЧЕНИЕ И КЛАССЫ ПРЕРЫВАНИЙ

Назначение прерываний

Прерывание - это прекращение выполнения текущей команды или текущей последовательности команд для обработки некоторого события специальной программой - обработчиком прерывания, с последующим возвратом к выполнению прерванной программы.

Прерывания являются основной движущей силой любой операционной системы. Отключите систему прерываний - и "жизнь" в операционной системе немедленно остановится. Периодические прерывания от таймера вызывают смену процессов в мультипрограммной ОС, а прерывания от устройств ввода-вывода управляют потоками данных, которыми вычислительная система обменивается с внешним миром.

Как верно было замечено: "Прерывания названы так весьма удачно, поскольку они прерывают нормальную работу системы". Другими словами, система прерываний переводит процессор на выполнение потока команд, отличного от того, который выполнялся до сих пор, с последующим возвратом к исходному коду. Из сказанного можно сделать вывод о том, что механизм прерываний очень похож на механизм выполнения процедур. Это на самом деле так, хотя между этими механизмами имеется важное отличие. Переключение по прерыванию отличается от переключения, которое происходит по команде безусловного или условного перехода, предусмотренной программистом в потоке команд приложения. Переход по команде происходит в заранее определенных программистом точках программы в зависимости от исходных данных, обрабатываемых программой. Прерывание же происходит в произвольной точке потока команд программы, которую программист не может прогнозировать. Прерывание возникает либо в зависимости от внешних по отношению к процессу выполнения программы событий, либо при появлении непредвиденных аварийных ситуаций в процессе выполнения данной программы. Сходство же прерываний с процедурами состоит в том, что в обоих случаях выполняется некоторая подпрограмма, обрабатывающая специальную ситуацию, а затем продолжается выполнение основной ветви программы.

В зависимости от источника прерывания делятся на три больших класса:

- внешние;
- внутренние;
- программные.

Прерывания обычно обрабатываются модулями операционной системы, так как действия, выполняемые по прерыванию, относятся к управлению разделяемыми ресурсами вычислительной системы - принтером, диском, таймером, процессором и т. п. Процедуры, вызываемые по прерываниям, обычно называют обработчиками прерываний, или процедурами обслуживания прерываний (Interrupt Service Routine, ISR). Аппаратные прерывания обрабатываются драйверами соответствующих внешних устройств, исключе-ния - специальными модулями ядра, а программные прерывания - процедурами ОС, обслуживающими системные вызовы. Кроме этих модулей в операционной системе может находиться так называемый диспетчер прерываний, который координирует работу отдельных обработчиков прерываний.

Внешние прерывания

Внешние прерывания могут возникать в результате действий пользователя или оператора за терминалом, или же в результате поступления сигналов от аппаратных устройств

- сигналов завершения операций ввода-вывода, вырабатываемых контроллерами внешних устройств компьютера, такими как принтер или накопитель на жестких дисках, или же сигналов от датчиков управляемых компьютером технических объектов. Внешние прерывания называют также аппаратными, отражая тот факт, что прерывание возникает вследствие подачи некоторой аппаратурой (например, контроллером принтера) электрического сигнала, который передается (возможно, проходя через другие блоки компьютера, например контроллер прерываний) на специальный вход прерывания процессора. Данный класс прерываний является асинхронным по отношению к потоку инструкций прерываемой программы. Аппаратура процессора работает так, что асинхронные прерывания возникают между выполнением двух соседних инструкций, при этом система после обработки прерывания продолжает выполнение процесса, уже начиная со следующей инструкции.

Они бывают:

- маскируемые, которые могут быть замаскированы программными средствами компьютера – то есть сделать так, чтобы прерывание не приходило, например прерывание таймера;
- немаскируемые, запрос от которых таким образом замаскирован быть не может.

Внутренние прерывания

Внутренние прерывания, называемые также исключениями (exception), происходят синхронно выполнению программы при появлении аварийной ситуации в ходе исполнения некоторой инструкции программы. Примерами исключений являются деление на нуль, ошибки защиты памяти, обращения по несуществующему адресу, попытка выполнить привилегированную инструкцию в пользовательском режиме и т.п. Исключения возникают непосредственно в ходе выполнения тактов команды ("внутри" выполнения).

В качестве примеров внутренних прерываний можно взять прерывание «деление на ноль» или по «Trap Flag» - флаг в регистре PSW, установка 1 в котором вызывает останов после каждой команды для отладки.

Программные прерывания

Программные прерывания отличаются от предыдущих двух классов тем, что они по своей сути не являются "истинными" прерываниями. Программное прерывание возникает при выполнении особой команды процессора, выполнение которой имитирует прерывание, то есть переход на новую последовательность инструкций. Причины использования программных прерываний вместо обычных инструкций вызова процедур будут изложены ниже, после рассмотрения механизма прерываний.

Пример – вызов прерывания командой INT (например INT 21h или 13h).

6.2 МЕХАНИЗМ ОБРАБОТКИ ПРЕРЫВАНИЙ

Механизм прерываний поддерживается аппаратными средствами компьютера и программными средствами операционной системы. Аппаратная поддержка прерываний имеет свои особенности, зависящие от типа процессора и других аппаратных компонентов, передающих сигнал запроса прерывания от внешнего устройства к процессору (таких, как контроллер внешнего устройства, шины подключения внешних устройств, контроллер прерываний, являющийся посредником между сигналами шины и сигналами процессора). Особенности аппаратной реализации прерываний оказывают влияние на средства программной поддержки прерываний, работающие в составе ОС.

Существуют два основных способа, с помощью которых шины выполняют прерывания: векторный (vectored) и опрашиваемый (polled). В обоих способах процессору предоставляется информация об уровне приоритета прерывания на шине подключения внешних устройств. В случае векторных прерываний в процессор передается также информация о начальном адресе программы обработки возникшего прерывания - обработчика прерываний.

Устройствам, которые используют векторные прерывания, назначается вектор прерываний. Он представляет собой электрический сигнал, выставляемый на соответствующие шины процессора и несущий в себе информацию об определенном, закрепленном за данным устройством номере, который идентифицирует соответствующий обработчик прерываний. Этот вектор может быть фиксированным, конфигурируемым (например, с использованием переключателей) или программируемым. Операционная система может предусматривать процедуру регистрации вектора обработки прерываний для определенного устройства, которая связывает некоторую подпрограмму обработки прерываний с определенным вектором. При получении сигнала запроса прерывания процессор выполняет специальный цикл подтверждения прерывания, в котором устройство должно идентифицировать себя. В течение этого цикла устройство отвечает, выставляя на шину вектор прерываний. Затем процессор использует этот вектор для нахождения обработчика данного прерывания. Примером шины подключения внешних устройств, которая поддерживает векторные прерывания, является шина VMEbus.

При использовании опрашиваемых прерываний процессор получает от запросившего прерывание устройства только информацию об уровне приоритета прерывания (например, номере IRQ на шине ISA или номере IPL на шине SBus компьютеров SPARC). С каждым уровнем прерываний может быть связано несколько устройств и соответственно несколько программ - обработчиков прерываний. При возникновении прерывания процессор должен определить, какое устройство из тех, которые связаны с данным уровнем прерываний, действительно запросило прерывание. Это достигается вызовом всех обработчиков прерываний для данного уровня приоритета, пока один из обработчиков не подтвердит, что прерывание пришло от обслуживаемого им устройства. Если же с каждым уровнем прерываний связано только одно устройство, то определение нужной программы обработки прерывания происходит немедленно, как и при векторном прерывании. Опрашиваемые прерывания поддерживают шины ISA, EISA, MCA, PCI и Sbus.

Механизм прерываний некоторой аппаратной платформы может сочетать векторный и опрашиваемый типы прерываний. Типичным примером такой реализации является платформа персональных компьютеров на основе процессоров Intel Pentium. Шины PCI, ISA, EISA или MCA, используемые в этой платформе в качестве шин подключения внешних устройств, поддерживают механизм опрашиваемых прерываний. Контроллеры периферийных устройств выставляют на шину не вектор, а сигнал запроса прерывания определенного уровня IRQ. Однако в процессоре Pentium система прерываний является векторной. Вектор прерываний в процессор Pentium поставляет контроллер прерываний, который отображает поступающий от шины сигнал IRQ на определенный номер вектора.

Вектор прерываний, передаваемый в процессор, представляет собой целое число в диапазоне от 0 до 255, указывающее на одну из 256 программ обработки прерываний, адреса которых хранятся в таблице обработчиков прерываний.

Стоит отметить, что само название «вектор прерывания» употребляется в двух вариантах. Это и адрес обработчика в таблице прерываний (состоит из двух слов, то есть

четырёх байт, для регистров IP и CS), и поэтому называется вектором в отличие от скалярных значений. А также вектором чаще называется индекс прерывания в этой таблице, как в данном случае.

В том случае, когда к каждой линии IRQ подключается только одно устройство, процедура обработки прерываний работает так, как если бы система прерываний была чисто векторной, то есть процедура не выполняет никаких дополнительных опросов для выяснения того, какое именно устройство запросило прерывание. Однако при совместном использовании одного уровня IRQ несколькими устройствами программа обработки прерываний должна работать в соответствии со схемой опрашиваемых прерываний, то есть дополнительно выполнить опрос всех устройств, подключенных к данному уровню IRQ.

Обобщенно последовательность действий аппаратных и программных средств по обработке прерывания можно описать следующим образом.

1. При возникновении сигнала (для аппаратных прерываний) или условия (для внутренних прерываний) прерывания происходит первичное аппаратное распознавание типа прерывания. Если прерывания данного типа в настоящий момент запрещены (приоритетной схемой или механизмом маскирования), то процессор продолжает поддерживать естественный ход выполнения команд. В противном случае в зависимости от поступившей в процессор информации (уровень прерывания, вектор прерывания или тип условия внутреннего прерывания) происходит автоматический вызов процедуры обработки прерывания, адрес которой находится в специальной таблице операционной системы, размещаемой либо в регистрах процессора, либо в определенном месте оперативной памяти.

2. Автоматически сохраняется некоторая часть контекста прерванного потока, которая позволит ядру возобновить исполнение потока процесса после обработки прерывания. В это подмножество обычно включаются значения счетчика команд, слова состояния машины, хранящего признаки основных режимов работы процессора (пример такого слова - регистр EFLAGS в Intel Pentium), а также нескольких регистров общего назначения, которые требуются программе обработки прерывания. Может быть сохранен и полный контекст процесса, если ОС обслуживает данное прерывание со сменой процесса. Однако в общем случае это не обязательно, часто обработка прерываний выполняется без вытеснения текущего процесса .

3. Одновременно с загрузкой адреса процедуры обработки прерываний в счетчик команд может автоматически выполняться загрузка нового значения слова состояния машины (или другой системной структуры, например селектора кодового сегмента в процессоре Pentium), которое определяет режимы работы процессора при обработке прерывания, в том числе работу в привилегированном режиме. В некоторых моделях про-цессоров переход в привилегированный режим за счет смены состояния машины при обработке прерывания является единственным способом смены режима. Прерывания практически во всех мультипрограммных ОС обрабатываются в привилегированном режиме модулями ядра, так как при этом обычно нужно выполнить ряд критических операций, от которых зависит жизнеспособность системы, - управлять внешними устройствами, перепланировать потоки и т.п.

4. Временно запрещаются прерывания данного типа, чтобы не образовалась очередь вложенных друг в друга потоков одной и той же процедуры. Детали выполнения этой операции зависят от особенностей аппаратной платформы, например может использоваться механизм маскирования прерываний. Многие процессоры автоматически устанавливают признак

запрета прерываний в начале цикла обработки прерывания, в противном случае это делает программа обработки прерываний.

5. После того как прерывание обработано ядром операционной системы, прерванный контекст восстанавливается и работа потока возобновляется с прерванного места. Часть контекста восстанавливается аппаратно по команде возврата из прерываний (например, адрес следующей команды и слово состояния машины), а часть - программным способом, с помощью явных команд извлечения данных из стека. При возврате из прерывания блокировка повторных прерываний данного типа снимается.

6.3 УЧЕТ ПРИОРИТЕТА ПРЕРЫВАНИЙ

Механизм прерываний чаще всего поддерживает приоритезацию и маскирование прерываний. Приоритезация означает, что все источники прерываний делятся на классы и каждому классу назначается свой уровень приоритета запроса на прерывание. Приоритеты могут обслуживаться как относительные и абсолютные. Обслуживание запросов прерываний по схеме с относительными приоритетами заключается в том, что при одновременном поступлении запросов прерываний из разных классов выбирается запрос, имеющий высший приоритет. Однако в дальнейшем при обслуживании этого запроса процедура обработки прерывания уже не откладывается даже в том случае, когда появляются более приоритетные запросы - решение о выборе нового запроса принимается только в момент завершения обслуживания очередного прерывания. Если же более приоритетным прерываниям разрешается приостанавливать работу процедур обслуживания менее приоритетных прерываний, то это означает, что работает схема приоритезации с абсолютными приоритетами.

Если процессор (или компьютер, когда поддержка приоритезации прерываний вынесена во внешний по отношению к процессору блок) работает по схеме с абсолютными приоритетами, то он поддерживает в одном из своих внутренних регистров переменную, фиксирующую уровень приоритета обслуживаемого в данный момент прерывания. При поступлении запроса из определенного класса его приоритет сравнивается с текущим приоритетом процессора, и если приоритет запроса выше, то текущая процедура обработки прерываний вытесняется, а по завершении обслуживания нового прерыва-ния происходит возврат к прерванной процедуре.

Упорядоченное обслуживание запросов прерываний наряду со схемами приоритетной обработки запросов может выполняться механизмом маскирования запросов. Собственно говоря, в описанной схеме абсолютных приоритетов выполняется маскирование - при обслуживании некоторого запроса все запросы с равным или более низким приоритетом маскируются, то есть не обслуживаются. Схема маскирования предполагает возможность временного маскирования прерываний любого класса независимо от уровня приоритета.

6.4 СИСТЕМНЫЕ ВЫЗОВЫ

Системный вызов — обращение прикладной программы к ядру операционной системы для выполнения какой-либо операции.

Системный вызов позволяет приложению обратиться к операционной системе с просьбой выполнить то или иное действие, оформленное как процедура (или набор процедур) кодового сегмента ОС. Для прикладного программиста операционная система выглядит как некая библиотека, предоставляющая некоторый набор полезных функций, с помощью которых

можно упростить прикладную программу или выполнить действия, запрещенные в пользовательском режиме, например обмен данными с устройством ввода-вывода.

Реализация системных вызовов должна удовлетворять следующим требованиям:

- обеспечивать переключение в привилегированный режим;
- обладать высокой скоростью вызова процедур ОС;
- обеспечивать по возможности единообразное обращение к системным вызовам для всех аппаратных платформ, на которых работает ОС;
- допускать легкое расширение набора системных вызовов;
- обеспечивать контроль со стороны ОС за корректным использованием системных вызовов.

Первое требование для большинства аппаратных платформ может быть выполнено только с помощью механизма программных прерываний. Поэтому будем считать, что остальные требования нужно обеспечить именно для такой реализации системных вызовов. Как это обычно бывает, некоторые из этих требований взаимно противоречивы.

Для обеспечения высокой скорости было бы полезно использовать векторные свойства системы программных прерываний, имеющиеся во многих процессорах, то есть закрепить за каждым системным вызовом определенное значение вектора. Приложение при таком способе вызова непосредственно указывает в аргументе запроса значение вектора, после чего управление немедленно передается требуемой процедуре операционной системы. Однако этот децентрализованный способ передачи управления привязан к особенностям аппаратной платформы, а также не позволяет операционной системе легко модифицировать набор системных вызовов и контролировать их использование. Например, в процессоре Pentium количество системных вызовов определяется количеством векторов прерываний, выделенных для этой цели из общего пула в 256 элементов (часть которых используется под аппаратные прерывания и обработку исключений). Добавление нового системного вызова требует от системного программиста тщательного поиска свободного элемента в таблице прерываний, которого к тому же на каком-то этапе развития ОС может и не оказаться.

6.5 СХЕМА ОБРАБОТКИ СИСТЕМНЫХ ВЫЗОВОВ

В большинстве ОС системные вызовы обслуживаются по централизованной схеме, основанной на существовании диспетчера системных вызовов. При любом системном вызове приложение выполняет программное прерывание с определенным и единственным номером вектора. Например, ОС Linux использует для системных вызовов команду INT 80h, а ОС Windows NT (при работе на платформе Pentium) - INT 2Eh. Перед выполнением программного прерывания приложение тем или иным способом передает операционной системе номер системного вызова, который является индексом в таблице адресов процедур ОС, реализующих системные вызовы. Способ передачи зависит от реализации, например номер можно поместить в определенный регистр общего назначения процессора или передать через стек (в этом случае после прерывания и перехода в привилегированный режим их нужно будет скопировать в системный стек из пользовательского, это действие в некоторых процессорах автоматизировано). Также некоторым способом передаются аргументы системного вызова, они могут как помещаться в регистры общего назначения, так и передаваться через стек или массив, находящийся в оперативной памяти. Массив удобен при большом объеме данных,

передаваемых в качестве аргументов, при этом в регистре общего назначения указывается адрес этого массива.

Диспетчер системных вызовов обычно представляет собой простую программу, которая сохраняет содержимое регистров процессора в системном стеке (поскольку в результате программного прерывания процессор переходит в привилегированный режим), проверяет, попадает ли запрошенный номер вызова в поддерживаемый ОС диапазон (то есть не выходит ли номер за границы таблицы) и передает управление процедуре ОС, адрес которой задан в таблице адресов системных вызовов.

Процедура реализации системного вызова извлекает из системного стека аргументы и выполняет заданное действие. Это действие может быть весьма простым, например чтение значения системных часов, так что системный вызов оформляется в виде одной функции. Более сложные системные вызовы, такие как чтение из файла или выделение процессу дополнительного сегмента памяти, требуют обращения основной функции системного вызова к нескольким внутренним процедурам ядра ОС, принадлежащим к различным подсистемам, таким как подсистема ввода-вывода или управления памятью.

После завершения работы системного вызова управление возвращается диспетчеру, при этом он получает также код завершения этого вызова. Диспетчер восстанавливает регистры процессора, помещает в определенный регистр код возврата и выполняет инструкцию возврата из прерывания, которая восстанавливает непривилегированный режим работы процессора.

Для приложения системный вызов внешне ничем не отличается от вызова обычной библиотечной функции языка С, связанной (динамически или статически) с объектным кодом приложения и выполняющейся в пользовательском режиме. И такая ситуация действительно имеет место - для всех системных вызовов в библиотеках, предоставляемых компилятором С, имеются так называемые "заглушки" (в англоязычном варианте используется термин "stub" - остаток, огрызок). Каждая заглушка оформлена как С-функция, при этом она содержит несколько ассемблерных строк, нужных для выполнения инструкции программного прерывания. Таким образом, пользовательская программа вызывает заглушку, а та, в свою очередь, вызывает процедуру ОС.

Для ускорения выполнения некоторых достаточно простых системных вызовов, которым к тому же не требуется работа в привилегированном режиме, требуемая работа полностью выполняется библиотечной функцией, которую несправедливо называть в данном случае заглушкой. Более точно, такая функция не является системным вызовом, а представляет собой "чистую" библиотечную функцию, выполняющую всю свою работу в пользовательском режиме в виртуальном адресном пространстве процесса, но прикладной программист может об этом и не знать - для него системные вызовы и библиотечные функции выглядят единообразно. Прикладной программист имеет дело с набором функций прикладного программного интерфейса - API (например, Win32 или POSIX), - состоящего и из библиотечных функций, часть из которых пользуется для завершения работы системными вызовами, а часть - нет.

Описанный табличный способ организации системных вызовов принят практически во всех операционных системах. Он позволяет легко модифицировать состав системных вызовов, просто добавив в таблицу новый адрес и расширив диапазон допустимых номеров вызовов.

Операционная система может выполнять системные вызовы в синхронном или асинхронном режимах. Синхронный системный вызов означает, что процесс, сделавший такой

вызов, приостанавливается (переводится планировщиком ОС в состояние ожидания) до тех пор, пока системный вызов не выполнит всю требующуюся от него работу. После этого планировщик переводит процесс в состояние готовности и при очередном выполнении процесс гарантированно может воспользоваться результатами завершившегося к этому времени системного вызова. Синхронные вызовы называются также блокирующими, так как вызвавший системное действие процесс блокируется до его завершения.

Асинхронный системный вызов не приводит к переводу процесса в режим ожидания после выполнения некоторых начальных системных действий, например запуска операции вывода-вывода, управление возвращается прикладному процессу.

Большинство системных вызовов в операционных системах являются синхронными, так как этот режим избавляет приложение от работы по выяснению момента появления результата вызова. Вместе с тем в новых версиях операционных систем количество асинхронных системных вызовов постепенно увеличивается, что дает больше свободы разработчикам сложных приложений. Особенно нужны асинхронные системные вызовы в операционных системах на основе микроядерного подхода, так как при этом в пользовательском режиме работает часть ОС, которым необходимо иметь полную свободу в организации своей работы, а такую свободу дает только асинхронный режим обслуживания вызовов микроядром.

Рассмотрим наиболее часто применяемых системных вызовов стандарта POSIX. В POSIX существует более 100 системных вызовов.

fork - создание нового процесса
exit - завершение процесса
open - открывает файл
close - закрывает файл
read - читает данные из файла в буфер
write - пишет данные из буфера в файл
stat - получает информацию о состоянии файла
mkdir - создает новый каталог
rmdir - удаляет каталог
link - создает ссылку
unlink - удаляет ссылку
mount - монтирует файловую систему
umount - демонтирует файловую систему
chdir - изменяет рабочий каталог

В UNIX вызовы почти один к одному идентичны библиотечным процедурам, которые используются для обращения к системным вызовам.

Рассмотрим интерфейс прикладного программирования для Windows - Win32 API. Win32 API отделен от системных вызовов. Это позволяет в разных версиях менять системные вызовы, не переписывая программы.

Поэтому непонятно является ли вызов системным (выполняется ядром), или он обрабатывается в пространстве пользователя.

В Win32 API существует более 1000 вызовов. Такое количество связано и с тем, что графический интерфейс пользователя UNIX запускается в пользовательском режиме, а в Windows встроен в ядро. Поэтому Win32 API имеет много вызовов для управления окнами, текстом, шрифтами т.д.

Рассмотрим вызовы Win32 API, которые подобны вызовам стандарта POSIX.

CreatProcess (fork) - создание нового процесса
ExitProcess (exit) - завершение процесса
CreatFile (open) - открывает файл
CloseHandle (close) - закрывает файл
ReadFile (read) - читает данные из файла в буфер
WriteFile (write) - пишет данные из буфера в файл
CreatDirectory (mkdir) - создает новый каталог
RemoveDirectory (rmdir) - удаляет каталог
SetCurrentDirectory (chdir) - изменяет рабочий каталог

7 МЕЖПРОЦЕССНОЕ ВЗАИМОДЕЙСТВИЕ
7.1 СРЕДСТВА ВЗАИМОДЕЙСТВИЯ

Для достижения поставленной цели различные процессы (возможно, даже принадлежащие разным пользователям) могут исполняться псевдопараллельно на одной вычислительной системе или параллельно на разных вычислительных системах, взаимодействуя между собой.

Для чего процессам нужно заниматься совместной деятельностью? Какие существуют причины для их кооперации?

- Повышение скорости работы. Пока один процесс ожидает наступления некоторого события (например, окончания операции ввода-вывода), другие могут заниматься полезной работой, направленной на решение общей задачи. В многопроцессорных вычислительных системах программа разбивается на отдельные кусочки, каждый из которых будет исполняться на своем процессоре.
- Совместное использование данных. Различные процессы могут, к примеру, работать с одной и той же динамической базой данных или с разделяемым файлом, совместно изменяя их содержимое.
- Модульная конструкция какой-либо системы. Типичным примером может служить микроядерный способ построения операционной системы, когда различные ее части представляют собой отдельные процессы, взаимодействующие путем передачи сообщений через микроядро.
- Наконец, это может быть необходимо просто для удобства работы пользователя, желающего, например, редактировать и отлаживать программу одновременно. В этой ситуации процессы редактора и отладчика должны уметь взаимодействовать друг с другом.

Процессы не могут взаимодействовать, не общаясь, то есть не обмениваясь информацией. "Общение" процессов обычно приводит к изменению их поведения в зависимости от полученной информации. Если деятельность процессов остается неизменной при любой принятой ими информации, то это означает, что они на самом деле в "общении" не нуждаются. Процессы, которые влияют на поведение друг друга путем обмена информацией, принято называть кооперативными или взаимодействующими процессами, в отличие от независимых процессов, не оказывающих друг на друга никакого воздействия.

Различные процессы в вычислительной системе изначально представляют собой обособленные сущности. Работа одного процесса не должна приводить к нарушению работы другого процесса. Для этого, в частности, разделены их адресные пространства и системные ресурсы, и для обеспечения корректного взаимодействия процессов требуются специальные средства и действия операционной системы. Нельзя просто поместить значение, вычисленное в одном процессе, в область памяти, соответствующую переменной в другом процессе, не предприняв каких-либо дополнительных усилий. Давайте рассмотрим основные аспекты организации совместной работы процессов.

Процессы могут взаимодействовать друг с другом, только обмениваясь информацией. По объему передаваемой информации и степени возможного воздействия на поведение другого процесса все средства такого обмена можно разделить на три категории.

- Сигнальные. Передается минимальное количество информации – вплоть до одного бита, "да" или "нет". Используются, как правило, для извещения процесса о наступлении какого-либо события. Степень воздействия на поведение процесса, получившего

информацию, минимальна. Все зависит от того, знает ли он, что означает полученный сигнал, надо ли на него реагировать и каким образом. Неправильная реакция на сигнал или его игнорирование могут привести к трагическим последствиям. Вспомним профессора Плейшнера из кинофильма "Семнадцать мгновений весны". Сигнал тревоги – цветочный горшок на подоконнике – был ему передан, но профессор проигнорировал его. И к чему это привело?

- Канальные. "Общение" процессов происходит через линии связи, предоставленные операционной системой, и напоминает общение людей по телефону, с помощью записок, писем или объявлений. Объем передаваемой информации в единицу времени ограничен пропускной способностью линий связи. С увеличением количества информации возрастает и возможность влияния на поведение другого процесса.
- Разделяемая память. Два или более процессов могут совместно использовать некоторую область адресного пространства. Созданием разделяемой памяти занимается операционная система (если, конечно, ее об этом попросят). "Общение" процессов напоминает совместное проживание студентов в одной комнате общежития. Возможность обмена информацией максимальна, как, впрочем, и влияние на поведение другого процесса, но требует повышенной осторожности (если вы переложили на другое место вещи вашего соседа по комнате, а часть из них еще и выбросили). Использование разделяемой памяти для передачи/получения информации осуществляется с помощью средств обычных языков программирования, в то время как сигнальным и канальным средствам коммуникации для этого необходимы специальные системные вызовы. Разделяемая память представляет собой наиболее быстрый способ взаимодействия процессов в одной вычислительной системе.

7.2 ЛОГИЧЕСКАЯ ОРГАНИЗАЦИЯ ВЗАИМОДЕЙСТВИЯ

При рассмотрении любого из средств коммуникации нас будет интересовать не их физическая реализация (общая шина данных, прерывания, аппаратно разделяемая память и т. д.), а логическая, определяющая в конечном счете механизм их использования. Некоторые важные аспекты логической реализации являются общими для всех категорий средств связи, некоторые относятся к отдельным категориям. Давайте кратко охарактеризуем основные вопросы, требующие разъяснения при изучении того или иного способа обмена информацией.

Как устанавливается связь?

Могу ли я использовать средство связи непосредственно для обмена информацией сразу после создания процесса или первоначально необходимо предпринять определенные действия для инициализации обмена? Например, для использования общей памяти различными процессами потребуется специальное обращение к операционной системе, которая выделит необходимую область адресного пространства. Но для передачи сигнала от одного процесса к другому никакая инициализация не нужна. В то же время передача информации по линиям связи может потребовать первоначального резервирования такой линии для процессов, желающих обменяться информацией.

К этому же вопросу тесно примыкает вопрос о способе адресации при использовании средства связи. Если я передаю некоторую информацию, я должен указать, куда я ее передаю. Если я желаю получить некоторую информацию, то мне нужно знать, откуда я могу ее получить.

Различают два способа адресации: прямую и непрямую. В случае прямой адресации взаимодействующие процессы непосредственно общаются друг с другом, при каждой

операции обмена данными явно указывая имя или номер процесса, которому информация предназначена или от которого она должна быть получена. Если и процесс, от которого данные исходят, и процесс, принимающий данные, указывают имена своих партнеров по взаимодействию, то такая схема адресации называется симметричной прямой адресацией. Ни один другой процесс не может вмешаться в процедуру симметричного прямого общения двух процессов, перехватить посланные или подменить ожидаемые данные. Если только один из взаимодействующих процессов, например передающий, указывает имя своего партнера по кооперации, а второй процесс в качестве возможного партнера рассматривает любой процесс в системе, например ожидает получения информации от произвольного источника, то такая схема адресации называется асимметричной прямой адресацией.

При непрямой адресации данные помещаются передающим процессом в некоторый промежуточный объект для хранения данных, имеющий свой адрес, откуда они могут быть затем изъяты каким-либо другим процессом. Примером такого объекта может служить обычная доска объявлений или рекламная газета. При этом передающий процесс не знает, как именно идентифицируется процесс, который получит информацию, а принимающий процесс не имеет представления об идентификаторе процесса, от которого он должен ее получить.

При использовании прямой адресации связь между процессами в классической операционной системе устанавливается автоматически, без дополнительных инициализирующих действий. Единственное, что нужно для использования средства связи, – это знать, как идентифицируются процессы, участвующие в обмене данными.

При использовании непрямой адресации инициализация средства связи может и не требоваться. Информация, которой должен обладать процесс для взаимодействия с другими процессами, – это некий идентификатор промежуточного объекта для хранения данных, если он, конечно, не является единственным и неповторимым в вычислительной системе для всех процессов.

Информационная валентность процессов и средств связи

Следующий важный вопрос – это вопрос об информационной валентности связи. Слово "валентность" здесь использовано по аналогии с химией. Сколько процессов может быть одновременно ассоциировано с конкретным средством связи? Сколько таких средств связи может быть задействовано между двумя процессами?

Понятно, что при прямой адресации только одно фиксированное средство связи может быть задействовано для обмена данными между двумя процессами, и только эти два процесса могут быть ассоциированы с ним. При непрямой адресации может существовать более двух процессов, использующих один и тот же объект для данных, и более одного объекта может быть использовано двумя процессами.

К этой же группе вопросов следует отнести и вопрос о направленности связи. Является ли связь однонаправленной или двунаправленной? Под однонаправленной связью мы будем понимать связь, при которой каждый процесс, ассоциированный с ней, может использовать средство связи либо только для приема информации, либо только для ее передачи. При двунаправленной связи каждый процесс, участвующий в общении, может использовать связь и для приема, и для передачи данных. В коммуникационных системах принято называть однонаправленную связь симплексной, двунаправленную связь с поочередной передачей информации в разных направлениях – полудуплексной, а двунаправленную связь с возможностью одновременной передачи информации в разных направлениях – дуплексной. Прямая и непрямая адресация не имеет непосредственного отношения к направленности связи.

7.3 СИГНАЛЬНЫЕ СРЕДСТВА СВЯЗИ

Сигнал дает возможность задаче реагировать на событие, источником которого может быть операционная система или другая задача. Сигналы вызывают прерывание задачи и выполнение заранее предусмотренных действий. Сигналы могут вырабатываться синхронно, то есть как результат работы самого процесса, а могут быть направлены процессу другим процессом, то есть вырабатываться асинхронно. Синхронные сигналы чаще всего приходят от системы прерываний процессора и свидетельствуют о действиях процесса, блокируемых аппаратурой, например деление на нуль, ошибка адресации, нарушение защиты памяти и т.д.

Примером асинхронного сигнала является сигнал с терминала. Во многих ОС предусматривается оперативное снятие процесса с выполнения. Для этого пользователь может нажать некоторую комбинацию клавиш (Ctrl+C, Ctrl+Break), в результате чего ОС вырабатывает сигнал и направляет его активному процессу. Сигнал может поступить в любой момент выполнения процесса (то есть он является асинхронным), требуя от процесса немедленного завершения работы. В данном случае реакцией на сигнал является безусловное завершение процесса.

В системе может быть определен набор сигналов. Программный код процесса, которому поступил сигнал, может либо проигнорировать его, либо прореагировать на него стандартным действием (например, завершиться), либо выполнить специфические действия, определенные прикладным программистом. В последнем случае в программном коде необходимо предусмотреть специальные системные вызовы, с помощью которых операционная система информируется, какую процедуру надо выполнить в ответ на поступление того или иного сигнала.

Сигналы обеспечивают логическую связь между процессами, а также между процессами и пользователями (терминалами). Поскольку посылка сигнала предусматривает знание идентификатора процесса, то взаимодействие посредством сигналов возможно только между родственными процессами, которые могут получить данные об идентификаторах друг друга.

7.4 КАНАЛЬНЫЕ СРЕДСТВА СВЯЗИ

Может ли линия связи сохранять информацию, переданную одним процессом, до ее получения другим процессом или помещения в промежуточный объект? Каков объем этой информации? Иными словами, речь идет о том, обладает ли канал связи буфером и каков объем этого буфера. Здесь можно выделить три принципиальных варианта (см. рисунок 7.1).

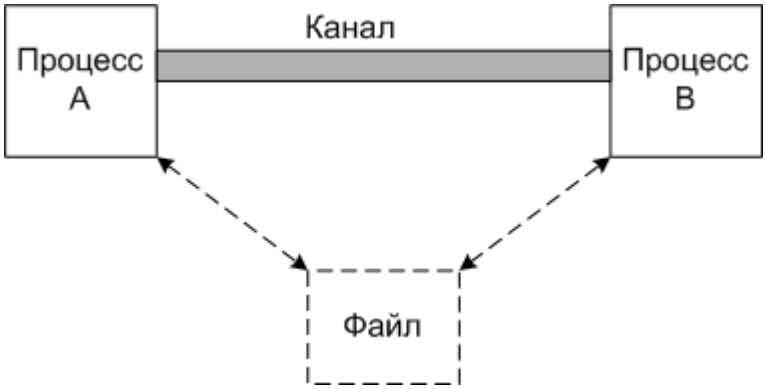


Рисунок 7.1 — Канальные средства связи с буфером

- Буфер нулевой емкости или отсутствует. Никакая информация не может сохраняться на линии связи. В этом случае процесс, посылающий информацию, должен ожидать, пока процесс, принимающий информацию, не соблаговолит ее получить, прежде чем заниматься своими дальнейшими делами (в реальности этот случай никогда не реализуется).
- Буфер ограниченной емкости. Размер буфера равен n, то есть линия связи не может хранить до момента получения более чем n единиц информации. Если в момент передачи данных в буфере хватает места, то передающий процесс не должен ничего ожидать. Информация просто копируется в буфер. Если же в момент передачи данных буфер заполнен или места недостаточно, то необходимо задержать работу процесса отправителя до появления в буфере свободного пространства.
- Буфер неограниченной емкости. Теоретически это возможно, но практически вряд ли реализуемо. Процесс, посылающий информацию, никогда не ждет окончания ее передачи и приема другим процессом.

При использовании канального средства связи с непрямой адресацией под емкостью буфера обычно понимается количество информации, которое может быть помещено в промежуточный объект для хранения данных.

Существует две модели передачи данных по каналам связи – поток ввода-вывода и сообщения. При передаче данных с помощью потоковой модели операции передачи/приема информации вообще не интересуются содержимым данных. Процесс, прочитавший 100 байт из линии связи, не знает и не может знать, были ли они переданы одновременно, т. е. одним куском или порциями по 20 байт, пришли они от одного процесса или от разных. Данные представляют собой простой поток байтов, без какой-либо их интерпретации со стороны системы. Примерами потоковых каналов связи могут служить pipe и FIFO, описанные ниже.

Программные каналы

Одним из наиболее простых способов передачи информации между процессами по линиям связи является передача данных через pipe (канал, трубу или, как его еще называют в литературе, конвейер). Представим себе, что у нас есть некоторая труба в вычислительной системе, в один из концов которой процессы могут "сливать" информацию, а из другого конца принимать полученный поток. Такой способ реализует потоковую модель ввода/вывода. Информацией о расположении трубы в операционной системе обладает только процесс, создавший ее. Этой информацией он может поделиться исключительно со своими наследниками – процессами-детьми и их потомками. Поэтому использовать pipe для связи между собой могут только родственные процессы, имеющие общего предка, создавшего данный канал связи.

Именованные каналы

Если разрешить процессу, создавшему трубу, сообщать о ее местонахождении в системе другим процессам, сделав вход и выход трубы каким-либо образом видимыми для всех остальных, например, зарегистрировав ее в операционной системе под определенным именем, мы получим объект, который принято называть FIFO или именованный pipe. Именованный pipe может использоваться для организации связи между любыми процессами в системе.

Сообщения

В модели сообщений процессы налагают на передаваемые данные некоторую структуру. Весь поток информации они разделяют на отдельные сообщения, вводя между

данными, по крайней мере, границы сообщений. Примером границ сообщений являются точки между предложениями в сплошном тексте или границы абзаца. Кроме того, к передаваемой информации могут быть присоединены указания на то, кем конкретное сообщение было послано и для кого оно предназначено. Примером указания отправителя могут служить подписи под эпиграфами в книге. Все сообщения могут иметь одинаковый фиксированный размер или могут быть переменной длины (см. рисунок 7.2). В вычислительных системах используются разнообразные средства связи для передачи сообщений: очереди сообщений, сокеты и т. д.

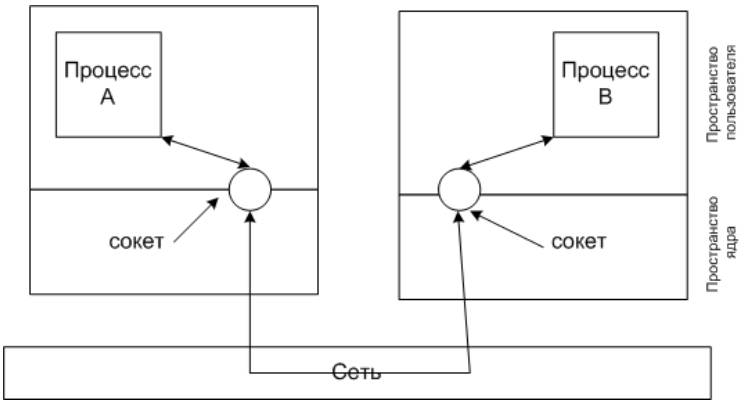


Рисунок 7.2 — Канальные средства связи «Сообщения»

И потоковые линии связи, и каналы сообщений всегда имеют буфер конечной длины. Когда мы будем говорить о емкости буфера для потоков данных, мы будем измерять ее в байтах. Когда мы будем говорить о емкости буфера для сообщений, мы будем измерять ее в сообщениях.

Для прямой и непрямой адресации достаточно двух примитивов, чтобы описать передачу сообщений по линии связи – send и receive. В случае прямой адресации мы будем обозначать их так:

send(P, message) – послать сообщение message процессу P;

receive(Q, message) – получить сообщение message от процесса Q.

В случае непрямой адресации мы будем обозначать их так:

send(A, message) – послать сообщение message в почтовый ящик A;

receive(A, message) – получить сообщение message из почтового ящика A.

Примитивы send и receive уже имеют скрытый от наших глаз механизм взаимоисключения. Более того, в большинстве систем они уже имеют и скрытый механизм блокировки при чтении из пустого буфера и при записи в полностью заполненный буфер. Реализация решения задачи producer-consumer для таких примитивов становится неприлично тривиальной. Надо отметить, что, несмотря на простоту использования, передача сообщений в пределах одного компьютера происходит существенно медленнее, чем работа с семафорами и мониторами.

8 АЛГОРИТМЫ СИНХРОНИЗАЦИИ

В предыдущей лекции мы говорили о внешних проблемах кооперации, связанных с организацией взаимодействия процессов со стороны операционной системы. Предположим, что надежная связь процессов организована, и они умеют обмениваться информацией. Нужно ли нам предпринимать еще какие-либо действия для организации правильного решения задачи взаимодействующими процессами? Нужно ли изменять их внутреннее поведение? Разъяснению этих вопросов и посвящена данная лекция.

Interleaving, race condition и взаимоисключения

Давайте временно отвлечемся от операционных систем, процессов и нитей исполнения и поговорим о некоторых "активностях". Под активностями мы будем понимать последовательное выполнение ряда действий, направленных на достижение определенной цели. Активности могут иметь место в программном и техническом обеспечении, в обычной деятельности людей и животных. Мы будем разбивать активности на некоторые неделимые, или атомарные, операции. Например, активность "приготовление бутерброда" можно разбить на следующие атомарные операции:

1. Отрезать ломтик хлеба.
2. Отрезать ломтик колбасы.
3. Намазать ломтик хлеба маслом.
4. Положить ломтик колбасы на подготовленный ломтик хлеба.

Неделимые операции могут иметь внутренние невидимые действия (взять батон хлеба в левую руку, взять нож в правую руку, произвести отрезание). Мы же называем их неделимыми потому, что считаем выполняемыми за раз, без прерывания деятельности.

Пусть имеется две активности

P: a b c

Q: d e f

где a, b, c, d, e, f – атомарные операции. При последовательном выполнении активностей мы получаем такую последовательность атомарных действий:

PQ: a b c d e f

Что произойдет при исполнении этих активностей псевдопараллельно, в режиме разделения времени? Активности могут расслоиться на неделимые операции с различным чередованием, то есть может произойти то, что на английском языке принято называть словом interleaving. Возможные варианты чередования:

a b c d e f
a b d c e f
a b d e c f
a b d e f c
a d b c e f
.....
d e f a b c

Атомарные операции активностей могут чередоваться всевозможными различными способами с сохранением порядка расположения внутри активностей. Так как псевдопараллельное выполнение двух активностей приводит к чередованию их неделимых операций, результат псевдопараллельного выполнения может отличаться от результата последовательного выполнения. Рассмотрим пример. Пусть у нас имеется две активности P и Q, состоящие из двух атомарных операций каждая:

P: x=2 Q: x=3
y=x-1 y=x+1

Что мы получим в результате их псевдопараллельного выполнения, если переменные x и y являются для активностей общими? Очевидно, что возможны четыре разных набора значений для пары (x, y): (3, 4), (2, 1), (2, 3) и (3, 2). . Мы будем говорить, что набор активностей (например, программ) детерминирован, если всякий раз при псевдопараллельном исполнении для одного и того же набора входных данных он дает одинаковые выходные данные. В противном случае он недетерминирован. Выше приведен пример недетерминированного набора программ. Понятно, что детерминированный набор активностей можно безбоязненно выполнять в режиме разделения времени. Для недетерминированного набора такое исполнение нежелательно.

Можно ли до получения результатов определить, является ли набор активностей детерминированным или нет? Для этого существуют достаточные условия Бернстайна. Изложим их применительно к программам с разделяемыми переменными.

Введем наборы входных и выходных переменных программы. Для каждой атомарной операции наборы входных и выходных переменных – это наборы переменных, которые атомарная операция считывает и записывает. Набор входных переменных программы R(P) (R от слова read) суть объединение наборов входных переменных для всех ее неделимых действий. Аналогично, набор выходных переменных программы W(P) (W от слова write) суть объединение наборов выходных переменных для всех ее неделимых действий. Например, для программы

P: x=u+v
y=x*w

получаем R(P) = {u, v, x, w}, W(P) = {x, y}. Заметим, что переменная x присутствует как в R(P), так и в W(P).

Теперь сформулируем ***условия Бернстайна.***

Если для двух данных активностей P и Q:

- пересечение W(P) и W(Q) пусто,
- пересечение W(P) с R(Q) пусто,
- пересечение R(P) и W(Q) пусто,

тогда выполнение P и Q детерминировано.

Если эти условия не соблюдены, возможно, параллельное выполнение P и Q детерминировано, а может быть, и нет.

Случай двух активностей естественным образом обобщается на их большее количество.

Условия Бернстайна информативны, но слишком жестки. По сути дела, они требуют практически невзаимодействующих процессов. А нам хотелось бы, чтобы детерминированный набор образовывали активности, совместно использующие информацию и обменивающиеся ею. Для этого нам необходимо ограничить число возможных чередований атомарных операций, исключив некоторые чередования с помощью механизмов синхронизации выполнения программ, обеспечив тем самым упорядоченный доступ программ к некоторым данным.

Про недетерминированный набор программ (и активностей вообще) говорят, что он имеет race condition (состояние гонки , состояние состязания). В приведенном выше примере процессы состязаются за вычисление значений переменных x и y.

Задачу упорядоченного доступа к разделяемым данным (устранение race condition) в том случае, когда нам не важна его очередность, можно решить, если обеспечить каждому процессу эксклюзивное право доступа к этим данным. Каждый процесс, обращающийся к разделяемым ресурсам, исключает для всех других процессов возможность одновременного общения с этими ресурсами, если это может привести к недетерминированному поведению набора процессов. Такой прием называется взаимоисключением (mutual exclusion). Если очередность доступа к разделяемым ресурсам важна для получения правильных результатов, то одними взаимоисключениями уже не обойтись, нужна взаимосинхронизация поведения программ.

Критическая секция

Важным понятием при изучении способов синхронизации процессов является понятие критической секции (critical section) программы. Критическая секция – это часть программы, исполнение которой может привести к возникновению race condition для определенного набора программ. Чтобы исключить эффект гонок по отношению к некоторому ресурсу, необходимо организовать работу так, чтобы в каждый момент времени только один процесс мог находиться в своей критической секции, связанной с этим ресурсом. Иными словами, необходимо обеспечить реализацию взаимоисключения для критических секций программ. Реализация взаимоисключения для критических секций программ с практической точки зрения означает, что по отношению к другим процессам, участвующим во взаимодействии, критическая секция начинает выполняться как атомарная операция. Давайте рассмотрим следующий пример, в котором псевдопараллельные взаимодействующие процессы представлены действиями различных студентов (таблица 8.1):

Таблица 8.1 — Действия студентов

Время	Студент 1	Студент 2	Студент 3
17-05	Приходит в комнату		
17-07	Обнаруживает,что хлеба нет		
17-09	Уходит в магазин		
17-11		Приходит в комнату	
17-13		Обнаруживает, что хлеба нет	
17-15		Уходит в магазин	
17-17			Приходит в комнату
17-19			Обнаруживает,что хлеба нет
17-21			Уходит в магазин
17-23	Приходит в магазин		
17-25	Покупает 2 батона на всех		
17-27	Уходит из магазина		
17-29		Приходит в магазин	
17-31		Покупает 2 батона на всех	
17-33		Уходит из магазина	
17-35			Приходит в магазин
17-37			Покупает 2 батона на всех

Время	Студент 1	Студент 2	Студент 3
17-39			Уходит из магазина
17-41	Возвращается в комнату		
17-43			
17-45			
17-47		Возвращается в комнату	
17-49			
17-51			
17-53			Возвращается в комнату

Здесь критический участок для каждого процесса – от операции "Обнаруживает, что хлеба нет" до операции "Возвращается в комнату" включительно. В результате отсутствия взаимоисключения мы из ситуации "Нет хлеба" попадаем в ситуацию "Слишком много хлеба". Если бы этот критический участок выполнялся как атомарная операция – "Достает два батона хлеба", то проблема образования излишков была бы снята.Сделать процесс добывания хлеба атомарной операцией можно было бы следующим образом: перед началом этого процесса закрыть дверь изнутри на засов и уходить добывать хлеб через окно, а по окончании процесса вернуться в комнату через окно и отодвинуть засов. Тогда пока один студент добывает хлеб, все остальные находятся в состоянии ожидания под дверью (таблица 8.2).

Таблица 8.2 — Действия студентов

Время	Студент 1	Студент 2	Студент 3
17-05	Приходит в комнату		
17-07	Достает два батона хлеба		
17-43		Приходит в комнату	
17-47			Приходит в комнату

Итак, для решения задачи необходимо, чтобы в том случае, когда процесс находится в своем критическом участке, другие процессы не могли войти в свои критические участки. Мы видим, что критический участок должен сопровождаться прологом (entry section) – "закрыть дверь изнутри на засов" – и эпилогом (exit section) – "отодвинуть засов", которые не имеют отношения к активности одиночного процесса. Во время выполнения пролога процесс должен, в частности, получить разрешение на вход в критический участок, а во время выполнения эпилога – сообщить другим процессам, что он покинул критическую секцию.

В общем случае структура процесса, участвующего во взаимодействии, может быть представлена следующим образом:

```
while (some condition) {
  entry section
    critical section
  exit section
    remainder section
}
```

Здесь под remainder section понимаются все атомарные операции, не входящие в критическую секцию.

Оставшаяся часть этой лекции посвящена различным способам программной организации пролога и эпилога критического участка в случае, когда очередность доступа к критическому участку не имеет значения.

Программные алгоритмы организации взаимодействия процессов

Требования, предъявляемые к алгоритмам

Организация взаимоисключения для критических участков, конечно, позволит избежать возникновения race condition, но не является достаточной для правильной и эффективной параллельной работы кооперативных процессов. Сформулируем пять условий, которые должны выполняться для хорошего программного алгоритма организации взаимодействия процессов, имеющих критические участки, если они могут проходить их в произвольном порядке.

1. Задача должна быть решена чисто программным способом на обычной машине, не имеющей специальных команд взаимоисключения. При этом предполагается, что основные инструкции языка программирования (такие примитивные инструкции, как load, store, test) являются атомарными операциями.
2. Не должно существовать никаких предположений об относительных скоростях выполняющихся процессов или числе процессоров, на которых они исполняются.
3. Если процесс Pi исполняется в своем критическом участке, то не существует никаких других процессов, которые исполняются в соответствующих критических секциях. Это условие получило название условия взаимоисключения (mutual exclusion).
4. Процессы, которые находятся вне своих критических участков и не собираются входить в них, не могут препятствовать другим процессам входить в их собственные критические участки. Если нет процессов в критических секциях и имеются процессы, желающие войти в них, то только те процессы, которые не исполняются в remainder section, должны принимать решение о том, какой процесс войдет в свою критическую секцию. Такое решение не должно приниматься бесконечно долго. Это условие получило название условия прогресса (progress).
5. Не должно возникать неограниченно долгого ожидания для входа одного из процессов в свой критический участок. От того момента, когда процесс запросил разрешение на вход в критическую секцию, и до того момента, когда он это разрешение получил, другие процессы могут пройти через свои критические участки лишь ограниченное число раз. Это условие получило название условия ограниченного ожидания (bound waiting).

Надо заметить, что описание соответствующего алгоритма в нашем случае означает описание способа организации пролога и эпилога для критической секции.

Запрет прерываний

Наиболее простым решением поставленной задачи является следующая организация пролога и эпилога:

```
while (some condition) {
  запретить все прерывания
    critical section
  разрешить все прерывания
    remainder section
}
```

Поскольку выход процесса из состояния исполнение без его завершения осуществляется по прерыванию, внутри критической секции никто не может вмешаться в его работу. Однако такое решение может иметь далеко идущие последствия, поскольку позволяет процессу пользователя разрешать и запрещать прерывания во всей вычислительной системе. Допустим, что пользователь случайно или по злому умыслу запретил прерывания в системе и зациклил или завершил свой процесс. Без перезагрузки системы в такой ситуации не обойтись.

Тем не менее запрет и разрешение прерываний часто применяются как пролог и эпилог к критическим секциям внутри самой операционной системы, например при обновлении содержимого PCB.

Переменная-замок

В качестве следующей попытки решения задачи для пользовательских процессов рассмотрим другое предложение. Возьмем некоторую переменную, доступную всем процессам, с начальным значением равным 0. Процесс может войти в критическую секцию только тогда, когда значение этой переменной-замка равно 0, одновременно изменяя ее значение на 1 – закрывая замок. При выходе из критической секции процесс сбрасывает ее значение в 0 – замок открывается (как в случае с покупкой хлеба студентами в разделе "Критическая секция").

```
shared int lock = 0;
/* shared означает, что  переменная является разделяемой */
while (some condition) {
  while(lock); lock = 1;
    critical section
  lock = 0;
    remainder section
}
```

К сожалению, при внимательном рассмотрении мы видим, что такое решение не удовлетворяет условию взаимоисключения, так как действие while(lock); lock = 1; не является атомарным. Допустим, процесс P0 протестировал значение переменной lock и принял решение двигаться дальше. В этот момент, еще до присвоения переменной lock значения 1, планировщик передал управление процессу P1. Он тоже изучает содержимое переменной lock и тоже принимает решение войти в критический участок. Мы получаем два процесса, одновременно выполняющих свои критические секции.

Строгое чередование

Попробуем решить задачу сначала для двух процессов. Очередной подход будет также использовать общую для них обоих переменную с начальным значением 0. Только теперь она будет играть не роль замка для критического участка, а явно указывать, кто может следующим войти в него. Для i-го процесса это выглядит так:

```
shared int turn = 0;
while (some condition) {
  while(turn != i);
    critical section
  turn = 1-i;
    remainder section
}
```

Очевидно, что взаимоисключение гарантируется, процессы входят в критическую секцию строго по очереди: P0, P1, P0, P1, P0, ... Но наш алгоритм не удовлетворяет условию прогресса. Например, если значение turn равно 1, и процесс P0 готов войти в критический участок, он не может сделать этого, даже если процесс P1 находится в remainder section.

Флаги готовности

Недостаток предыдущего алгоритма заключается в том, что процессы ничего не знают о состоянии друг друга в текущий момент времени. Давайте попробуем исправить эту ситуацию. Пусть два наших процесса имеют разделяемый массив флагов готовности входа процессов в критический участок shared int ready[2] = {0, 0}.

Когда i-й процесс готов войти в критическую секцию, он присваивает элементу массива ready[i] значение равное 1. После выхода из критической секции он, естественно, сбрасывает это значение в 0. Процесс не входит в критическую секцию, если другой процесс уже готов к входу в критическую секцию или находится в ней.

```
while (some condition) {
  ready[i] = 1;
  while(ready[1-i]);
    critical section
  ready[i] = 0;
    remainder section
}
```

Полученный алгоритм обеспечивает взаимоисключение, позволяет процессу, готовому к входу в критический участок, войти в него сразу после завершения эпилога в другом процессе, но все равно нарушает условие прогресса. Пусть процессы практически одновременно подошли к выполнению пролога. После выполнения присваивания ready[0]=1 планировщик передал процессор от процесса 0 процессу 1, который также выполнил присваивание ready[1]=1. После этого оба процесса бесконечно долго ждут друг друга на входе в критическую секцию. Возникает ситуация, которую принято называть тупиковой (deadlock).

Алгоритм Петерсона

Первое решение проблемы, удовлетворяющее всем требованиям и использующее идеи ранее рассмотренных алгоритмов, было предложено датским математиком Деккером (Dekker). В 1981 году Петерсон (Peterson) предложил более изящное решение. Пусть оба процесса имеют доступ к массиву флагов готовности и к переменной очередности.

```
shared int ready[2] = {0, 0};
shared int turn;
while (some condition) {
  ready[i] = 1;
  turn =1-i;
  while(ready[1-i] && turn == 1-i);
    critical section
  ready[i] = 0;
    remainder section
}
```

При исполнении пролога критической секции процесс Pi заявляет о своей готовности выполнить критический участок и одновременно предлагает другому процессу приступить к его выполнению. Если оба процесса подошли к прологу практически одновременно, то они оба объявят о своей готовности и предложат выполняться друг другу. При этом одно из предложений всегда следует после другого. Тем самым работу в критическом участке продолжит процесс, которому было сделано последнее предложение.

Давайте докажем, что все пять наших требований к алгоритму действительно удовлетворяются.

Удовлетворение требований 1 и 2 очевидно.

Докажем выполнение условия взаимоисключения методом от противного. Пусть оба процесса одновременно оказались внутри своих критических секций. Заметим, что процесс Pi может войти в критическую секцию, только если ready[1-i] == 0 или turn == i. Заметим также, что если оба процесса выполняют свои критические секции одновременно, то значения флагов готовности для обоих процессов совпадают и равны 1. Могли ли оба процесса войти в критические секции из состояния, когда они оба одновременно находились в процессе выполнения цикла while? Нет, так как в этом случае переменная turn должна была бы одновременно иметь значения 0 и 1 (когда оба процесса выполняют цикл, значения переменных измениться не могут). Пусть процесс P0 первым вошел в критический участок, тогда процесс P1 должен был выполнить перед вхождением в цикл while по крайней мере один предваряющий оператор (turn = 0;). Однако после этого он не может выйти из цикла до окончания критического участка процесса P0, так как при входе в цикл ready[0] == 1 и turn == 0, и эти значения не могут измениться до тех пор, пока процесс P0 не покинет свой критический участок. Мы пришли к противоречию. Следовательно, имеет место взаимоисключение.

Докажем выполнение условия прогресса. Возьмем, без ограничения общности, процесс P0. Заметим, что он не может войти в свою критическую секцию только при совместном выполнении условий ready[1] == 1 и turn == 1. Если процесс P1 не готов к выполнению критического участка, то ready[1] == 0, и процесс P0 может осуществить вход. Если процесс P1 готов к выполнению критического участка, то ready[1] == 1 и переменная turn имеет значение 0 либо 1, позволяя процессу P0 либо процессу P1 начать выполнение критической секции. Если процесс P1 завершил выполнение критического участка, то он сбросит свой флаг готовности ready[1] == 0, разрешая процессу P0 приступить к выполнению критической работы. Таким образом, условие прогресса выполняется.

Отсюда же вытекает выполнение условия ограниченного ожидания. Так как в процессе ожидания разрешения на вход процесс P0 не изменяет значения переменных, он сможет начать исполнение своего критического участка после не более чем одного прохода по критической секции процесса P1.

Алгоритм булочной (Bakery algorithm)

Алгоритм Петерсона дает нам решение задачи корректной организации взаимодействия двух процессов. Давайте рассмотрим теперь соответствующий алгоритм для n взаимодействующих процессов, который получил название алгоритм булочной, хотя применительно к нашим условиям его следовало бы скорее назвать алгоритм регистратуры в поликлинике. Основная его идея выглядит так. Каждый вновь прибывающий клиент (он же процесс) получает талончик на обслуживание с номером. Клиент с наименьшим номером на талончике обслуживается следующим. К сожалению, из-за неатомарности операции вычисления следующего номера алгоритм булочной не гарантирует, что у всех процессов

будут талончики с разными номерами. В случае равенства номеров на талончиках у двух или более клиентов первым обслуживается клиент с меньшим значением имени (имена можно сравнивать в лексикографическом порядке). Разделяемые структуры данных для алгоритма – это два массива.

```
shared enum {false, true} choosing[n];
shared int number[n];
```

Изначально элементы этих массивов инициируются значениями false и 0 соответственно. Введем следующие обозначения.

(a,b) < (c,d), если a < c
или если a == c и b < d
max(a0, a1, ..., an) – это число k такое, что
k >= ai для всех i = 0, ...,n

Структура процесса Pi для алгоритма булочной приведена ниже.

```
while (some condition) {
  choosing[i] = true;
  number[i] = max(number[0], ...,
              number[n-1]) + 1;
  choosing[i] = false;
  for(j = 0; j < n; j++){
    while(choosing[j]);
    while(number[j] != 0 && (number[j],j) <
      (number[i],i));
  }
    critical section
  number[i] = 0;
    remainder section
}
```

Доказательство того, что этот алгоритм удовлетворяет условиям 1 – 5, выполните самостоятельно в качестве упражнения.

Аппаратная поддержка взаимоисключений

Наличие аппаратной поддержки взаимоисключений позволяет упростить алгоритмы и повысить их эффективность точно так же, как это происходит и в других областях программирования. Мы уже обращались к общепринятому hardware для решения задачи реализации взаимоисключений, когда говорили об использовании механизма запрета/разрешения прерываний.

Многие вычислительные системы помимо этого имеют специальные команды процессора, которые позволяют проверить и изменить значение машинного слова или поменять местами значения двух машинных слов в памяти, выполняя эти действия как атомарные операции. Давайте обсудим, как концепции таких команд могут использоваться для реализации взаимоисключений.

Команда Test-and-Set (проверить и присвоить 1)

О выполнении команды Test-and-Set, осуществляющей проверку значения логической переменной с одновременной установкой ее значения в 1, можно думать как о выполнении функции.

```
int Test_and_Set (int *target){
  int tmp = *target;
  *target = 1;
  return tmp;
}
```

С использованием этой атомарной команды мы можем модифицировать наш алгоритм для переменной-замка, так чтобы он обеспечивал взаимоисключения.

```
shared int lock = 0;

while (some condition) {
  while(Test_and_Set(&lock));
    critical section
  lock = 0;
    remainder section
}
```

К сожалению, даже в таком виде полученный алгоритм не удовлетворяет условию ограниченного ожидания для алгоритмов. Подумайте, как его следует изменить для соблюдения всех условий.

Команда Swap (обменять значения)

Выполнение команды Swap, обменивающей два значения, находящихся в памяти, можно проиллюстрировать следующей функцией:

```
void Swap (int *a, int *b){
  int tmp = *a;
  *a = *b;
  *b = tmp;
}
```

Применяя атомарную команду Swap, мы можем реализовать предыдущий алгоритм, введя дополнительную логическую переменную key, локальную для каждого процесса:

```
shared int lock = 0;
int key;

while (some condition) {
  key = 1;
  do Swap(&lock,&key);
  while (key);
    critical section
  lock = 0;
    remainder section
}
```

Заключение

Последовательное выполнение некоторых действий, направленных на достижение определенной цели, называется активностью. Активности состоят из атомарных операций, выполняемых неразрывно, как единичное целое. При исполнении нескольких активностей в псевдопараллельном режиме атомарные операции различных активностей могут перемешиваться между собой с соблюдением порядка следования внутри активностей. Это явление получило название interleaving (чередование). Если результаты выполнения нескольких активностей не зависят от варианта чередования, то такой набор активностей называется детерминированным. В противном случае он носит название недетерминированного. Существует достаточное условие Бернстайна для определения детерминированности набора активностей, но оно накладывает очень жесткие ограничения на набор, требуя практически не взаимодействующих активностей. Про недетерминированный набор активностей говорят, что он имеет race condition (условие гонки, состязания). Устранение race condition возможно при ограничении допустимых вариантов чередований атомарных операций с помощью синхронизации поведения активностей. Участки активностей, выполнение которых может привести к race condition, называют критическими участками. Необходимым условием для устранения race condition является организация взаимоисключения на критических участках: внутри соответствующих критических участков не может одновременно находиться более одной активности.

Для эффективных программных алгоритмов устранения race condition помимо условия взаимоисключения требуется выполнение следующих условий: алгоритмы не используют специальных команд процессора для организации взаимоисключений, алгоритмы ничего не знают о скоростях выполнения процессов, алгоритмы удовлетворяют условиям прогресса и ограниченного ожидания. Все эти условия выполняются в алгоритме Петерсона для двух процессов и алгоритме булочной – для нескольких процессов.

Применение специальных команд процессора, выполняющих ряд действий как атомарную операцию, – Test-and-Set, Swap – позволяет существенно упростить алгоритмы синхронизации процессов.

8.1 МЕХАНИЗМЫ СИНХРОНИЗАЦИИ

Рассмотренные в конце предыдущей лекции алгоритмы хотя и являются корректными, но достаточно громоздки и не обладают элегантностью. Более того, процедура ожидания входа в критический участок предполагает достаточно длительное вращение процесса в пустом цикле, то есть напрасную трату драгоценного времени процессора. Существуют и другие серьезные недостатки у алгоритмов, построенных средствами обычных языков программирования. Допустим, что в вычислительной системе находятся два взаимодействующих процесса: один из них – H – с высоким приоритетом, другой – L – с низким приоритетом. Пусть планировщик устроен так, что процесс с высоким приоритетом вытесняет низкоприоритетный процесс всякий раз, когда он готов к исполнению, и занимает процессор на все время своего CPU burst (если не появится процесс с еще большим приоритетом). Тогда в случае, если процесс L находится в своей критической секции, а процесс H, получив процессор, подошел ко входу в критическую область, мы получаем тупиковую ситуацию. Процесс H не может войти в критическую область, находясь в цикле, а процесс L не получает управления, чтобы покинуть критический участок.

Для того чтобы не допустить возникновения подобных проблем, были разработаны различные механизмы синхронизации более высокого уровня. Описанию ряда из них – семафоров, мониторов и сообщений – и посвящена данная лекция.

Семафоры

Одним из первых механизмов, предложенных для синхронизации поведения процессов, стали семафоры, концепцию которых описал Дейкстра (Dijkstra) в 1965 году.

Концепция семафоров

Семафор представляет собой целую переменную, принимающую неотрицательные значения, доступ любого процесса к которой, за исключением момента ее инициализации, может осуществляться только через две атомарные операции: P (от датского слова proberen – проверять) и V (от verhogen – увеличивать). Классическое определение этих операций выглядит следующим образом:

```
P(S):  пока S == 0 процесс блокируется;
       S = S – 1;
V(S):  S = S + 1;
```

Эта запись означает следующее: при выполнении операции P над семафором S сначала проверяется его значение. Если оно больше 0, то из S вычитается 1. Если оно меньше или равно 0, то процесс блокируется до тех пор, пока S не станет больше 0, после чего из S вычитается 1. При выполнении операции V над семафором S к его значению просто прибавляется 1. В момент создания семафор может быть инициализирован любым неотрицательным значением.

Подобные переменные-семафоры могут с успехом применяться для решения различных задач организации взаимодействия процессов. В ряде языков программирования они были непосредственно введены в синтаксис языка (например, в ALGOL-68), в других случаях реализуются с помощью специальных системных вызовов. Соответствующая целая переменная располагается внутри адресного пространства ядра операционной системы. Операционная система обеспечивает атомарность операций P и V, используя, например, метод запрета прерываний на время выполнения соответствующих системных вызовов. Если при выполнении операции P заблокированными оказались несколько процессов, то порядок их разблокирования может быть произвольным, например, FIFO.

Решение проблемы producer-consumer с помощью семафоров

Одной из типовых задач, требующих организации взаимодействия процессов, является задача producer-consumer (производитель-потребитель). Пусть два процесса обмениваются информацией через буфер ограниченного размера. Производитель закладывает информацию в буфер, а потребитель извлекает ее оттуда. На этом уровне деятельность потребителя и производителя можно описать следующим образом.

```
Producer:  while(1) {
             produce_item;
             put_item;
           }
Consumer:  while(1) {
             get_item;
             consume_item;
           }
```

Если буфер заполнен, то производитель должен ждать, пока в нем появится место, чтобы положить туда новую порцию информации. Если буфер пуст, то потребитель должен

дожидаться нового сообщения. Как можно реализовать эти условия с помощью семафоров? Возьмем три семафора: empty, full и mutex. Семафор full будем использовать для гарантии того, что потребитель будет ждать, пока в буфере появится информация. Семафор empty будем использовать для организации ожидания производителя при заполненном буфере, а семафор mutex – для организации взаимоисключения на критических участках, которыми являются действия put_item и get_item (операции "положить информацию" и "взять информацию" не могут пересекаться, так как в этом случае возникнет опасность искажения информации). Тогда решение задачи на С-подобном языке выглядит так:

```
  Semaphore mutex = 1;
  Semaphore empty = N; /* где N – емкость буфера*/
  Semaphore full = 0;
Producer:
  while(1) {
    produce_item;
    P(empty);
    P(mutex);
    put_item;
    V(mutex);
    V(full);
  }
Consumer:
  while(1) {
    P(full);
    P(mutex);
    get_item;
    V(mutex);
    V(empty);
    consume_item;
  }
```

Легко убедиться, что это действительно корректное решение поставленной задачи. Попутно заметим, что семафоры использовались здесь для достижения двух целей: организации взаимоисключения на критическом участке и взаимосинхронизации скорости работы процессов.

Мониторы

Хотя решение задачи producer-consumer с помощью семафоров выглядит достаточно изящно, программирование с их использованием требует повышенной осторожности и внимания, чем отчасти напоминает программирование на языке Ассемблера. Допустим, что в рассмотренном примере мы случайно поменяли местами операции P, сначала выполнив операцию для семафора mutex, а уже затем для семафоров full и empty. Допустим теперь, что потребитель, войдя в свой критический участок (mutex сброшен), обнаруживает, что буфер пуст. Он блокируется и начинает ждать появления сообщений. Но производитель не может войти в критический участок для передачи информации, так как тот заблокирован потребителем. Получаем тупиковую ситуацию.

В сложных программах произвести анализ правильности использования семафоров с карандашом в руках становится очень непросто. В то же время обычные способы отладки программ зачастую не дают результата, поскольку возникновение ошибок зависит от

interleaving атомарных операций, и ошибки могут быть трудновоспроизводимы. Для того чтобы облегчить работу программистов, в 1974 году Хором (Hoare) был предложен механизм еще более высокого уровня, чем семафоры, получивший название мониторов. Мы с вами рассмотрим конструкцию, несколько отличающуюся от оригинальной.

Мониторы представляют собой тип данных, который может быть с успехом внедрен в объектно-ориентированные языки программирования. Монитор обладает собственными переменными, определяющими его состояние. Значения этих переменных извне могут быть изменены только с помощью вызова функций-методов, принадлежащих монитору. В свою очередь, эти функции-методы могут использовать в работе только данные, находящиеся внутри монитора, и свои параметры. На абстрактном уровне можно описать структуру монитора следующим образом:

```
monitor monitor_name {
  описание внутренних переменных ;

  void m1(...){...
  }
  void m2(...){...
  }
  ...
  void mn(...){...
  }

  {
    блок инициализации
    внутренних переменных;
  }
}
```

Здесь функции m1,..., mn представляют собой функции-методы монитора, а блок инициализации внутренних переменных содержит операции, которые выполняются один и только один раз: при создании монитора или при самом первом вызове какой-либо функции-метода до ее исполнения.

Важной особенностью мониторов является то, что в любой момент времени только один процесс может быть активен, т. е. находиться в состоянии готовность или исполнение, внутри данного монитора. Поскольку мониторы представляют собой особые конструкции языка программирования, компилятор может отличить вызов функции, принадлежащей монитору, от вызовов других функций и обработать его специальным образом, добавив к нему пролог и эпилог, реализующий взаимоисключение. Так как обязанность конструирования механизма взаимоисключений возложена на компилятор, а не на программиста, работа программиста при использовании мониторов существенно упрощается, а вероятность возникновения ошибок становится меньше.

Однако одних только взаимоисключений недостаточно для того, чтобы в полном объеме реализовать решение задач, возникающих при взаимодействии процессов. Нам нужны еще и средства организации очередности процессов, подобно семафорам full и empty в предыдущем примере. Для этого в мониторах было введено понятие условных переменных

(condition variables), над которыми можно совершать две операции wait и signal, отчасти похожие на операции P и V над семафорами.

Если функция монитора не может выполняться дальше, пока не наступит некоторое событие, она выполняет операцию wait над какой-либо условной переменной. При этом процесс, выполнивший операцию wait, блокируется, становится неактивным, и другой процесс получает возможность войти в монитор.

Когда ожидаемое событие происходит, другой процесс внутри функции-метода совершает операцию signal над той же самой условной переменной. Это приводит к пробуждению ранее заблокированного процесса, и он становится активным. Если несколько процессов дожидались операции signal для этой переменной, то активным становится только один из них. Что можно предпринять для того, чтобы у нас не оказалось двух процессов, разбудившего и пробужденного, одновременно активных внутри монитора? Хор предложил, чтобы пробужденный процесс подавлял исполнение разбудившего процесса, пока он сам не покинет монитор. Несколько позже Хансен (Hansen) предложил другой механизм: разбудивший процесс покидает монитор немедленно после исполнения операции signal. Мы будем придерживаться подхода Хансена.

Необходимо отметить, что условные переменные, в отличие от семафоров Дейкстры, не умеют запоминать предысторию. Это означает, что операция signal всегда должна выполняться после операции wait. Если операция signal выполняется над условной переменной, с которой не связано ни одного заблокированного процесса, то информация о произошедшем событии будет утеряна. Следовательно, выполнение операции wait всегда будет приводить к блокированию процесса.

Давайте применим концепцию мониторов к решению задачи производитель-потребитель.

```
monitor ProducerConsumer {
  condition full, empty;
  int count;
  void put() {
    if(count == N) full.wait;
    put_item;
    count += 1;
    if(count == 1) empty.signal;
  }
  void get() {
    if (count == 0) empty.wait;
    get_item();
    count -= 1;
    if(count == N-1) full.signal;
  }
  {
    count = 0;
  }
}
```

Producer:

```
while(1) {
  produce_item;
```

```
    ProducerConsumer.put();
  }
```

Consumer:

```
  while(1) {
    ProducerConsumer.get();
    consume_item;
  }
```

Легко убедиться, что приведенный пример действительно решает поставленную задачу.

Реализация мониторов требует разработки специальных языков программирования и компиляторов для них. Мониторы встречаются в таких языках, как параллельный Евклид, параллельный Паскаль, Java и т. д. Эмуляция мониторов с помощью системных вызовов для обычных широко используемых языков программирования не так проста, как эмуляция семафоров. Поэтому можно пользоваться еще одним механизмом со скрытыми взаимоисключениями, механизмом, о котором мы уже упоминали, – передачей сообщений.

Сообщения

Для прямой и непрямой адресации достаточно двух примитивов, чтобы описать передачу сообщений по линии связи – send и receive. В случае прямой адресации мы будем обозначать их так:

send(P, message) – послать сообщение message процессу P;

receive(Q, message) – получить сообщение message от процесса Q.

В случае непрямой адресации мы будем обозначать их так:

send(A, message) – послать сообщение message в почтовый ящик A;

receive(A, message) – получить сообщение message из почтового ящика A.

Примитивы send и receive уже имеют скрытый от наших глаз механизм взаимоисключения. Более того, в большинстве систем они уже имеют и скрытый механизм блокировки при чтении из пустого буфера и при записи в полностью заполненный буфер. Реализация решения задачи producer-consumer для таких примитивов становится неприлично тривиальной. Надо отметить, что, несмотря на простоту использования, передача сообщений в пределах одного компьютера происходит существенно медленнее, чем работа с семафорами и мониторами.

8.2 ТУПИКИ

В предыдущих лекциях мы рассматривали способы синхронизации процессов, которые позволяют процессам успешно кооперироваться. Однако в некоторых случаях могут возникнуть непредвиденные затруднения. Предположим, что несколько процессов конкурируют за обладание конечным числом ресурсов. Если запрашиваемый процессом ресурс недоступен, ОС переводит данный процесс в состояние ожидания. В случае когда требуемый ресурс удерживается другим ожидающим процессом, первый процесс не сможет сменить свое состояние. Такая ситуация называется тупиком (deadlock). Говорят, что в мультипрограммной системе процесс находится в состоянии тупика, если он ожидает события, которое никогда не произойдет. Системная тупиковая ситуация, или "зависание системы", является следствием того, что один или более процессов находятся в состоянии тупика. Иногда

подобные ситуации называют взаимоблокировками. В общем случае проблема тупиков эффективного решения не имеет.

Рассмотрим пример. Предположим, что два процесса осуществляют вывод с ленты на принтер. Один из них успел монополизировать ленту и претендует на принтер, а другой наоборот. После этого оба процесса оказываются заблокированными в ожидании второго ресурса.

Множество процессов находится в тупиковой ситуации, если каждый процесс из множества ожидает события, которое может вызвать только другой процесс данного множества. Так как все процессы чего-то ожидают, то ни один из них не сможет инициировать событие, которое разбудило бы другого члена множества и, следовательно, все процессы будут спать вместе.

Выше приведен пример взаимоблокировки, возникающей при работе с так называемыми выделенными устройствами. Тупики, однако, могут иметь место и в других ситуациях. Например, в системах управления базами данных записи могут быть локализованы процессами, чтобы избежать состояния гонок (см. раздел "Алгоритмы синхронизации"). В этом случае может получиться так, что один из процессов заблокировал записи, необходимые другому процессу, и наоборот. Таким образом, тупики могут иметь место как на аппаратных, так и на программных ресурсах.

Тупики также могут быть вызваны ошибками программирования. Например, процесс может напрасно ждать открытия семафора, потому что в некорректно написанном приложении эту операцию забыли предусмотреть. Другой причиной бесконечного ожидания может быть дискриминационная политика по отношению к некоторым процессам. Однако чаще всего событие, которого ждет процесс в тупиковой ситуации, – освобождение ресурса, поэтому в дальнейшем будут рассмотрены методы борьбы с тупиками ресурсного типа.

Ресурсами могут быть как устройства, так и данные. Некоторые ресурсы допускают разделение между процессами, то есть являются разделяемыми ресурсами. Например, память, процессор, диски коллективно используются процессами. Другие не допускают разделения, то есть являются выделенными, например лентопротяжное устройство. К взаимоблокировке может привести использование как выделенных, так и разделяемых ресурсов. Например, чтение с разделяемого диска может одновременно осуществляться несколькими процессами, тогда как запись предполагает исключительный доступ к данным на диске. Можно считать, что часть диска, куда происходит запись, выделена конкретному процессу. Поэтому в дальнейшем мы будем исходить из предположения, что тупики связаны с выделенными ресурсами , то есть тупики возникают, когда процессу предоставляется эксклюзивный доступ к устройствам, файлам и другим ресурсам.

Традиционная последовательность событий при работе с ресурсом состоит из запроса, использования и освобождения ресурса. Тип запроса зависит от природы ресурса и от ОС. Запрос может быть явным, например специальный вызов request, или неявным – open для открытия файла. Обычно, если ресурс занят и запрос отклонен, запрашивающий процесс переходит в состояние ожидания.

Далее в данной лекции будут рассматриваться вопросы обнаружения, предотвращения, обхода тупиков и восстановления после тупиков. Как правило, борьба с тупиками – очень дорогостоящее мероприятие. Тем не менее для ряда систем, например для систем реального времени, иного выхода нет.

Условия возникновения тупиков

Условия возникновения тупиков были сформулированы Коффманом, Элфиком и Шошани в 1970 г.

1. Условие взаимоисключения (Mutual exclusion). Одновременно использовать ресурс может только один процесс.
2. Условие ожидания ресурсов (Hold and wait). Процессы удерживают ресурсы, уже выделенные им, и могут запрашивать другие ресурсы.
3. Условие неперераспределяемости (No preemtion). Ресурс, выделенный ранее, не может быть принудительно забран у процесса. Освобождены они могут быть только процессом, который их удерживает.
4. Условие кругового ожидания (Circular wait). Существует кольцевая цепь процессов, в которой каждый процесс ждет доступа к ресурсу, удерживаемому другим процессом цепи.

Для образования тупика необходимым и достаточным является выполнение всех четырех условий.

Обычно тупик моделируется циклом в графе, состоящем из узлов двух видов: прямоугольников – процессов и эллипсов – ресурсов. Стрелки, направленные от ресурса к процессу, показывают, что ресурс выделен данному процессу. Стрелки, направленные от процесса к ресурсу, означают, что процесс запрашивает данный ресурс.

Основные направления борьбы с тупиками

Проблема тупиков инициировала много интересных исследований в области информатики. Очевидно, что условие циклического ожидания отличается от остальных. Первые три условия формируют правила, существующие в системе, тогда как четвертое условие описывает ситуацию, которая может сложиться при определенной неблагоприятной последовательности событий. Поэтому методы предотвращения взаимоблокировок ориентированы главным образом на нарушение первых трех условий путем введения ряда ограничений на поведение процессов и способы распределения ресурсов. Методы обнаружения и устранения менее консервативны и сводятся к поиску и разрыву цикла ожидания ресурсов.

Итак, основные направления борьбы с тупиками:

- Игнорирование проблемы в целом.
- Предотвращение тупиков.
- Обнаружение тупиков.
- Восстановление после тупиков.

Игнорирование проблемы тупиков

Простейший подход – не замечать проблему тупиков. Для того чтобы принять такое решение, необходимо оценить вероятность возникновения взаимоблокировки и сравнить ее с вероятностью ущерба от других отказов аппаратного и программного обеспечения. Проектировщики обычно не желают жертвовать производительностью системы или удобством пользователей для внедрения сложных и дорогостоящих средств борьбы с тупиками.

Любая ОС, имеющая в ядре ряд массивов фиксированной размерности, потенциально страдает от тупиков, даже если они не обнаружены. Таблица открытых файлов, таблица процессов, фактически каждая таблица являются ограниченными ресурсами. Заполнение всех записей таблицы процессов может привести к тому, что очередной запрос на создание процесса может быть отклонен. При неблагоприятном стечении обстоятельств несколько процессов могут выдать такой запрос одновременно и оказаться в тупике. Следует ли отказываться от вызова CreateProcess, чтобы решить эту проблему?

Подход большинства популярных ОС (Unix, Windows и др.) состоит в том, чтобы игнорировать данную проблему в предположении, что маловероятный случайный тупик предпочтительнее, чем нелепые правила, заставляющие пользователей ограничивать число процессов, открытых файлов и т. п. Сталкиваясь с нежелательным выбором между строгостью и удобством, трудно найти решение, которое устраивало бы всех.

Способы предотвращения тупиков

Цель предотвращения тупиков – обеспечить условия, исключающие возможность возникновения тупиковых ситуаций. Большинство методов связано с предотвращением одного из условий возникновения взаимоблокировки.

Система, предоставляя ресурс в распоряжение процесса, должна принять решение, безопасно это или нет. Возникает вопрос: есть ли такой алгоритм, который помогает всегда избегать тупиков и делать правильный выбор. Ответ – да, мы можем избегать тупиков, но только если определенная информация известна заранее.

Способы предотвращения тупиков путем тщательного распределения ресурсов. Алгоритм банкира

Можно избежать взаимоблокировки, если распределять ресурсы, придерживаясь определенных правил. Среди такого рода алгоритмов наиболее известен алгоритм банкира, предложенный Дейкстрой, который базируется на так называемых безопасных или надежных состояниях (safe state). Безопасное состояние – это такое состояние, для которого имеется по крайней мере одна последовательность событий, которая не приведет к взаимоблокировке. Модель алгоритма основана на действиях банкира, который, имея в наличии капитал, выдает кредиты.

Суть алгоритма состоит в следующем.

- Предположим, что у системы в наличии n устройств, например лент.
- ОС принимает запрос от пользовательского процесса, если его максимальная потребность не превышает n.
- Пользователь гарантирует, что если ОС в состоянии удовлетворить его запрос, то все устройства будут возвращены системе в течение конечного времени.
- Текущее состояние системы называется надежным, если ОС может обеспечить всем процессам их выполнение в течение конечного времени.
- В соответствии с алгоритмом банкира выделение устройств возможно, только если состояние системы остается надежным.

Рассмотрим пример надежного состояния для системы с 3 пользователями и 11 устройствами, где 9 устройств задействовано, а 2 имеется в резерве. Пусть текущая ситуация такова:

Пользователи	Максимальная потребность в ресурсах	Выделенное пользователям количество ресурсов
Первый	9	6
Второй	10	2
Третий	3	1

Рисунок 8.1 — Пример надежного состояния для системы с 3 пользователями и 11 устройствами

Данное состояние надежно. Последующие действия системы могут быть таковы. Вначале удовлетворить запросы третьего пользователя, затем дождаться, когда он закончит работу и освободит свои три устройства. Затем можно обслужить первого и второго пользователей. То есть система удовлетворяет только те запросы, которые оставляют ее в надежном состоянии, и отклоняет остальные.

Термин ненадежное состояние не предполагает, что обязательно возникнут тупики. Он лишь говорит о том, что в случае неблагоприятной последовательности событий система может зайти в тупик.

Данный алгоритм обладает тем достоинством, что при его использовании нет необходимости в перераспределении ресурсов и откате процессов назад. Однако использование этого метода требует выполнения ряда условий.

- Число пользователей и число ресурсов фиксировано.
- Число работающих пользователей должно оставаться постоянным.
- Алгоритм требует, чтобы клиенты гарантированно возвращали ресурсы.
- Должны быть заранее указаны максимальные требования процессов к ресурсам. Чаще всего данная информация отсутствует.

Наличие таких жестких и зачастую неприемлемых требований может склонить разработчиков к выбору других решений проблемы взаимоблокировки.

Предотвращение тупиков за счет нарушения условий возникновения тупиков

В отсутствие информации о будущих запросах единственный способ избежать взаимоблокировки – добиться невыполнения хотя бы одного из условий раздела "Условия возникновения тупиков".

Нарушение условия взаимоисключения

В общем случае избежать взаимоисключений невозможно. Доступ к некоторым ресурсам должен быть исключительным. Тем не менее некоторые устройства удается обобществить. В качестве примера рассмотрим принтер. Известно, что пытаться осуществлять вывод на принтер могут несколько процессов. Во избежание хаоса организуют промежуточное формирование всех выходных данных процесса на диске, то есть разделяемом устройстве. Лишь один системный процесс, называемый сервисом или демоном принтера, отвечающий за вывод документов на печать по мере освобождения принтера, реально с ним взаимодействует. Эта схема называется спулингом (spooling). Таким образом, принтер становится разделяемым устройством, и тупик для него устранен.

К сожалению, не для всех устройств и не для всех данных можно организовать спулинг. Неприятным побочным следствием такой модели может быть потенциальная тупиковая ситуация из-за конкуренции за дисковое пространство для буфера спулинга. Тем не менее в той или иной форме эта идея применяется часто.

Нарушение условия ожидания дополнительных ресурсов

Условия ожидания ресурсов можно избежать, потребовав выполнения стратегии двухфазного захвата.

- В первой фазе процесс должен запрашивать все необходимые ему ресурсы сразу. До тех пор пока они не предоставлены, процесс не может продолжать выполнение.
- Если в первой фазе некоторые ресурсы, которые были нужны данному процессу, уже заняты другими процессами, он освобождает все ресурсы, которые были ему выделены, и пытается повторить первую фазу.

В известном смысле этот подход напоминает требование захвата всех ресурсов заранее. Естественно, что только специально организованные программы могут быть приостановлены в течение первой фазы и рестартованы впоследствии.

Таким образом, один из способов – заставить все процессы затребовать нужные им ресурсы перед выполнением ("все или ничего"). Если система в состоянии выделить процессу все необходимое, он может работать до завершения. Если хотя бы один из ресурсов занят, процесс будет ждать.

Данное решение применяется в пакетных мэйнфреймах (mainframe), которые требуют от пользователей перечислить все необходимые его программе ресурсы. Другим примером может служить механизм двухфазной локализации записей в СУБД. Однако в целом подобный подход не слишком привлекателен и приводит к неэффективному использованию компьютера. Как уже отмечалось, перечень будущих запросов к ресурсам редко удается спрогнозировать. Если такая информация есть, то можно воспользоваться алгоритмом банкира. Заметим также, что описываемый подход противоречит парадигме модульности в программировании, поскольку приложение должно знать о предполагаемых запросах к ресурсам во всех модулях.

Нарушение принципа отсутствия перераспределения

Если бы можно было отбирать ресурсы у удерживающих их процессов до завершения этих процессов, то удалось бы добиться невыполнения третьего условия возникновения тупиков. Перечислим минусы данного подхода.

Во-первых, отбирать у процессов можно только те ресурсы, состояние которых легко сохранить, а позже восстановить, например состояние процессора. Во-вторых, если процесс в течение некоторого времени использует определенные ресурсы, а затем освобождает эти ресурсы, он может потерять результаты работы, проделанной до настоящего момента. Наконец, следствием данной схемы может быть дискриминация отдельных процессов, у которых постоянно отбирают ресурсы.

Весь вопрос в цене подобного решения, которая может быть слишком высокой, если необходимость отбирать ресурсы возникает часто.

Нарушение условия кругового ожидания

Трудно предложить разумную стратегию, чтобы избежать последнего условия из раздела "Условия возникновения тупиков" – циклического ожидания.

Один из способов – упорядочить ресурсы. Например, можно присвоить всем ресурсам уникальные номера и потребовать, чтобы процессы запрашивали ресурсы в порядке их возрастания. Тогда круговое ожидание возникнуть не может. После последнего запроса и освобождения всех ресурсов можно разрешить процессу опять осуществить первый запрос. Очевидно, что практически невозможно найти порядок, который удовлетворит всех.

Один из немногих примеров упорядочивания ресурсов – создание иерархии спин-блокировок в Windows 2000. Спин-блокировка – простейший способ синхронизации (вопросы синхронизации процессов рассмотрены в соответствующей лекции). Спин-блокировка может быть захвачена и освобождена процессом. Классическая тупиковая ситуация возникает, когда процесс P1 захватывает спин-блокировку S1 и претендует на спин-блокировку S2, а процесс P2, захватывает спин-блокировку S2 и хочет дополнительно захватить спин-блокировку S1. Чтобы этого избежать, все спин-блокировки помещаются в упорядоченный список. Захват может осуществляться только в порядке, указанном в списке.

Другой способ атаки условия кругового ожидания – действовать в соответствии с правилом, согласно которому каждый процесс может иметь только один ресурс в каждый

момент времени. Если нужен второй ресурс – освободи первый. Очевидно, что для многих процессов это неприемлемо.

Таким образом, технология предотвращения циклического ожидания, как правило, неэффективна и может без необходимости закрывать доступ к ресурсам.

Обнаружение тупиков

Обнаружение взаимоблокировки сводится к фиксации тупиковой ситуации и выявлению вовлеченных в нее процессов. Для этого производится проверка наличия циклического ожидания в случаях, когда выполнены первые три условия возникновения тупика. Методы обнаружения активно используют графы распределения ресурсов.

Рассмотрим модельную ситуацию.

- Процесс P1 ожидает ресурс R1.
- Процесс P2 удерживает ресурс R2 и ожидает ресурс R1.
- Процесс P3 удерживает ресурс R1 и ожидает ресурс R3.
- Процесс P4 ожидает ресурс R2.
- Процесс P5 удерживает ресурс R3 и ожидает ресурс R2.

Вопрос состоит в том, является ли данная ситуация тупиковой, и если да, то какие процессы в ней участвуют. Для ответа на этот вопрос можно сконструировать граф ресурсов, как показано на рисунке 8.2. Из рисунка видно, что имеется цикл, моделирующий условие кругового ожидания, и что процессы P2,P3,P5, а может быть, и другие находятся в тупиковой ситуации.

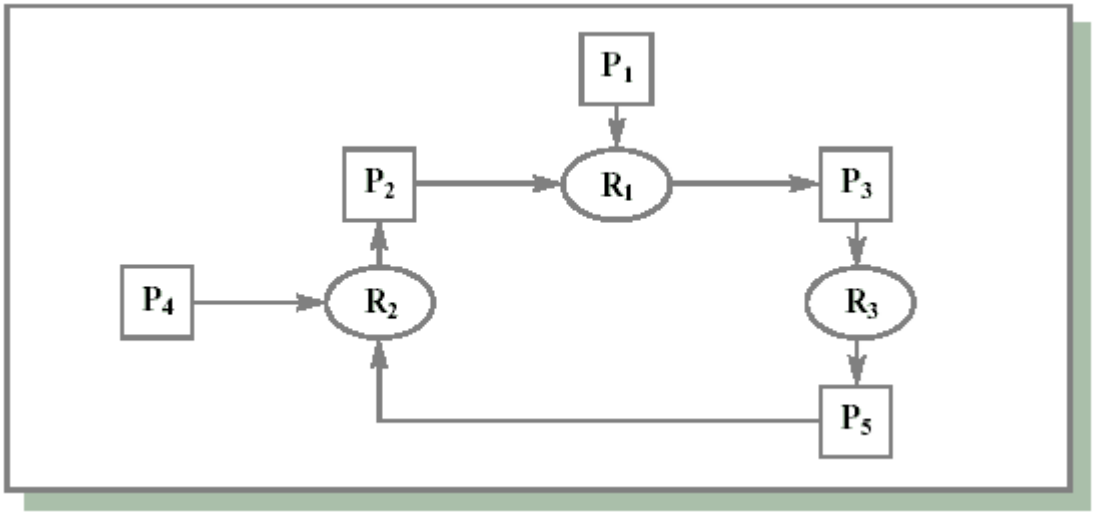


Рисунок 8.2 — Граф ресурсов

Визуально легко обнаружить наличие тупика, но нужны также формальные алгоритмы, реализуемые на компьютере.

Восстановление после тупиков

Обнаружив тупик, можно вывести из него систему, нарушив одно из условий существования тупика. При этом, возможно, несколько процессов частично или полностью потеряют результаты проделанной работы.

Сложность восстановления обусловлена рядом факторов.

- В большинстве систем нет достаточно эффективных средств, чтобы приостановить процесс, вывести его из системы и возобновить впоследствии с того места, где он был остановлен.

- Если даже такие средства есть, то их использование требует затрат и внимания оператора.
- Восстановление после тупика может потребовать значительных усилий.

Самый простой и наиболее распространенный способ устранить тупик – завершить выполнение одного или более процессов, чтобы впоследствии использовать его ресурсы. Тогда в случае удачи остальные процессы смогут выполняться. Если это не помогает, можно ликвидировать еще несколько процессов. После каждой ликвидации должен запускаться алгоритм обнаружения тупика.

По возможности лучше ликвидировать тот процесс, который может быть без ущерба возвращен к началу (такие процессы называются идемпотентными). Примером такого процесса может служить компиляция. С другой стороны, процесс, который изменяет содержимое базы данных, не всегда может быть корректно запущен повторно.

В некоторых случаях можно временно забрать ресурс у текущего владельца и передать его другому процессу. Возможность забрать ресурс у процесса, дать его другому процессу и затем без ущерба вернуть назад сильно зависит от природы ресурса. Подобное восстановление часто затруднительно, если не невозможно.

В ряде систем реализованы средства отката и перезапуска или рестарта с контрольной точки (сохранение состояния системы в какой-то момент времени). Если проектировщики системы знают, что тупик вероятен, они могут периодически организовывать для процессов контрольные точки. Иногда это приходится делать разработчикам прикладных программ.

Когда тупик обнаружен, видно, какие ресурсы вовлечены в цикл кругового ожидания. Чтобы осуществить восстановление, процесс, который владеет таким ресурсом, должен быть отброшен к моменту времени, предшествующему его запросу на этот ресурс.

Заключение

Возникновение тупиков является потенциальной проблемой любой операционной системы. Они возникают, когда имеется группа процессов, каждый из которых пытается получить исключительный доступ к некоторым ресурсам и претендует на ресурсы, принадлежащие другому процессу. В итоге все они оказываются в состоянии бесконечного ожидания.

С тупиками можно бороться, можно их обнаруживать, избегать и восстанавливать систему после тупиков. Однако цена подобных действий высока и соответствующие усилия должны предприниматься только в системах, где игнорирование тупиковых ситуаций приводит к катастрофическим последствиям.

9 УПРАВЛЕНИЕ ПАМЯТЬЮ

Главная задача компьютерной системы – выполнять программы. Программы вместе с данными, к которым они имеют доступ, в процессе выполнения должны (по крайней мере частично) находиться в оперативной памяти. Операционной системе приходится решать задачу распределения памяти между пользовательскими процессами и компонентами ОС. Эта деятельность называется управлением памятью. Таким образом, память (storage, memory) является важнейшим ресурсом, требующим тщательного управления. В недавнем прошлом память была самым дорогим ресурсом.

Часть ОС, которая отвечает за управление памятью, называется менеджером памяти.

Физическая организация памяти компьютера

Запоминающие устройства компьютера разделяют, как минимум, на два уровня: основную (главную, оперативную, физическую) и вторичную (внешнюю) память.

Основная память представляет собой упорядоченный массив однобайтовых ячеек, каждая из которых имеет свой уникальный адрес (номер). Процессор извлекает команду из основной памяти, декодирует и выполняет ее. Для выполнения команды могут потребоваться обращения еще к нескольким ячейкам основной памяти. Обычно основная память изготавливается с применением полупроводниковых технологий и теряет свое содержимое при отключении питания.

Вторичную память (это главным образом диски) также можно рассматривать как одномерное линейное адресное пространство, состоящее из последовательности байтов. В отличие от оперативной памяти, она является энергонезависимой, имеет существенно большую емкость и используется в качестве расширения основной памяти.

Эту схему можно дополнить еще несколькими промежуточными уровнями, как показано на рисунке 9.1. Разновидности памяти могут быть объединены в иерархию по убыванию времени доступа, возрастанию цены и увеличению емкости.

Рисунок 9.1 — Иерархия памяти

Многоуровневую схему используют следующим образом. Информация, которая находится в памяти верхнего уровня, обычно хранится также на уровнях с большими номерами. Если процессор не обнаруживает нужную информацию на i-м уровне, он начинает искать ее на следующих уровнях. Когда нужная информация найдена, она переносится в более быстрые уровни.

Локальность

Оказывается, при таком способе организации по мере снижения скорости доступа к уровню памяти снижается также и частота обращений к нему.

Ключевую роль здесь играет свойство реальных программ, в течение ограниченного отрезка времени способных работать с небольшим набором адресов памяти. Это эмпирически наблюдаемое свойство известно как принцип локальности или локализации обращений.

Адреса в основной памяти, характеризующие реальное расположение данных в физической памяти, называются физическими адресами. Набор физических адресов, с которым работает программа, называют физическим адресным пространством.

Логическая память

Аппаратная организация памяти в виде линейного набора ячеек не соответствует представлениям программиста о том, как организовано хранение программ и данных. Большинство программ представляет собой набор модулей, созданных независимо друг от друга. Иногда все модули, входящие в состав процесса, располагаются в памяти один за другим, образуя линейное пространство адресов. Однако чаще модули помещаются в разные области памяти и используются по-разному.

Схема управления памятью, поддерживающая этот взгляд пользователя на то, как хранятся программы и данные, называется сегментацией. Сегмент – область памяти определенного назначения, внутри которой поддерживается линейная адресация. Сегменты содержат процедуры, массивы, стек или скалярные величины, но обычно не содержат информацию смешанного типа.

По-видимому, вначале сегменты памяти появились в связи с необходимостью обобществления процессами фрагментов программного кода (текстовый редактор, тригонометрические библиотеки и т. д.), без чего каждый процесс должен был хранить в своем адресном пространстве дублирующую информацию. Эти отдельные участки памяти, хранящие информацию, которую система отображает в память нескольких процессов, получили название сегментов. Память, таким образом, перестала быть линейной и превратилась в двумерную. Адрес состоит из двух компонентов: номер сегмента, смещение внутри сегмента. Далее оказалось удобным размещать в разных сегментах различные компоненты процесса (код программы, данные, стек и т. д.). Попутно выяснилось, что можно контролировать характер работы с конкретным сегментом, приписав ему атрибуты, например права доступа или типы операций, которые разрешается производить с данными, хранящимися в сегменте.

Некоторые сегменты, описывающие адресное пространство процесса, показаны на рисунке 9.2. Большинство современных ОС поддерживают сегментную организацию памяти. В некоторых архитектурах (Intel, например) сегментация поддерживается оборудованием.

Адреса, к которым обращается процесс, таким образом, отличаются от адресов, реально существующих в оперативной памяти. В каждом конкретном случае используемые программой адреса могут быть представлены различными способами. Например, адреса в исходных текстах обычно символические. Компилятор связывает эти символические адреса с перемещаемыми адресами (такими, как n байт от начала модуля). Подобный адрес, сгенерированный программой, обычно называют логическим (в системах с виртуальной памятью он часто называется виртуальным) адресом. Совокупность всех логических адресов называется логическим (виртуальным) адресным пространством.

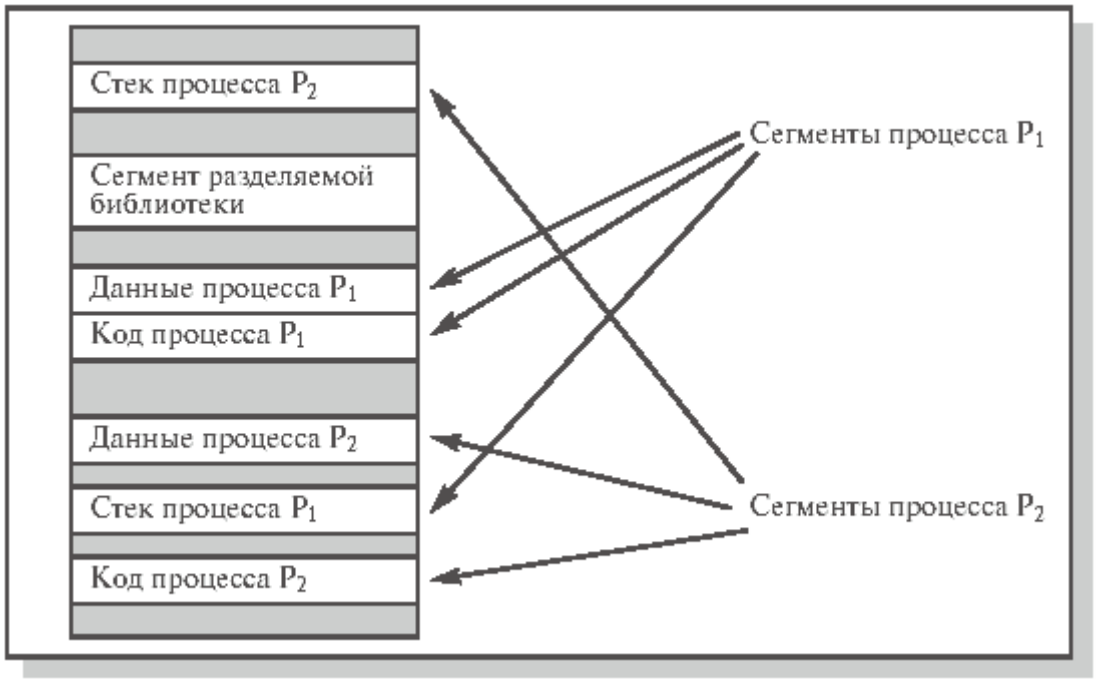


Рисунок 9.2 — Расположение сегментов процессов в памяти компьютера

Связывание адресов

Итак логические и физические адресные пространства ни по организации, ни по размеру не соответствуют друг другу. Максимальный размер логического адресного пространства обычно определяется разрядностью процессора (например, 232) и в современных системах значительно превышает размер физического адресного пространства. Следовательно, процессор и ОС должны быть способны отобразить ссылки в коде программы в реальные физические адреса, соответствующие текущему расположению программы в основной памяти. Такое отображение адресов называют трансляцией (привязкой) адреса или связыванием адресов (см. рисунок 9.3).

Связывание логического адреса, порожденного оператором программы, с физическим должно быть осуществлено до начала выполнения оператора или в момент его выполнения. Таким образом, привязка инструкций и данных к памяти в принципе может быть сделана на следующих шагах:

- Этап компиляции (Compile time). Когда на стадии компиляции известно точное место размещения процесса в памяти, тогда непосредственно генерируются физические адреса. При изменении стартового адреса программы необходимо перекомпилировать ее код. В качестве примера можно привести .com программы MS-DOS, которые связывают ее с физическими адресами на стадии компиляции.
- Этап загрузки (Load time). Если информация о размещении программы на стадии компиляции отсутствует, компилятор генерирует перемещаемый код. В этом случае окончательное связывание откладывается до момента загрузки. Если стартовый адрес меняется, нужно всего лишь перезагрузить код с учетом измененной величины.
- Этап выполнения (Execution time). Если процесс может быть перемещен во время выполнения из одной области памяти в другую, связывание откладывается до стадии выполнения. Здесь желательно наличие специализированного оборудования, например регистров перемещения. Их значение прибавляется к каждому адресу, сгенерированному

процессом. Большинство современных ОС осуществляет трансляцию адресов на этапе выполнения, используя для этого специальный аппаратный механизм.

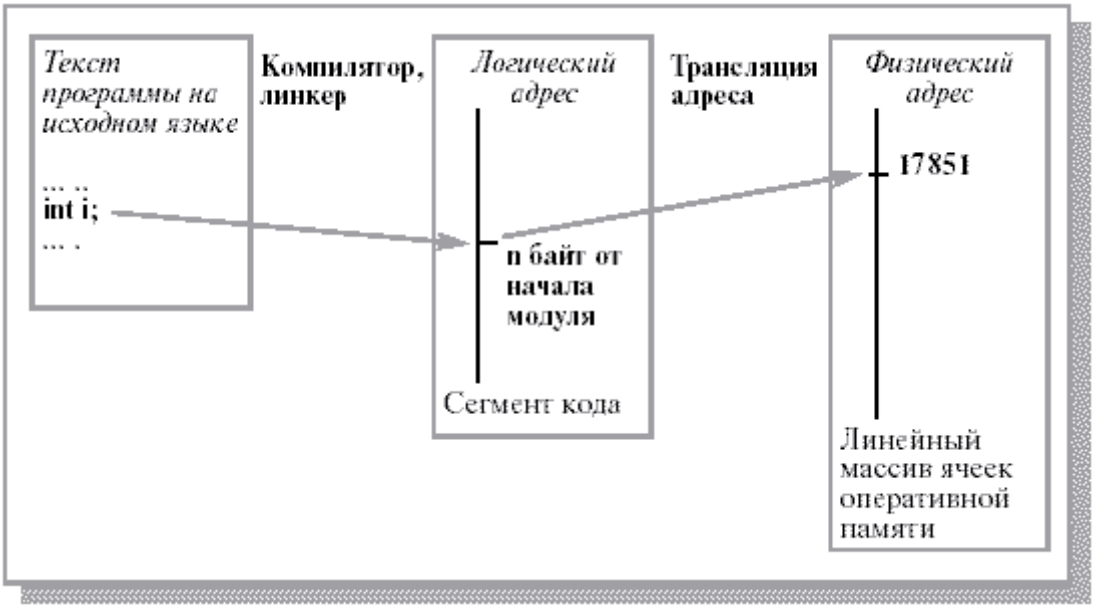


Рисунок 9.3 — Формирование логического адреса и связывание логического адреса с физическим

Функции системы управления памятью

Чтобы обеспечить эффективный контроль использования памяти, ОС должна выполнять следующие функции:

- отображение адресного пространства процесса на конкретные области физической памяти;
- распределение памяти между конкурирующими процессами;
- контроль доступа к адресным пространствам процессов;
- выгрузка процессов (целиком или частично) во внешнюю память, когда в оперативной памяти недостаточно места;
- учет свободной и занятой памяти.

В следующих разделах лекции рассматривается ряд конкретных схем управления памятью. Каждая схема включает в себя определенную идеологию управления, а также алгоритмы и структуры данных и зависит от архитектурных особенностей используемой системы. Вначале будут рассмотрены простейшие схемы. Доминирующая на сегодня схема виртуальной памяти будет описана в последующих лекциях.

Простейшие схемы управления памятью

Первые ОС применяли очень простые методы управления памятью. Вначале каждый процесс пользователя должен был полностью поместиться в основной памяти, занимать непрерывную область памяти, а система принимала к обслуживанию дополнительные пользовательские процессы до тех пор, пока все они одновременно помещались в основной памяти. Затем появился "простой свопинг" (система по-прежнему размещает каждый процесс в основной памяти целиком, но иногда на основании некоторого критерия целиком сбрасывает образ некоторого процесса из основной памяти во внешнюю и заменяет его в основной памяти образом другого процесса). Такого рода схемы имеют не только историческую ценность. В настоящее время они применяются в учебных и научно-исследовательских модельных ОС, а также в ОС для встроенных (embedded) компьютеров.

Схема с фиксированными разделами

Самым простым способом управления оперативной памятью является ее предварительное (обычно на этапе генерации или в момент загрузки системы) разбиение на несколько разделов фиксированной величины. Поступающие процессы помещаются в тот или иной раздел. При этом происходит условное разбиение физического адресного пространства. Связывание логических и физических адресов процесса происходит на этапе его загрузки в конкретный раздел, иногда – на этапе компиляции.

Каждый раздел может иметь свою очередь процессов, а может существовать и глобальная очередь для всех разделов(см. рисунок 9.4).

Эта схема была реализована в IBM OS/360 (MFT), DEC RSX-11 и ряде других систем.

Подсистема управления памятью оценивает размер поступившего процесса, выбирает подходящий для него раздел, осуществляет загрузку процесса в этот раздел и настройку адресов.

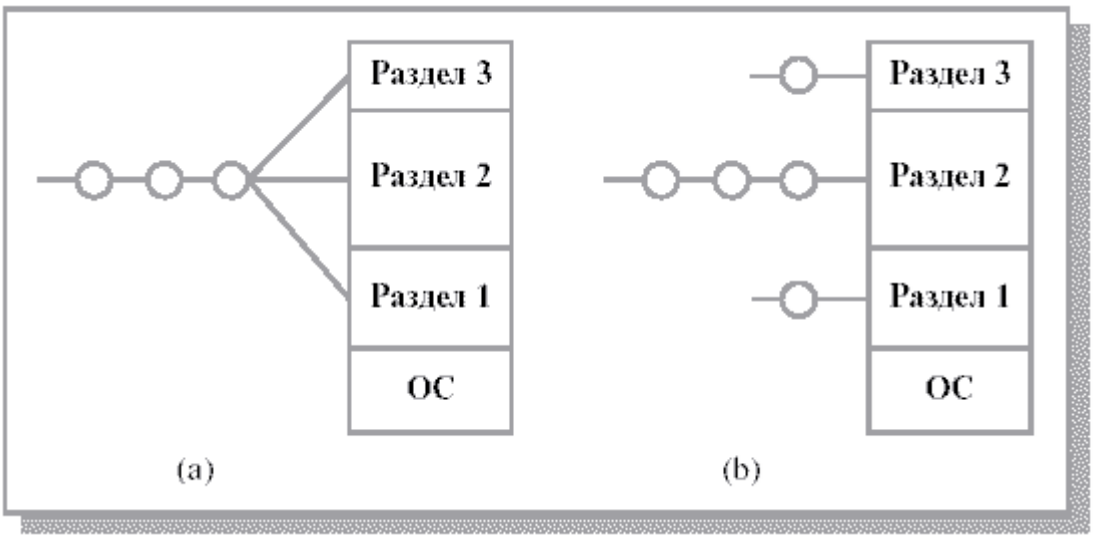


Рисунок 9.4 — Схема с фиксированными разделами: (a) – с общей очередью процессов, (b) – с отдельными очередями процессов

Очевидный недостаток этой схемы – число одновременно выполняемых процессов ограничено числом разделов.

Другим существенным недостатком является то, что предлагаемая схема сильно страдает от внутренней фрагментации – потери части памяти, выделенной процессу, но не используемой им. Фрагментация возникает потому, что процесс не полностью занимает выделенный ему раздел или потому, что некоторые разделы слишком малы для выполняемых пользовательских программ.

Один процесс в памяти

Частный случай схемы с фиксированными разделами – работа менеджера памяти однозадачной ОС. В памяти размещается один пользовательский процесс. Остается определить, где располагается пользовательская программа по отношению к ОС – в верхней части памяти, в нижней или в средней. Причем часть ОС может быть в ROM (например, BIOS, драйверы устройств). Главный фактор, влияющий на это решение, – расположение вектора прерываний, который обычно локализован в нижней части памяти, поэтому ОС также размещают в нижней. Примером такой организации может служить ОС MS-DOS.

Защита адресного пространства ОС от пользовательской программы может быть организована при помощи одного граничного регистра, содержащего адрес границы ОС.

Оверлейная структура

Так как размер логического адресного пространства процесса может быть больше, чем размер выделенного ему раздела (или больше, чем размер самого большого раздела), иногда используется техника, называемая оверлей (overlay) или организация структуры с перекрытием. Основная идея – держать в памяти только те инструкции программы, которые нужны в данный момент.

Потребность в таком способе загрузки появляется, если логическое адресное пространство системы мало, например 1 Мбайт (MS-DOS) или даже всего 64 Кбайта (PDP-11), а программа относительно велика. На современных 32-разрядных системах, где виртуальное адресное пространство измеряется гигабайтами, проблемы с нехваткой памяти решаются другими способами (см. раздел "Виртуальная память").

Коды ветвей оверлейной структуры программы находятся на диске как абсолютные образы памяти и считываются драйвером оверлеев при необходимости. Для описания оверлейной структуры обычно используется специальный несложный язык (overlay description language). Совокупность файлов исполняемой программы дополняется файлом (обычно с расширением .odl), описывающим дерево вызовов внутри программы. Для примера, приведенного на рисунке 9.5, текст этого файла может выглядеть так:

A-(B,C)

C-(D,E)

Синтаксис подобного файла может распознаваться загрузчиком. Привязка к физической памяти происходит в момент очередной загрузки одной из ветвей программы.

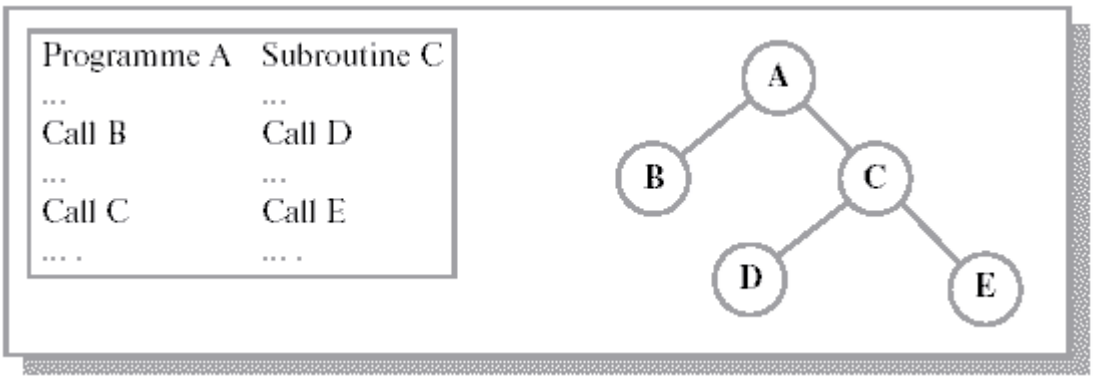


Рисунок 9.5 — Организация структуры с перекрытием. Можно поочередно загружать в память ветви A-B, A-C-D и A-C-E программы

Оверлеи могут быть полностью реализованы на пользовательском уровне в системах с простой файловой структурой. ОС при этом лишь делает несколько больше операций ввода-вывода. Типовое решение – порождение линкером специальных команд, которые включают загрузчик каждый раз, когда требуется обращение к одной из перекрывающихся ветвей программы.

Тщательное проектирование оверлейной структуры отнимает много времени и требует знания устройства программы, ее кода, данных и языка описания оверлейной структуры. По этой причине применение оверлеев ограничено компьютерами с небольшим логическим адресным пространством. Как мы увидим в дальнейшем, проблема оверлейных

сегментов, контролируемых программистом, отпадает благодаря появлению систем виртуальной памяти.

Заметим, что возможность организации структур с перекрытиями во многом обусловлена свойством локальности, которое позволяет хранить в памяти только ту информацию, которая необходима в конкретный момент вычислений.

Динамическое распределение. Свопинг

Имея дело с пакетными системами, можно обходиться фиксированными разделами и не использовать ничего более сложного. В системах с разделением времени возможна ситуация, когда память не в состоянии содержать все пользовательские процессы. Приходится прибегать к свопингу (swapping) – перемещению процессов из главной памяти на диск и обратно целиком. Частичная выгрузка процессов на диск осуществляется в системах со страничной организацией (paging) и будет рассмотрена ниже.

Выгруженный процесс может быть возвращен в то же самое адресное пространство или в другое. Это ограничение диктуется методом связывания. Для схемы связывания на этапе выполнения можно загрузить процесс в другое место памяти.

Свопинг не имеет непосредственного отношения к управлению памятью, скорее он связан с подсистемой планирования процессов. Очевидно, что свопинг увеличивает время переключения контекста. Время выгрузки может быть сокращено за счет организации специально отведенного пространства на диске (раздел для свопинга). Обмен с диском при этом осуществляется блоками большего размера, то есть быстрее, чем через стандартную файловую систему. Во многих версиях Unix свопинг начинает работать только тогда, когда возникает необходимость в снижении загрузки системы.

Схема с переменными разделами

В принципе, система свопинга может базироваться на фиксированных разделах. Более эффективной, однако, представляется схема динамического распределения или схема с переменными разделами, которая может использоваться и в тех случаях, когда все процессы целиком помещаются в памяти, то есть в отсутствие свопинга. В этом случае вначале вся память свободна и не разделена заранее на разделы. Вновь поступающей задаче выделяется строго необходимое количество памяти, не более. После выгрузки процесса память временно освобождается. По истечении некоторого времени память представляет собой переменное число разделов разного размера (см. рисунок 9.6). Смежные свободные участки могут быть объединены.

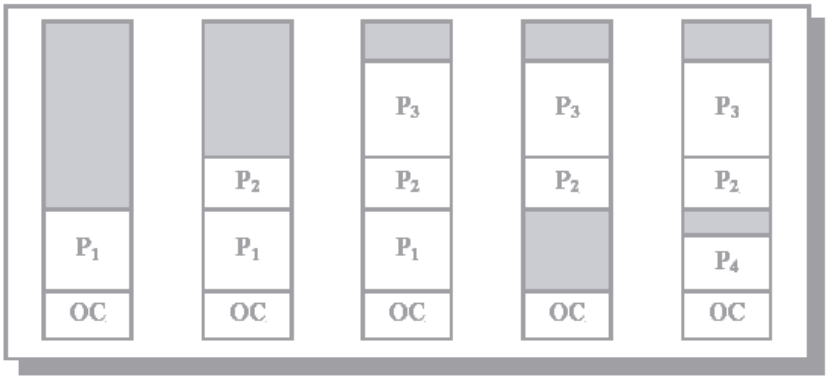


Рисунок 9.6 — Динамика распределения памяти между процессами (серым цветом показана неиспользуемая память)

В какой раздел помещать процесс? Наиболее распространены три стратегии.

- Стратегия первого подходящего (First fit). Процесс помещается в первый подходящий по размеру раздел.
- Стратегия наиболее подходящего (Best fit). Процесс помещается в тот раздел, где после его загрузки останется меньше всего свободного места.
- Стратегия наименее подходящего (Worst fit). При помещении в самый большой раздел в нем остается достаточно места для возможного размещения еще одного процесса.

Моделирование показало, что доля полезно используемой памяти в первых двух случаях больше, при этом первый способ несколько быстрее. Попутно заметим, что перечисленные стратегии широко применяются и другими компонентами ОС, например для размещения файлов на диске.

Типовой цикл работы менеджера памяти состоит в анализе запроса на выделение свободного участка (раздела), выборе его среди имеющихся в соответствии с одной из стратегий (первого подходящего, наиболее подходящего и наименее подходящего), загрузке процесса в выбранный раздел и последующих изменениях таблиц свободных и занятых областей. Аналогичная корректировка необходима и после завершения процесса. Связывание адресов может осуществляться на этапах загрузки и выполнения.

Этот метод более гибок по сравнению с методом фиксированных разделов, однако ему присуща внешняя фрагментация – наличие большого числа участков неиспользуемой памяти, не выделенной ни одному процессу. Выбор стратегии размещения процесса между первым подходящим и наиболее подходящим слабо влияет на величину фрагментации. Любопытно, что метод наиболее подходящего может оказаться наихудшим, так как он оставляет множество мелких незанятых блоков.

Статистический анализ показывает, что пропадает в среднем 1/3 памяти! Это известное правило 50% (два соседних свободных участка в отличие от двух соседних процессов могут быть объединены).

Одно из решений проблемы внешней фрагментации – организовать сжатие, то есть перемещение всех занятых (свободных) участков в сторону возрастания (убывания) адресов, так, чтобы вся свободная память образовала непрерывную область. Этот метод иногда называют схемой с перемещаемыми разделами. В идеале фрагментация после сжатия должна отсутствовать. Сжатие, однако, является дорогостоящей процедурой, алгоритм выбора оптимальной стратегии сжатия очень труден и, как правило, сжатие осуществляется в комбинации с выгрузкой и загрузкой по другим адресам.

Страничная память

Описанные выше схемы недостаточно эффективно используют память, поэтому в современных схемах управления памятью не принято размещать процесс в оперативной памяти одним непрерывным блоком.

В самом простом и наиболее распространенном случае страничной организации памяти (или paging) как логическое адресное пространство, так и физическое представляются состоящими из наборов блоков или страниц одинакового размера. При этом образуются логические страницы (page), а соответствующие единицы в физической памяти называют физическими страницами или страничными кадрами (page frames). Страницы (и страничные кадры) имеют фиксированную длину, обычно являющуюся степенью числа 2, и не могут перекрываться. Каждый кадр содержит одну страницу данных. При такой организации внешняя фрагментация отсутствует, а потери из-за внутренней фрагментации, поскольку процесс занимает целое число страниц, ограничены частью последней страницы процесса.

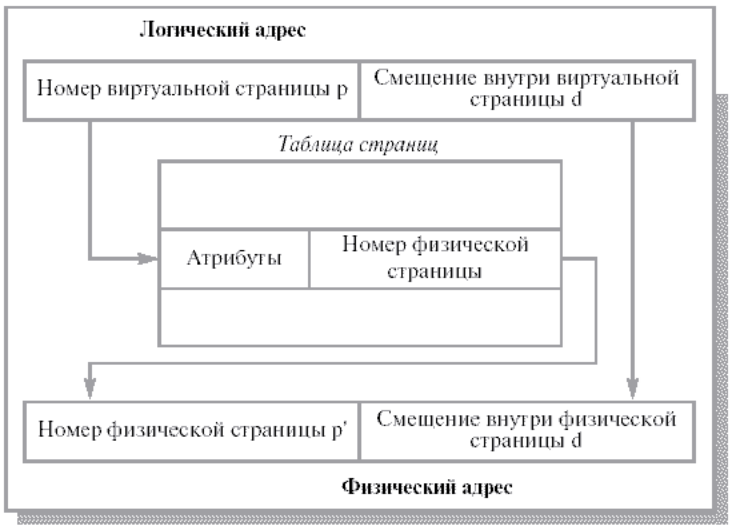
Логический адрес в страничной системе – упорядоченная пара (p,d), где p – номер страницы в виртуальной памяти, а d – смещение в рамках страницы p, на которой размещается адресуемый элемент. Заметим, что разбиение адресного пространства на страницы осуществляется вычислительной системой незаметно для программиста. Поэтому адрес является двумерным лишь с точки зрения операционной системы, а с точки зрения программиста адресное пространство процесса остается линейным.

Описываемая схема позволяет загрузить процесс, даже если нет непрерывной области кадров, достаточной для размещения процесса целиком. Но одного базового регистра для осуществления трансляции адреса в данной схеме недостаточно. Система отображения логических адресов в физические сводится к системе отображения логических страниц в физические и представляет собой таблицу страниц, которая хранится в оперативной памяти. Иногда говорят, что таблица страниц – это кусочно-линейная функция отображения, заданная в табличном виде.

Интерпретация логического адреса показана на рисунке 9.7. Если выполняемый процесс обращается к логическому адресу v = (p,d), механизм отображения ищет номер страницы p в таблице страниц и определяет, что эта страница находится в страничном кадре p', формируя реальный адрес из p' и d.

Таблица страниц (page table) адресуется при помощи специального регистра процессора и позволяет определить номер кадра по логическому адресу. Помимо этой основной задачи, при помощи атрибутов, записанных в строке таблицы страниц, можно организовать контроль доступа к конкретной странице и ее защиту.

Отметим еще раз различие точек зрения пользователя и системы на используемую память. С точки зрения пользователя, его память – единое непрерывное пространство, содержащее только одну программу. Реальное отображение скрыто от пользователя и контролируется ОС. Заметим, что процессу пользователя чужая память недоступна. Он не имеет возможности адресовать память за пределами своей таблицы страниц, которая включает только его собственные страницы.



Рисунке 9.7 — Связь логического и физического адресов при страничной организации памяти

Для управления физической памятью ОС поддерживает структуру таблицы кадров. Она имеет одну запись на каждый физический кадр, показывающий его состояние.

Отображение адресов должно быть осуществлено корректно даже в сложных случаях и обычно реализуется аппаратно. Для ссылки на таблицу процессов используется специальный регистр. При переключении процессов необходимо найти таблицу страниц нового процесса, указатель на которую входит в контекст процесса.

Сегментная и сегментно-страничная организация памяти

Существуют две другие схемы организации управления памятью: сегментная и сегментно-страничная. Сегменты, в отличие от страниц, могут иметь переменный размер. Идея сегментации изложена во введении. При сегментной организации виртуальный адрес является двумерным как для программиста, так и для операционной системы, и состоит из двух полей – номера сегмента и смещения внутри сегмента. Подчеркнем, что в отличие от страничной организации, где линейный адрес преобразован в двумерный операционной системой для удобства отображения, здесь двумерность адреса является следствием представления пользователя о процессе не в виде линейного массива байтов, а как набор сегментов переменного размера (данные, код, стек...).

Программисты, пишущие на языках низкого уровня, должны иметь представление о сегментной организации, явным образом меняя значения сегментных регистров (это хорошо видно по текстам программ, написанных на Ассемблере). Логическое адресное пространство – набор сегментов. Каждый сегмент имеет имя, размер и другие параметры (уровень привилегий, разрешенные виды обращений, флаги присутствия). В отличие от страничной схемы, где пользователь задает только один адрес, который разбивается на номер страницы и смещение прозрачным для программиста образом, в сегментной схеме пользователь специфицирует каждый адрес двумя величинами: именем сегмента и смещением.

Каждый сегмент – линейная последовательность адресов, начинающаяся с 0. Максимальный размер сегмента определяется разрядностью процессора (при 32-разрядной адресации это 232 байт или 4 Гбайт). Размер сегмента может меняться динамически (например, сегмент стека). В элементе таблицы сегментов помимо физического адреса начала сегмента обычно содержится и длина сегмента. Если размер смещения в виртуальном адресе выходит за пределы размера сегмента, возникает исключительная ситуация.

Логический адрес – упорядоченная пара v=(s,d), номер сегмента и смещение внутри сегмента.

В системах, где сегменты поддерживаются аппаратно, эти параметры обычно хранятся в таблице дескрипторов сегментов, а программа обращается к этим дескрипторам по номерам-селекторам. При этом в контекст каждого процесса входит набор сегментных регистров, содержащих селекторы текущих сегментов кода, стека, данных и т. д. и определяющих, какие сегменты будут использоваться при разных видах обращений к памяти. Это позволяет процессору уже на аппаратном уровне определять допустимость обращений к памяти, упрощая реализацию защиты информации от повреждения и несанкционированного доступа.

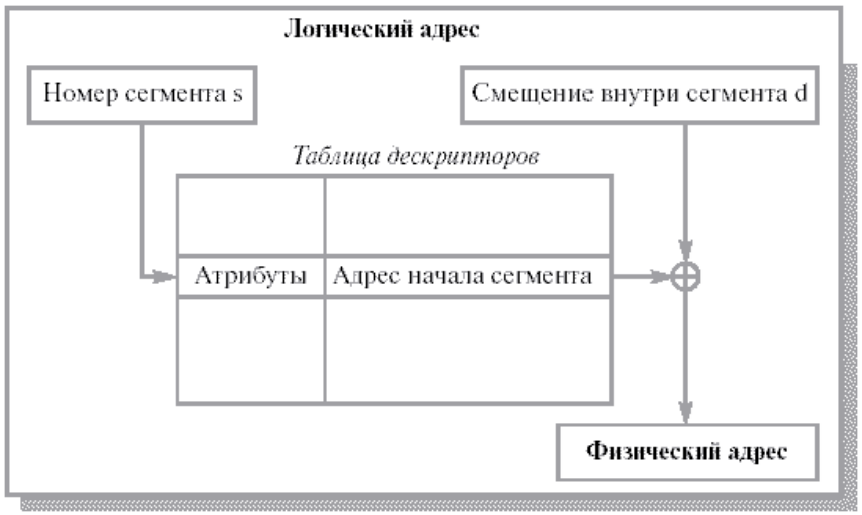


Рисунок 9.8 — Преобразование логического адреса при сегментной организации памяти

Аппаратная поддержка сегментов распространена мало (главным образом на процессорах Intel). В большинстве ОС сегментация реализуется на уровне, не зависящем от аппаратуры.

Хранить в памяти сегменты большого размера целиком так же неудобно, как и хранить процесс непрерывным блоком. Напрашивается идея разбиения сегментов на страницы. При сегментно-страничной организации памяти происходит двухуровневая трансляция виртуального адреса в физический. В этом случае логический адрес состоит из трех полей: номера сегмента логической памяти, номера страницы внутри сегмента и смещения внутри страницы. Соответственно, используются две таблицы отображения – таблица сегментов, связывающая номер сегмента с таблицей страниц, и отдельная таблица страниц для каждого сегмента.

Рисунок 9.9 — Упрощенная схема формирования физического адреса при сегментно-страничной организации памяти

Сегментно-страничная и страничная организация памяти позволяет легко организовать совместное использование одних и тех же данных и программного кода разными задачами. Для этого различные логические блоки памяти разных процессов отображают в один и тот же блок физической памяти, где размещается разделяемый фрагмент кода или данных.

Заключение

В настоящей лекции описаны простейшие способы управления памятью в ОС. Физическая память компьютера имеет иерархическую структуру. Программа представляет собой набор сегментов в логическом адресном пространстве. ОС осуществляет связывание логических и физических адресных пространств. В последующих лекциях будут рассматриваться современные решения, связанные с поддержкой виртуальной памяти.

10 СТРАНИЧНЫЙ МЕХАНИЗМ ТРАНСЛЯЦИИ

Понятие виртуальной памяти

Впервые она была реализована в 1959 г. на компьютере "Атлас", разработанном в Манчестерском университете.

Суть концепции виртуальной памяти заключается в следующем. Информация, с которой работает активный процесс, должна располагаться в оперативной памяти. В схемах виртуальной памяти у процесса создается иллюзия того, что вся необходимая ему информация имеется в основной памяти. Для этого, во-первых, занимаемая процессом память разбивается на несколько частей, например страниц. Во-вторых, логический адрес (логическая страница), к которому обращается процесс, динамически транслируется в физический адрес (физическую страницу). И, наконец, в тех случаях, когда страница, к которой обращается процесс, не находится в физической памяти, нужно организовать ее подкачку с диска. Для контроля наличия страницы в памяти вводится специальный бит присутствия, входящий в состав атрибутов страницы в таблице страниц.

Таким образом, в наличии всех компонентов процесса в основной памяти необходимости нет. Важным следствием такой организации является то, что размер памяти, занимаемой процессом, может быть больше, чем размер оперативной памяти. Принцип локальности обеспечивает этой схеме нужную эффективность.

Возможность выполнения программы, находящейся в памяти лишь частично, имеет ряд вполне очевидных преимуществ.

- Программа не ограничена объемом физической памяти. Упрощается разработка программ, поскольку можно задействовать большие виртуальные пространства, не заботясь о размере используемой памяти.
- Поскольку появляется возможность частичного помещения программы (процесса) в память и гибкого перераспределения памяти между программами, можно разместить в памяти больше программ, что увеличивает загрузку процессора и пропускную способность системы.
- Объем ввода-вывода для выгрузки части программы на диск может быть меньше, чем в варианте классического свопинга, в итоге каждая программа будет работать быстрее.

Таким образом, возможность обеспечения (при поддержке операционной системы) для программы "видимости" практически неограниченной (характерный размер для 32-разрядных архитектур 232 = 4 Гбайт) адресуемой пользовательской памяти (логическое адресное пространство) при наличии основной памяти существенно меньших размеров (физическое адресное пространство) – очень важный аспект.

Но введение виртуальной памяти позволяет решать другую, не менее важную задачу – обеспечение контроля доступа к отдельным сегментам памяти и, в частности, защиту пользовательских программ друг от друга и защиту ОС от пользовательских программ. Каждый процесс работает со своими виртуальными адресами, трансляцию которых в физические выполняет аппаратура компьютера. Таким образом, пользовательский процесс лишен возможности напрямую обратиться к страницам основной памяти, занятым информацией, относящейся к другим процессам.

Следует отметить, что оборудование компьютера принимает участие в трансляции адреса практически во всех схемах управления памятью. Но в случае виртуальной памяти это становится более сложным вследствие разрывности отображения и многомерности логического адресного пространства. Может быть, наиболее существенным вкладом

аппаратуры в реализацию описываемой схемы является автоматическая генерация исключительных ситуаций при отсутствии в памяти нужных страниц (page fault).

Любая из трех ранее рассмотренных схем управления памятью – страничной, сегментной и сегментно-страничной – пригодна для организации виртуальной памяти. Чаще всего используется сегментно-страничная модель, которая является синтезом страничной модели и идеи сегментации. Причем для тех архитектур, в которых сегменты не поддерживаются аппаратно, их реализация – задача архитектурно-независимого компонента менеджера памяти.

Сегментная организация в чистом виде встречается редко.

Архитектурные средства поддержки виртуальной памяти

В самом распространенном случае необходимо отобразить большое виртуальное адресное пространство в физическое адресное пространство существенно меньшего размера. Пользовательский процесс или ОС должны иметь возможность осуществить запись по виртуальному адресу, а задача ОС – сделать так, чтобы записанная информация оказалась в физической памяти (впоследствии при нехватке оперативной памяти она может быть вытеснена во внешнюю память). В случае виртуальной памяти система отображения адресных пространств помимо трансляции адресов должна предусматривать ведение таблиц, показывающих, какие области виртуальной памяти в данный момент находятся в физической памяти и где именно размещаются.

Страничная виртуальная память

Как и в случае простой страничной организации, страничная виртуальная память и физическая память представляются состоящими из наборов блоков или страниц одинакового размера. Виртуальные адреса делятся на страницы (page), соответствующие единицы в физической памяти образуют страничные кадры (page frames), а в целом система поддержки страничной виртуальной памяти называется пейджингом (paging). Передача информации между памятью и диском всегда осуществляется целыми страницами.

После разбиения менеджером памяти виртуального адресного пространства на страницы виртуальный адрес преобразуется в упорядоченную пару (p,d), где p – номер страницы в виртуальной памяти, а d – смещение в рамках страницы p, внутри которой размещается адресуемый элемент. Процесс может выполняться, если его текущая страница находится в оперативной памяти. Если текущей страницы в главной памяти нет, она должна быть переписана (подкачана) из внешней памяти. Поступившую страницу можно поместить в любой свободный страничный кадр.

Поскольку число виртуальных страниц велико, таблица страниц принимает специфический вид (см. раздел "Структура таблицы страниц"), структура записей становится более сложной, среди атрибутов страницы появляются биты присутствия, модификации и другие управляющие биты.

При отсутствии страницы в памяти в процессе выполнения команды возникает исключительная ситуация, называемая страничное нарушение (page fault) или страничный отказ. Обработка страничного нарушения заключается в том, что выполнение команды прерывается, затребованная страница подкачивается из конкретного места вторичной памяти в свободный страничный кадр физической памяти и попытка выполнения команды повторяется. При отсутствии свободных страничных кадров на диск выгружается редко используемая страница. Проблемы замещения страниц и обработки страничных нарушений рассматриваются в следующей лекции.

Для управления физической памятью ОС поддерживает структуру таблицы кадров. Она имеет одну запись на каждый физический кадр, показывающую его состояние.

В большинстве современных компьютеров со страничной организацией в основной памяти хранится лишь часть таблицы страниц, а быстрота доступа к элементам таблицы текущей виртуальной памяти достигается, как будет показано ниже, за счет использования сверхбыстродействующей памяти, размещенной в кэше процессора.

Сегментно-страничная организации виртуальной памяти

Как и в случае простой сегментации, в схемах виртуальной памяти сегмент – это линейная последовательность адресов, начинающаяся с 0. При организации виртуальной памяти размер сегмента может быть велик, например, может превышать размер оперативной памяти. Повторяя все ранее приведенные рассуждения о размещении в памяти больших программ, приходим к разбиению сегментов на страницы и необходимости поддержки своей таблицы страниц для каждого сегмента.

На практике, однако, появления в системе большого количества таблиц страниц стараются избежать, организуя неперекрывающиеся сегменты в одном виртуальном пространстве, для описания которого хватает одной таблицы страниц. Таким образом, одна таблица страниц отводится для всего процесса. Например, в популярных ОС Linux и Windows 2000 все сегменты процесса, а также область памяти ядра ограничены виртуальным адресным пространством объемом 4 Гбайт. При этом ядро ОС располагается по фиксированным виртуальным адресам вне зависимости от выполняемого процесса.

Структура таблицы страниц

Организация таблицы страниц – один из ключевых элементов отображения адресов в страничной и сегментно-страничной схемах. Рассмотрим структуру таблицы страниц для случая страничной организации более подробно.

Итак, виртуальный адрес состоит из виртуального номера страницы и смещения. Номер записи в таблице страниц соответствует номеру виртуальной страницы. Размер записи колеблется от системы к системе, но чаще всего он составляет 32 бита. Из этой записи в таблице страниц находится номер кадра для данной виртуальной страницы, затем прибавляется смещение и формируется физический адрес. Помимо этого запись в таблице страниц содержит информацию об атрибутах страницы. Это биты присутствия и защиты (например, 0 – read/write, 1 – read only...). Также могут быть указаны: бит модификации, который устанавливается, если содержимое страницы модифицировано, и позволяет контролировать необходимость перезаписи страницы на диск; бит ссылки, который помогает выделить малоиспользуемые страницы; бит, разрешающий кэширование, и другие управляющие биты. Заметим, что адреса страниц на диске не являются частью таблицы страниц.

Основную проблему для эффективной реализации таблицы страниц создают большие размеры виртуальных адресных пространств современных компьютеров, которые обычно определяются разрядностью архитектуры процессора. Самыми распространенными на сегодня являются 32-разрядные процессоры, позволяющие создавать виртуальные адресные пространства размером 4 Гбайт (для 64-разрядных компьютеров эта величина равна 264 байт). Кроме того, существует проблема скорости отображения, которая решается за счет использования так называемой ассоциативной памяти (см. следующий раздел).

Подсчитаем примерный размер таблицы страниц. В 32-битном адресном пространстве при размере страницы 4 Кбайт (Intel) получаем 232/212=220, то есть приблизительно миллион страниц, а в 64-битном и того более. Таким образом, таблица должна иметь примерно миллион

строк (entry), причем запись в строке состоит из нескольких байтов. Заметим, что каждый процесс нуждается в своей таблице страниц (а в случае сегментно-страничной схемы желательно иметь по одной таблице страниц на каждый сегмент).

Понятно, что количество памяти, отводимое таблицам страниц, не может быть так велико. Для того чтобы избежать размещения в памяти огромной таблицы, ее разбивают на ряд фрагментов. В оперативной памяти хранят лишь некоторые, необходимые для конкретного момента исполнения фрагменты таблицы страниц. В силу свойства локальности число таких фрагментов относительно невелико. Выполнить разбиение таблицы страниц на части можно по-разному. Наиболее распространенный способ разбиения – организация так называемой многоуровневой таблицы страниц. Для примера рассмотрим двухуровневую таблицу с размером страниц 4 Кбайт, реализованную в 32-разрядной архитектуре Intel.

Таблица, состоящая из 220 строк, разбивается на 210 таблиц второго уровня по 210 строк. Эти таблицы второго уровня объединены в общую структуру при помощи одной таблицы первого уровня, состоящей из 210 строк. 32-разрядный адрес делится на 10-разрядное поле p1, 10-разрядное поле p2 и 12-разрядное смещение d. Поле p1 указывает на нужную строку в таблице первого уровня, поле p2 – второго, а поле d локализует нужный байт внутри указанного страничного кадра (см. рисунок 10.1).

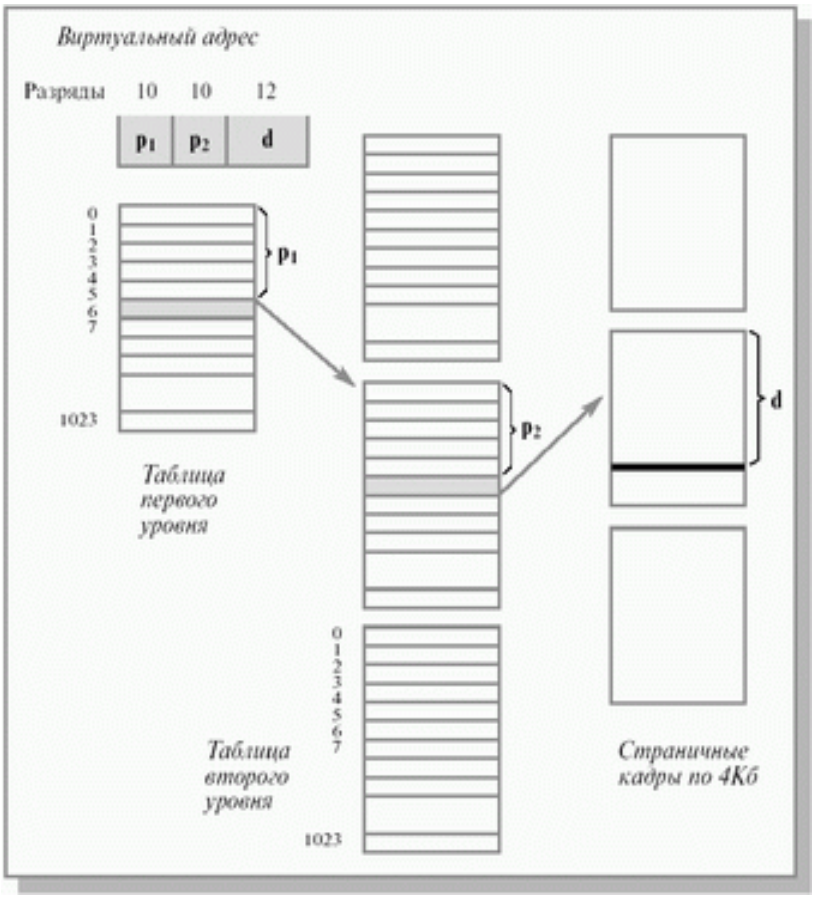


Рисунок 10.1 — Пример двухуровневой таблицы страниц

При помощи всего лишь одной таблицы второго уровня можно охватить 4 Мбайт (4 Кбайт x 1024) оперативной памяти. Таким образом, для размещения процесса с большим объемом занимаемой памяти достаточно иметь в оперативной памяти одну таблицу первого уровня и несколько таблиц второго уровня. Очевидно, что суммарное количество строк в этих

таблицах много меньше 220. Такой подход естественным образом обобщается на три и более уровней таблицы.

Наличие нескольких уровней, естественно, снижает производительность менеджера памяти. Несмотря на то что размеры таблиц на каждом уровне подобраны так, чтобы таблица помещалась целиком внутри одной страницы, обращение к каждому уровню – это отдельное обращение к памяти. Таким образом, трансляция адреса может потребовать нескольких обращений к памяти.

Количество уровней в таблице страниц зависит от конкретных особенностей архитектуры. Можно привести примеры реализации одноуровневого (DEC PDP-11), двухуровневого (Intel, DEC VAX), трехуровневого (Sun SPARC, DEC Alpha) пейджинга, а также пейджинга с заданным количеством уровней (Motorola). Функционирование RISC-процессора MIPS R2000 осуществляется вообще без таблицы страниц. Здесь поиск нужной страницы, если эта страница отсутствует в ассоциативной памяти, должна взять на себя ОС (так называемый zero level paging).

Ассоциативная память

Поиск номера кадра, соответствующего нужной странице, в многоуровневой таблице страниц требует нескольких обращений к основной памяти, поэтому занимает много времени. В некоторых случаях такая задержка недопустима. Проблема ускорения поиска решается на уровне архитектуры компьютера.

В соответствии со свойством локальности большинство программ в течение некоторого промежутка времени обращаются к небольшому количеству страниц, поэтому активно используется только небольшая часть таблицы страниц.

Естественное решение проблемы ускорения – снабдить компьютер аппаратным устройством для отображения виртуальных страниц в физические без обращения к таблице страниц, то есть иметь небольшую, быструю кэш-память, хранящую необходимую на данный момент часть таблицы страниц. Это устройство называется ассоциативной памятью, иногда также употребляют термин буфер поиска трансляции (translation lookaside buffer – TLB).

Одна запись таблицы в ассоциативной памяти (один вход) содержит информацию об одной виртуальной странице: ее атрибуты и кадр, в котором она находится. Эти поля в точности соответствуют полям в таблице страниц.

Так как ассоциативная память содержит только некоторые из записей таблицы страниц, каждая запись в TLB должна включать поле с номером виртуальной страницы. Память называется ассоциативной, потому что в ней происходит одновременное сравнение номера отображаемой виртуальной страницы с соответствующим полем во всех строках этой небольшой таблицы. Поэтому данный вид памяти достаточно дорого стоит. В строке, поле виртуальной страницы которой совпало с искомым значением, находится номер страничного кадра. Обычное число записей в TLB от 8 до 4096. Рост количества записей в ассоциативной памяти должен осуществляться с учетом таких факторов, как размер кэша основной памяти и количества обращений к памяти при выполнении одной команды.

Рассмотрим функционирование менеджера памяти при наличии ассоциативной памяти.

Вначале информация об отображении виртуальной страницы в физическую отыскивается в ассоциативной памяти. Если нужная запись найдена – все нормально, за исключением случаев нарушения привилегий, когда запрос на обращение к памяти отклоняется.

Если нужная запись в ассоциативной памяти отсутствует, отображение осуществляется через таблицу страниц. Происходит замена одной из записей в ассоциативной памяти найденной записью из таблицы страниц. Здесь мы сталкиваемся с традиционной для любого кэша проблемой замещения (а именно какую из записей в кэше необходимо изменить). Конструкция ассоциативной памяти должна организовывать записи таким образом, чтобы можно было принять решение о том, какая из старых записей должна быть удалена при внесении новых.

Число удачных поисков номера страницы в ассоциативной памяти по отношению к общему числу поисков называется hit (совпадение) ratio (пропорция, отношение). Иногда также используется термин "процент попаданий в кэш". Таким образом, hit ratio – часть ссылок, которая может быть сделана с использованием ассоциативной памяти. Обращение к одним и тем же страницам повышает hit ratio. Чем больше hit ratio, тем меньше среднее время доступа к данным, находящимся в оперативной памяти.

Предположим, например, что для определения адреса в случае кэш-промаха через таблицу страниц необходимо 100 нс, а для определения адреса в случае кэш-попадания через ассоциативную память – 20 нс. С 90% hit ratio среднее время определения адреса – 0,9x20+0,1x100 = 28 нс.

Вполне приемлемая производительность современных ОС доказывает эффективность использования ассоциативной памяти. Высокое значение вероятности нахождения данных в ассоциативной памяти связано с наличием у данных объективных свойств: пространственной и временной локальности.

Необходимо обратить внимание на следующий факт. При переключении контекста процессов нужно добиться того, чтобы новый процесс "не видел" в ассоциативной памяти информацию, относящуюся к предыдущему процессу, например очищать ее. Таким образом, использование ассоциативной памяти увеличивает время переключения контекста.

Рассмотренная двухуровневая (ассоциативная память + таблица страниц) схема преобразования адреса является ярким примером иерархии памяти, основанной на использовании принципа локальности, о чем говорилось во введении к предыдущей лекции.

Инвертированная таблица страниц

Несмотря на многоуровневую организацию, хранение нескольких таблиц страниц большого размера по-прежнему представляют собой проблему. Ее значение особенно актуально для 64-разрядных архитектур, где число виртуальных страниц очень велико. Вариантом решения является применение инвертированной таблицы страниц (inverted page table). Этот подход применяется на машинах PowerPC, некоторых рабочих станциях Hewlett-Packard, IBM RT, IBM AS/400 и ряде других.

В этой таблице содержится по одной записи на каждый страничный кадр физической памяти. Существенно, что достаточно одной таблицы для всех процессов. Таким образом, для хранения функции отображения требуется фиксированная часть основной памяти, независимо от разрядности архитектуры, размера и количества процессов. Например, для компьютера Pentium с 256 Мбайт оперативной памяти нужна таблица размером 64 Кбайт строк.

Несмотря на экономию оперативной памяти, применение инвертированной таблицы имеет существенный минус – записи в ней (как и в ассоциативной памяти) не отсортированы по возрастанию номеров виртуальных страниц, что усложняет трансляцию адреса. Один из способов решения данной проблемы – использование хеш-таблицы виртуальных адресов. При этом часть виртуального адреса, представляющая собой номер страницы, отображается в хеш-таблицу с использованием функции хеширования. Каждой странице физической памяти здесь

соответствует одна запись в хеш-таблице и инвертированной таблице страниц. Виртуальные адреса, имеющие одно значение хеш-функции, сцепляются друг с другом. Обычно длина цепочки не превышает двух записей.

Размер страницы

Разработчики ОС для существующих машин редко имеют возможность влиять на размер страницы. Однако для вновь создаваемых компьютеров решение относительно оптимального размера страницы является актуальным. Как и следовало ожидать, нет одного наилучшего размера. Скорее есть набор факторов, влияющих на размер. Обычно размер страницы – это степень двойки от 29 до 214 байт.

Чем больше размер страницы, тем меньше будет размер структур данных, обслуживающих преобразование адресов, но тем больше будут потери, связанные с тем, что память можно выделять только постранично.

Как следует выбирать размер страницы? Во-первых, нужно учитывать размер таблицы страниц, здесь желателен большой размер страницы (страниц меньше, соответственно и таблица страниц меньше). С другой стороны, память лучше утилизируется с маленьким размером страницы. В среднем половина последней страницы процесса пропадает. Необходимо также учитывать объем ввода-вывода для взаимодействия с внешней памятью и другие факторы. Проблема не имеет идеального решения. Историческая тенденция состоит в увеличении размера страницы.

Как правило, размер страниц задается аппаратно, например в DEC PDP-11 – 8 Кбайт, в DEC VAX – 512 байт, в других архитектурах, таких как Motorola 68030, размер страниц может быть задан программно. Учитывая все обстоятельства, в ряде архитектур возникают множественные размеры страниц, например в Pentium размер страницы колеблется от 4 Кбайт до 8 Кбайт. Тем не менее большинство коммерческих ОС ввиду сложности перехода на множественный размер страниц поддерживают только один размер страниц.

Заключение

В настоящей лекции рассмотрены аппаратные особенности поддержки виртуальной памяти. Разбиение адресного пространства процесса на части и динамическая трансляция адреса позволили выполнять процесс даже в отсутствие некоторых его компонентов в оперативной памяти. Подкачка недостающих компонентов с диска осуществляется операционной системой в тот момент, когда в них возникает необходимость. Следствием такой стратегии является возможность выполнения больших программ, размер которых может превышать размер оперативной памяти. Чтобы обеспечить данной схеме нужную производительность, отображение адресов осуществляется аппаратно при помощи многоуровневой таблицы страниц и ассоциативной памяти.

Большинство ОС используют сегментно-страничную виртуальную память. Для обеспечения нужной производительности менеджер памяти ОС старается поддерживать в оперативной памяти актуальную информацию, пытаясь угадать, к каким логическим адресам последует обращение в недалеком будущем. Решающую роль здесь играет удачный выбор стратегии замещения, реализованной в алгоритме выталкивания страниц.

11 ИСКЛЮЧИТЕЛЬНЫЕ СИТУАЦИИ ПРИ РАБОТЕ С ПАМЯТЬЮ

Из материала предыдущей лекции следует, что отображение виртуального адреса в физический осуществляется при помощи таблицы страниц. Для каждой виртуальной страницы запись в таблице страниц содержит номер соответствующего страничного кадра в оперативной памяти, а также атрибуты страницы для контроля обращений к памяти.

Что же происходит, когда нужной страницы в памяти нет или операция обращения к памяти недопустима? Естественно, что операционная система должна быть как-то оповещена о происшедшем. Обычно для этого используется механизм исключительных ситуаций (exceptions). При попытке выполнить подобное обращение к виртуальной странице возникает исключительная ситуация "страничное нарушение" (page fault), приводящая к вызову специальной последовательности команд для обработки конкретного вида страничного нарушения.

Страничное нарушение может происходить в самых разных случаях: при отсутствии страницы в оперативной памяти, при попытке записи в страницу с атрибутом "только чтение" или при попытке чтения или записи страницы с атрибутом "только выполнение". В любом из этих случаев вызывается обработчик страничного нарушения, являющийся частью операционной системы. Ему обычно передается причина возникновения исключительной ситуации и виртуальный адрес, обращение к которому вызвало нарушение.

Нас будет интересовать конкретный вариант страничного нарушения - обращение к отсутствующей странице, поскольку именно его обработка во многом определяет производительность страничной системы. Когда программа обращается к виртуальной странице, отсутствующей в основной памяти, операционная система должна выделить страницу основной памяти, переместить в нее копию виртуальной страницы из внешней памяти и модифицировать соответствующий элемент таблицы страниц.

Повышение производительности вычислительной системы может быть достигнуто за счет уменьшения частоты страничных нарушений, а также за счет увеличения скорости их обработки. Время эффективного доступа к отсутствующей в оперативной памяти странице складывается из:

- обслуживания исключительной ситуации (page fault);
- чтения (подкачки) страницы из вторичной памяти (иногда, при недостатке места в основной памяти, необходимо вытолкнуть одну из страниц из основной памяти во вторичную, то есть осуществить замещение страницы);
- возобновления выполнения процесса, вызвавшего данный page fault.

Для решения первой и третьей задач ОС выполняет до нескольких сот машинных инструкций в течение нескольких десятков микросекунд. Время подкачки страницы близко к нескольким десяткам миллисекунд. Проведенные исследования показывают, что вероятности page fault 5x10-7 оказывается достаточно, чтобы снизить производительность страничной схемы управления памятью на 10%. Таким образом, уменьшение частоты page faults является одной из ключевых задач системы управления памятью. Ее решение обычно связано с правильным выбором алгоритма замещения страниц.

Стратегии управления страничной памятью

Программное обеспечение подсистемы управления памятью связано с реализацией следующих стратегий:

Стратегия выборки (fetch policy) - в какой момент следует переписать страницу из вторичной памяти в первичную. Существует два основных варианта выборки - по запросу и с

упреждением. Алгоритм выборки по запросу вступает в действие в тот момент, когда процесс обращается к отсутствующей странице, содержимое которой находится на диске. Его реализация заключается в загрузке страницы с диска в свободную физическую страницу и коррекции соответствующей записи таблицы страниц.

Алгоритм выборки с упреждением осуществляет опережающее чтение, то есть кроме страницы, вызвавшей исключительную ситуацию, в память также загружается несколько страниц, окружающих ее (обычно соседние страницы располагаются во внешней памяти последовательно и могут быть считаны за одно обращение к диску). Такой алгоритм призван уменьшить накладные расходы, связанные с большим количеством исключительных ситуаций, возникающих при работе со значительными объемами данных или кода; кроме того, оптимизируется работа с диском.

Стратегия размещения (placement policy) - в какой участок первичной памяти поместить поступающую страницу. В системах со страничной организацией все просто - в любой свободный страничный кадр. В случае систем с сегментной организацией необходима стратегия, аналогичная стратегии с динамическим распределением.

Стратегия замещения (replacement policy) - какую страницу нужно вытолкнуть во внешнюю память, чтобы освободить место в оперативной памяти. Разумная стратегия замещения, реализованная в соответствующем алгоритме замещения страниц, позволяет хранить в памяти самую необходимую информацию и тем самым снизить частоту страничных нарушений. Замещение должно происходить с учетом выделенного каждому процессу количества кадров. Кроме того, нужно решить, должна ли замещаемая страница принадлежать процессу, который инициировал замещение, или она должна быть выбрана среди всех кадров основной памяти.

Алгоритмы замещения страниц

Итак, наиболее ответственным действием менеджера памяти является выделение кадра оперативной памяти для размещения в ней виртуальной страницы, находящейся во внешней памяти. Напомним, что мы рассматриваем ситуацию, когда размер виртуальной памяти для каждого процесса может существенно превосходить размер основной памяти. Это означает, что при выделении страницы основной памяти с большой вероятностью не удастся найти свободный страничный кадр. В этом случае операционная система в соответствии с заложенными в нее критериями должна:

- найти некоторую занятую страницу основной памяти;
- переместить в случае надобности ее содержимое во внешнюю память;
- переписать в этот страничный кадр содержимое нужной виртуальной страницы из внешней памяти;
- должным образом модифицировать необходимый элемент соответствующей таблицы страниц;
- продолжить выполнение процесса, которому эта виртуальная страница понадобилась.

Заметим, что при замещении приходится дважды передавать страницу между основной и вторичной памятью. Процесс замещения может быть оптимизирован за счет использования бита модификации (один из атрибутов страницы в таблице страниц). Бит модификации устанавливается компьютером, если хотя бы один байт был записан на страницу. При выборе кандидата на замещение проверяется бит модификации. Если бит не установлен, нет необходимости переписывать данную страницу на диск, ее копия на диске уже имеется.

Подобный метод также применяется к read-only-страницам, они никогда не модифицируются. Эта схема уменьшает время обработки page fault.

Существует большое количество разнообразных алгоритмов замещения страниц. Все они делятся на локальные и глобальные. Локальные алгоритмы, в отличие от глобальных, распределяют фиксированное или динамически настраиваемое число страниц для каждого процесса. Когда процесс израсходует все предназначенные ему страницы, система будет удалять из физической памяти одну из его страниц, а не из страниц других процессов. Глобальный же алгоритм замещения в случае возникновения исключительной ситуации удовлетворится освобождением любой физической страницы, независимо от того, какому процессу она принадлежала.

Глобальные алгоритмы имеют ряд недостатков. Во-первых, они делают одни процессы чувствительными к поведению других процессов. Например, если один процесс в системе одновременно использует большое количество страниц памяти, то все остальные приложения будут в результате ощущать сильное замедление из-за недостатка кадров памяти для своей работы. Во-вторых, некорректно работающее приложение может подорвать работу всей системы (если, конечно, в системе не предусмотрено ограничение на размер памяти, выделяемой процессу), пытаясь захватить больше памяти. Поэтому в многозадачной системе иногда приходится использовать более сложные локальные алгоритмы. Применение локальных алгоритмов требует хранения в операционной системе списка физических кадров, выделенных каждому процессу. Этот список страниц иногда называют резидентным множеством процесса. В одном из следующих разделов рассмотрен вариант алгоритма подкачки, основанный на приведении резидентного множества в соответствие так называемому рабочему набору процесса.

Эффективность алгоритма обычно оценивается на конкретной последовательности ссылок к памяти, для которой подсчитывается число возникающих page faults. Эта последовательность называется строкой обращений (reference string). Мы можем генерировать строку обращений искусственным образом при помощи датчика случайных чисел или трассируя конкретную систему. Последний метод дает слишком много ссылок, для уменьшения числа которых можно сделать две вещи:

- для конкретного размера страниц можно запоминать только их номера, а не адреса, на которые идет ссылка;
- несколько подряд идущих ссылок на одну страницу можно фиксировать один раз.

Как уже говорилось, большинство процессоров имеют простейшие аппаратные средства, позволяющие собирать некоторую статистику обращений к памяти. Эти средства обычно включают два специальных флага на каждый элемент таблицы страниц. Флаг ссылки (reference бит) автоматически устанавливается, когда происходит любое обращение к этой странице, а уже рассмотренный выше флаг изменения (modify бит) устанавливается, если производится запись в эту страницу. Операционная система периодически проверяет установку таких флагов, для того чтобы выделить активно используемые страницы, после чего значения этих флагов сбрасываются.

Рассмотрим ряд алгоритмов замещения страниц.

Алгоритм FIFO. Выталкивание первой пришедшей страницы

Простейший алгоритм. Каждой странице присваивается временная метка. Реализуется это просто созданием очереди страниц, в конец которой страницы попадают, когда загружаются в физическую память, а из начала берутся, когда требуется освободить память.

Для замещения выбирается старейшая страница. К сожалению, эта стратегия с достаточной вероятностью будет приводить к замещению активно используемых страниц, например страниц кода текстового процессора при редактировании файла. Заметим, что при замещении активных страниц все работает корректно, но page fault происходит немедленно.

Аномалия Билэди (Belady)

На первый взгляд кажется очевидным, что чем больше в памяти страничных кадров, тем реже будут иметь место page faults. Удивительно, но это не всегда так. Как установил Билэди с коллегами, определенные последовательности обращений к страницам в действительности приводят к увеличению числа страничных нарушений при увеличении кадров, выделенных процессу. Это явление носит название "аномалии Билэди" или "аномалии FIFO".

Система с тремя кадрами (9 faults) оказывается более производительной, чем с четырьмя кадрами (10 faults), для строки обращений к памяти 012301401234 при выборе стратегии FIFO.

Аномалию Билэди следует считать скорее курьезом, чем фактором, требующим серьезного отношения, который иллюстрирует сложность ОС, где интуитивный подход не всегда приемлем.

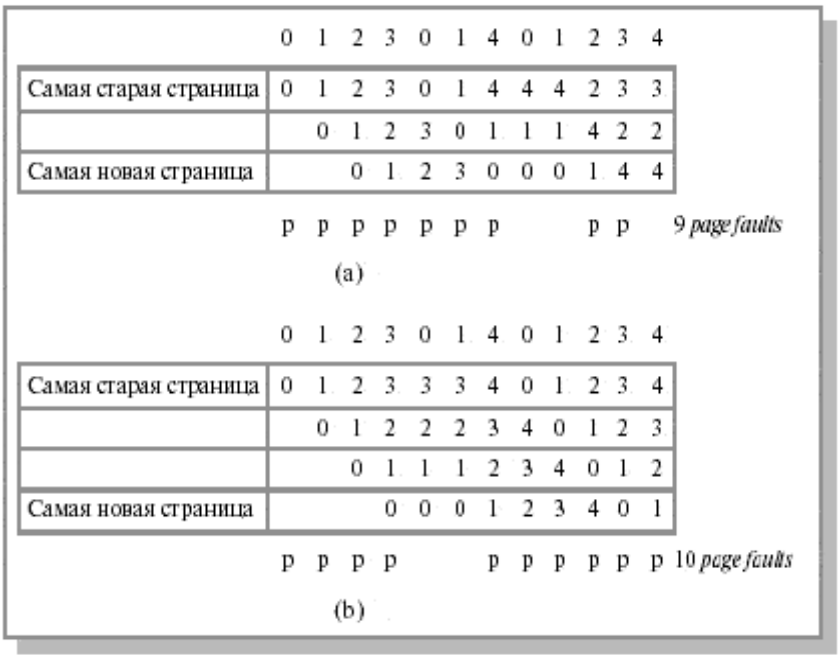


Рисунок 11.1 — Аномалия Билэди: (a) - FIFO с тремя страничными кадрами; (b) - FIFO с четырьмя страничными кадрами

Оптимальный алгоритм (OPT)

Одним из последствий открытия аномалии Билэди стал поиск оптимального алгоритма, который при заданной строке обращений имел бы минимальную частоту page faults среди всех других алгоритмов. Такой алгоритм был найден. Он прост: замещай страницу, которая не будет использоваться в течение самого длительного периода времени.

Каждая страница должна быть помечена числом инструкций, которые будут выполнены, прежде чем на эту страницу будет сделана первая ссылка. Выталкиваться должна страница, для которой это число наибольшее.

Этот алгоритм легко описать, но реализовать невозможно. ОС не знает, к какой странице будет следующее обращение. (Ранее такие проблемы возникали при планировании процессов - алгоритм SJF).

Зато мы можем сделать вывод, что для того, чтобы алгоритм замещения был максимально близок к идеальному алгоритму, система должна как можно точнее предсказывать обращения процессов к памяти. Данный алгоритм применяется для оценки качества реализуемых алгоритмов.

Выталкивание дольше всего не использовавшейся страницы. Алгоритм LRU

Одним из приближений к алгоритму OPT является алгоритм, исходящий из эвристического правила, что недавнее прошлое - хороший ориентир для прогнозирования ближайшего будущего.

Ключевое отличие между FIFO и оптимальным алгоритмом заключается в том, что один смотрит назад, а другой вперед. Если использовать прошлое для аппроксимации будущего, имеет смысл замещать страницу, которая не использовалась в течение самого долгого времени. Такой подход называется least recently used алгоритм (LRU). Работа алгоритма проиллюстрирована на рис. 11.2. Сравнивая рис. 10.1 b и 10.2, можно увидеть, что использование LRU алгоритма позволяет сократить количество страничных нарушений.

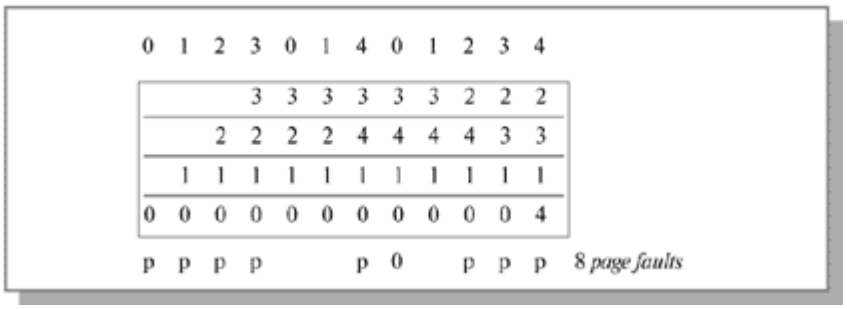


Рисунок 11.2 — Пример работы алгоритма LRU

LRU - хороший, но труднореализуемый алгоритм. Необходимо иметь связанный список всех страниц в памяти, в начале которого будут хранится недавно использованные страницы. Причем этот список должен обновляться при каждом обращении к памяти. Много времени нужно и на поиск страниц в таком списке.

Существует вариант реализации алгоритма LRU со специальным 64-битным указателем, который автоматически увеличивается на единицу после выполнения каждой инструкции, а в таблице страниц имеется соответствующее поле, в которое заносится значение указателя при каждой ссылке на страницу. При возникновении page fault выгружается страница с наименьшим значением этого поля.

Как оптимальный алгоритм, так и LRU не страдают от аномалии Билэди. Существует класс алгоритмов, для которых при одной и той же строке обращений множество страниц в памяти для n кадров всегда является подмножеством страниц для n+1 кадра. Эти алгоритмы не проявляют аномалии Билэди и называются стековыми (stack) алгоритмами.

Выталкивание редко используемой страницы. Алгоритм NFU

Поскольку большинство современных процессоров не предоставляют соответствующей аппаратной поддержки для реализации алгоритма LRU, хотелось бы иметь алгоритм, достаточно близкий к LRU, но не требующий специальной поддержки.

Программная реализация алгоритма, близкого к LRU, - алгоритм NFU(Not Frequently Used).

Для него требуются программные счетчики, по одному на каждую страницу, которые сначала равны нулю. При каждом прерывании по времени (а не после каждой инструкции) операционная система сканирует все страницы в памяти и у каждой страницы с установленным флагом обращения увеличивает на единицу значение счетчика, а флаг обращения сбрасывает.

Таким образом, кандидатом на освобождение оказывается страница с наименьшим значением счетчика, как страница, к которой реже всего обращались. Главный недостаток алгоритма NFU состоит в том, что он ничего не забывает. Например, страница, к которой очень часто обращались в течение некоторого времени, а потом обращаться перестали, все равно не будет удалена из памяти, потому что ее счетчик содержит большую величину. Например, в многопроходных компиляторах страницы, которые активно использовались во время первого прохода, могут надолго сохранить большие значения счетчика, мешая загрузке полезных в дальнейшем страниц.

К счастью, возможна небольшая модификация алгоритма, которая позволяет ему "забывать". Достаточно, чтобы при каждом прерывании по времени содержимое счетчика сдвигалось вправо на 1 бит, а уже затем производилось бы его увеличение для страниц с установленным флагом обращения.

Другим, уже более устойчивым недостатком алгоритма является длительность процесса сканирования таблиц страниц.

Другие алгоритмы

Для полноты картины можно упомянуть еще несколько алгоритмов.

Например, алгоритм Second-Chance - модификация алгоритма FIFO, которая позволяет избежать потери часто используемых страниц с помощью анализа флага обращений (бита ссылки) для самой старой страницы. Если флаг установлен, то страница, в отличие от алгоритма FIFO, не выталкивается, а ее флаг сбрасывается, и страница переносится в конец очереди. Если первоначально флаги обращений были установлены для всех страниц (на все страницы ссылались), алгоритм Second-Chance превращается в алгоритм FIFO. Данный алгоритм использовался в Multics и BSD Unix.

В компьютере Macintosh использован алгоритм NRU (Not Recently-Used), где страница-"жертва" выбирается на основе анализа битов модификации и ссылки. Интересные стратегии, основанные на буферизации страниц, реализованы в VAX/VMS и Mach.

Управление количеством страниц, выделенным процессу. Модель рабочего множества

В стратегиях замещения, рассмотренных в предыдущем разделе, прослеживается предположение о том, что количество кадров, принадлежащих процессу, нельзя увеличить. Это приводит к необходимости выталкивания страницы. Рассмотрим более общий подход, базирующийся на концепции рабочего множества, сформулированной Деннингом.

Итак, что делать, если в распоряжении процесса имеется недостаточное число кадров? Нужно ли его приостановить с освобождением всех кадров? Что следует понимать под достаточным количеством кадров?

Трешинг (Thrashing)

Хотя теоретически возможно уменьшить число кадров процесса до минимума, существует какое-то число активно используемых страниц, без которого процесс часто генерирует page faults. Высокая частота страничных нарушений называется трешинг

(thrashing, иногда употребляется русский термин "пробуксовка", см. рис. 11.3). Процесс находится в состоянии трешинга, если при его работе больше времени уходит на подкачку страниц, нежели на выполнение команд. Такого рода критическая ситуация возникает вне зависимости от конкретных алгоритмов замещения.

Рисунок 11.3 — Частота page faults в зависимости от количества кадров, выделенных процессу

Часто результатом трешинга является снижение производительности вычислительной системы. Один из нежелательных сценариев развития событий может выглядеть следующим образом. При глобальном алгоритме замещения процесс, которому не хватает кадров, начинает отбирать кадры у других процессов, которые в свою очередь начинают заниматься тем же. В результате все процессы попадают в очередь запросов к устройству вторичной памяти (находятся в состоянии ожидания), а очередь процессов в состоянии готовности пустеет. Загрузка процессора снижается. Операционная система реагирует на это увеличением степени мультипрограммирования, что приводит к еще большему трешингу и дальнейшему снижению загрузки процессора. Таким образом, пропускная способность системы падает из-за трешинга.

Эффект трешинга, возникающий при использовании глобальных алгоритмов, может быть ограничен за счет применения локальных алгоритмов замещения. При локальных алгоритмах замещения если даже один из процессов попал в трешинг, это не сказывается на других процессах. Однако он много времени проводит в очереди к устройству выгрузки, затрудняя подкачку страниц остальных процессов.

Критическая ситуация типа трешинга возникает вне зависимости от конкретных алгоритмов замещения. Единственным алгоритмом, теоретически гарантирующим отсутствие трешинга, является рассмотренный выше не реализуемый на практике оптимальный алгоритм.

Итак, трешинг - это высокая частота страничных нарушений. Необходимо ее контролировать. Когда она высока, процесс нуждается в кадрах. Можно, устанавливая желаемую частоту page faults, регулировать размер процесса, добавляя или отнимая у него кадры. Может оказаться целесообразным выгрузить процесс целиком. Освободившиеся кадры выделяются другим процессам с высокой частотой page faults.

Для предотвращения трешинга требуется выделять процессу столько кадров, сколько ему нужно. Но как узнать, сколько ему нужно? Необходимо попытаться выяснить, как много кадров процесс реально использует. Для решения этой задачи Деннинг использовал модель рабочего множества, которая основана на применении принципа локальности.

Модель рабочего множества

Рассмотрим поведение реальных процессов.

Процессы начинают работать, не имея в памяти необходимых страниц. В результате при выполнении первой же машинной инструкции возникает page fault, требующий подкачки порции кода. Следующий page fault происходит при локализации глобальных переменных и еще один - при выделении памяти для стека. После того как процесс собрал большую часть необходимых ему страниц, page faults возникают редко.

Таким образом, существует набор страниц (P1, P2, ... Pn), активно использующихся вместе, который позволяет процессу в момент времени t в течение некоторого периода T производительно работать, избегая большого количества page faults. Этот набор страниц называется рабочим множеством W(t,T) (working set) процесса. Число страниц в рабочем множестве определяется параметром T, является неубывающей функцией T и относительно невелико. Иногда T называют размером окна рабочего множества, через которое ведется наблюдение за процессом (см. рис. 11.4).

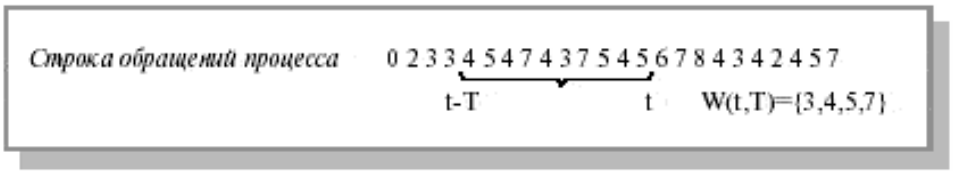


Рисунок 11.4 — Пример рабочего множества процесса

Легко написать тестовую программу, которая систематически работает с большим диапазоном адресов, но, к счастью, большинство реальных процессов не ведут себя подобным образом, а проявляют свойство локальности. В течение любой фазы вычислений процесс работает с небольшим количеством страниц.

Когда процесс выполняется, он двигается от одного рабочего множества к другому. Программа обычно состоит из нескольких рабочих множеств, которые могут перекрываться. Например, когда вызвана процедура, она определяет новое рабочее множество, состоящее из страниц, содержащих инструкции процедуры, ее локальные и глобальные переменные. После ее завершения процесс покидает это рабочее множество, но может вернуться к нему при новом вызове процедуры. Таким образом, рабочее множество определяется кодом и данными программы. Если процессу выделять меньше кадров, чем ему требуется для поддержки рабочего множества, он будет находиться в состоянии трешинга.

Принцип локальности ссылок препятствует частым изменениям рабочих наборов процессов. Формально это можно выразить следующим образом. Если в период времени (t-T, t) программа обращалась к страницам W(t,T), то при надлежащем выборе T с большой вероятностью эта программа будет обращаться к тем же страницам в период времени (t, t+T). Другими словами, принцип локальности утверждает, что если не слишком далеко заглядывать в будущее, то можно достаточно точно его прогнозировать исходя из прошлого. Понятно, что с течением времени рабочий набор процесса может изменяться (как по составу страниц, так и по их числу).

Наиболее важное свойство рабочего множества - его размер. ОС должна выделить каждому процессу достаточное число кадров, чтобы поместилось его рабочее множество. Если кадры еще остались, то может быть инициирован другой процесс. Если рабочие множества процессов не помещаются в память и начинается трешинг, то один из процессов можно выгрузить на диск.

Решение о размещении процессов в памяти должно, следовательно, базироваться на размере его рабочего множества. Для впервые инициируемых процессов это решение может быть принято эвристически. Во время работы процесса система должна уметь определять: расширяет процесс свое рабочее множество или перемещается на новое рабочее множество. Если в состав атрибутов страницы включить время последнего использования ti (для страницы с номером i), то принадлежность i-й страницы к рабочему набору, определяемому параметром T в момент времени t будет выражаться неравенством: t-T < ti < t. Алгоритм выталкивания страниц WSClock, использующий информацию о рабочем наборе процесса.

Другой способ реализации данного подхода может быть основан на отслеживании количества страничных нарушений, вызываемых процессом. Если процесс часто генерирует page faults и память не слишком заполнена, то система может увеличить число выделенных ему кадров. Если же процесс не вызывает исключительных ситуаций в течение некоторого времени и уровень генерации ниже какого-то порога, то число кадров процесса может быть урезано. Этот способ регулирует лишь размер множества страниц, принадлежащих процессу, и должен быть дополнен какой-либо стратегией замещения страниц. Несмотря на то что система при этом может пробуксовывать в моменты перехода от одного рабочего множества к другому, предлагаемое решение в состоянии обеспечить наилучшую производительность для каждого процесса, не требуя никакой дополнительной настройки системы.

Страничные демоны

Подсистема виртуальной памяти работает производительно при наличии резерва свободных страничных кадров. Алгоритмы, обеспечивающие поддержку системы в состоянии отсутствия трешинга, реализованы в составе фоновых процессов (их часто называют демонами или сервисами), которые периодически "просыпаются" и инспектируют состояние памяти. Если свободных кадров оказывается мало, они могут сменить стратегию замещения. Их задача - поддерживать систему в состоянии наилучшей производительности.

Примером такого рода процесса может быть фоновый процесс - сборщик страниц, реализующий облегченный вариант алгоритма откачки, основанный на использовании рабочего набора и применяемый во многих клонах ОС Unix. Данный демон производит откачку страниц, не входящих в рабочие наборы процессов. Он начинает активно работать, когда количество страниц в списке свободных страниц достигает установленного нижнего порога, и пытается выталкивать страницы в соответствии с собственной стратегией.

Но если возникает требование страницы в условиях, когда список свободных страниц пуст, то начинает работать механизм свопинга, поскольку простое отнятие страницы у любого процесса (включая тот, который затребовал бы страницу) потенциально вело бы к ситуации thrashing, и разрушало бы рабочий набор некоторого процесса. Любой процесс, затребовавший страницу не из своего текущего рабочего набора, становится в очередь на выгрузку в расчете на то, что после завершения выгрузки хотя бы одного из процессов свободной памяти уже может быть достаточно.

В ОС Windows 2000 аналогичную роль играет менеджер балансного набора (Working set manager), который вызывается раз в секунду или тогда, когда размер свободной памяти опускается ниже определенного предела, и отвечает за суммарную политику управления памятью и поддержку рабочих множеств.

Программная поддержка сегментной модели памяти процесса

Реализация функций операционной системы, связанных с поддержкой памяти, - ведение таблиц страниц, трансляция адреса, обработка страничных ошибок, управление ассоциативной памятью и др. - тесно связана со структурами данных, обеспечивающими

удобное представление адресного пространства процесса. Формат этих структур сильно зависит от аппаратуры и особенностей конкретной ОС.

Чаще всего виртуальная память процесса ОС разбивается на сегменты пяти типов: кода программы, данных, стека, разделяемый и сегмент файлов, отображаемых в память (см. рис. 11.5).

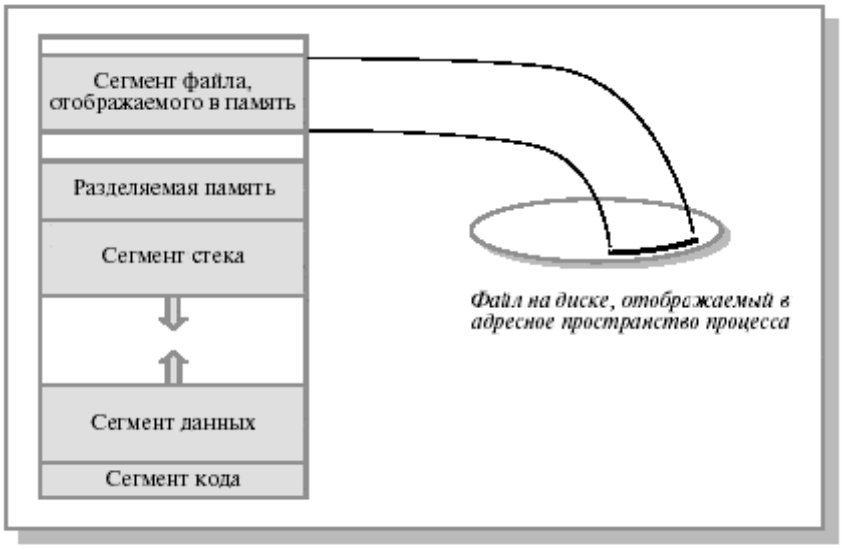


Рисунок 11.5 — Образ процесса в памяти

Сегмент программного кода содержит только команды. Сегмент программного кода не модифицируется в ходе выполнения процесса, обычно страницы данного сегмента имеют атрибут read-only. Следствием этого является возможность использования одного экземпляра кода для разных процессов.

Сегмент данных, содержащий переменные программы и сегмент стека, содержащий автоматические переменные, могут динамически менять свой размер (обычно данные в сторону увеличения адресов, а стек - в сторону уменьшения) и содержимое, должны быть доступны по чтению и записи и являются приватными сегментами процесса.

С целью обобществления памяти между несколькими процессами создаются разделяемые сегменты, допускающие доступ по чтению и записи. Вариантом разделяемого сегмента может быть сегмент файла, отображаемого в память. Специфика таких сегментов состоит в том, что из них откачка осуществляется не в системную область выгрузки, а непосредственно в отображаемый файл. Реализация разделяемых сегментов основана на том, что логические страницы различных процессов связываются с одними и теми же страничными кадрами.

Сегменты представляют собой непрерывные области (в Linux они так и называются - области) в виртуальном адресном пространстве процесса, выровненные по границам страниц. Каждая область состоит из набора страниц с одним и тем же режимом защиты. Между областями в виртуальном пространстве могут быть свободные участки. Естественно, что подобные объекты описаны соответствующими структурами (см., например, структуры mm_struct и vm_area_struct в Linux).

Часть работы по организации сегментов может происходить с участием программиста. Особенно это заметно при низкоуровневом программировании. В частности, отдельные области памяти могут быть поименованы и использоваться для обмена данными между

процессами. Два процесса могут общаться через разделяемую область памяти при условии, что им известно ее имя (пароль). Обычно это делается при помощи специальных вызовов (например, map и unmap), входящих в состав интерфейса виртуальной памяти.

Загрузка исполняемого файла (системный вызов exec) осуществляется обычно через отображение (mapping) его частей (кода, данных) в соответствующие сегменты адресного пространства процесса. Например, сегмент кода является сегментом отображаемого в память файла, содержащего исполняемую программу. При попытке выполнить первую же инструкцию система обнаруживает, что нужной части кода в памяти нет, генерирует page fault и подкачивает эту часть кода с диска. Далее процедура повторяется до тех пор, пока вся программа не окажется в оперативной памяти.

Как уже говорилось, размер сегмента данных динамически меняется. Рассмотрим, как организована поддержка сегментов данных в Unix. Пользователь, запрашивая (библиотечные вызовы malloc, new) или освобождая (free, delete) память для динамических данных, фактически изменяет границу выделенной процессу памяти через системный вызов brk (от слова break), который модифицирует значение переменной brk из структуры данных процесса. В результате происходит выделение физической памяти, граница brk смещается в сторону увеличения виртуальных адресов, а соответствующие строки таблиц страниц получают осмысленные значения. При помощи того же вызова brk пользователь может уменьшить размер сегмента данных. На практике освобожденная пользователем виртуальная память (библиотечные вызовы free, delete) системе не возвращается. На это есть две причины. Во-первых, для уменьшения размеров сегмента данных необходимо организовать его уплотнение или "сборку мусора". А во-вторых, незанятые внутри сегмента данных области естественным образом будут вытолкнуты из оперативной памяти вследствие того, что к ним не будет обращений. Ведение списков занятых и свободных областей памяти в сегменте данных пользователя осуществляется на уровне системных библиотек.

Отдельные аспекты функционирования менеджера памяти

Корректная работа менеджера памяти помимо принципиальных вопросов, связанных с выбором абстрактной модели виртуальной памяти и ее аппаратной поддержкой, обеспечивается также множеством нюансов и мелких деталей. В качестве примера такого рода компонента рассмотрим более подробно локализацию страниц в памяти, которая применяется в тех случаях, когда поддержка страничной системы приводит к необходимости разрешить определенным страницам, хранящим буферы ввода-вывода, другие важные данные и код, быть блокированными в памяти.

Рассмотрим случай, когда система виртуальной памяти может вступить в конфликт с подсистемой ввода-вывода. Например, процесс может запросить ввод в буфер и ожидать его завершения. Управление передастся другому процессу, который может вызвать page fault и, с отличной от нуля вероятностью, спровоцировать выгрузку той страницы, куда должен быть осуществлен ввод первым процессом. Подобные ситуации нуждаются в дополнительном контроле, особенно если ввод-вывод реализован с использованием механизма прямого доступа к памяти (DMA). Одно из решений данной проблемы - вводить данные в не вытесняемый буфер в пространстве ядра, а затем копировать их в пользовательское пространство.

Второе решение - локализовать страницы в памяти, используя специальный бит локализации, входящий в состав атрибутов страницы. Локализованная страница замещению не подлежит. Бит локализации сбрасывается после завершения операции ввода-вывода.

Другое использование бита локализации может иметь место и при нормальном замещении страниц. Рассмотрим следующую цепь событий. Низкоприоритетный процесс

после длительного ожидания получил в свое распоряжение процессор и подкачал с диска нужную ему страницу. Если он сразу после этого будет вытеснен высокоприоритетным процессом, последний может легко заместить вновь подкачанную страницу низкоприоритетного, так как на нее не было ссылок. Имеет смысл вновь загруженные страницы помечать битом локализации до первой ссылки, иначе низкоприоритетный процесс так и не начнет работать.

Использование бита локализации может быть опасным, если забыть его отключить. Если такая ситуация имеет место, страница становится неиспользуемой. SunOS разрешает использование данного бита в качестве подсказки, которую можно игнорировать, когда пул свободных кадров становится слишком маленьким.

Другим важным применением локализации является ее использование в системах мягкого реального времени. Рассмотрим процесс или нить реального времени. Вообще говоря, виртуальная память - антитеза вычислений реального времени, так как дает непредсказуемые задержки при подкачке страниц. Поэтому системы реального времени почти не используют виртуальную память. ОС Solaris поддерживает как реальное время, так и разделение времени. Для решения проблемы page faults, Solaris разрешает процессам сообщать системе, какие страницы важны для процесса, и локализовать их в памяти. В результате возможно выполнение процесса, реализующего задачу реального времени, содержащего локализованные страницы, где временные задержки страничной системы будут минимизированы.

Помимо системы локализации страниц, есть и другие интересные проблемы, возникающие в процессе управления памятью. Так, например, бывает непросто осуществить повторное выполнение инструкции, вызвавшей page fault. Представляют интерес и алгоритмы отложенного выделения памяти (копирование при записи и др.). Ограниченный объем данного курса не позволяет рассмотреть их более подробно.

Заключение

Описанная система управления памятью является совокупностью программно-технических средств, обеспечивающих производительное функционирование современных компьютеров. Успех реализации той части ОС, которая относится к управлению виртуальной памятью, определяется близостью архитектуры аппаратных средств, поддерживающих виртуальную память, к абстрактной модели виртуальной памяти ОС. Справедливости ради заметим, что в подавляющем большинстве современных компьютеров аппаратура выполняет функции, существенно превышающие потребности модели ОС, так что создание аппаратно-зависимой части подсистемы управления виртуальной памятью ОС в большинстве случаев не является чрезмерно сложной задачей.

12 УПРАВЛЕНИЕ ПАМЯТЬЮ ПРОЦЕССА В ОС UNIX

UNIX должен был быть мобильной операционной системой, легко переносимой на разные аппаратные платформы. Хотя на концептуальном уровне все аппаратные механизмы поддержки виртуальной памяти практически эквивалентны, реальные реализации часто весьма различаются. Невозможно создать полностью машинно-независимый компонент управления виртуальной памятью. С другой стороны, имеются существенные части программного обеспечения, связанного с управлением виртуальной памятью, для которых детали аппаратной реализации совершенно не важны. Одним из достижений ОС UNIX является грамотное и эффективное разделение средств управления виртуальной памятью на аппаратно-независимую и аппаратно-зависимую части.

Основная идея состоит в том, что ОС UNIX опирается на некоторое собственное представление организации виртуальной памяти, которое используется в аппаратно-независимой части подсистемы управления виртуальной памятью и связывается с конкретной аппаратной реализацией с помощью аппаратно-зависимой части. В чем же состоит это абстрактное представление виртуальной памяти?

Во-первых, виртуальная память каждого процесса представляется в виде набора сегментов (рисунок 12.1).

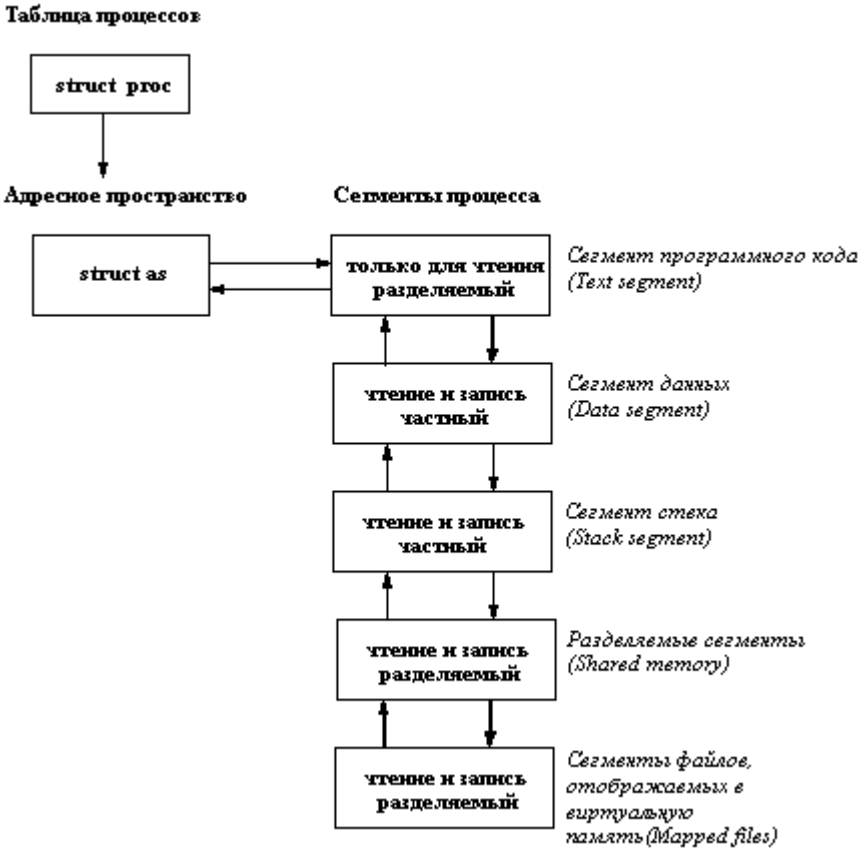


Рисунок 12.1 — Сегментная структура виртуального адресного пространства

Как видно из рисунка, виртуальная память процесса ОС UNIX разбивается на сегменты пяти разных типов. Три типа сегментов обязательны для каждой виртуальной памяти, и сегменты этих типов присутствуют в виртуальной памяти в одном экземпляре для каждого типа.

Сегмент программного кода содержит только команды. Реально в него помещается соответствующий сегмент выполняемого файла, который указывался в качестве параметра системного вызова exec для данного процесса. Сегмент программного кода не может модифицироваться в ходе выполнения процесса и потому возможно использование одного экземпляра кода для разных процессов.

Сегмент данных содержит инициализированные и неинициализированные статические переменные программы, выполняемой в данном процессе (на этом уровне изложения под статическими переменными лучше понимать области виртуальной памяти, адреса которых фиксируются в программе при ее загрузке и действуют на протяжении всего ее выполнения). Понятно, что поскольку речь идет о переменных, содержимое сегмента данных может изменяться в ходе выполнения процесса, следовательно, к сегменту должен обеспечиваться доступ и по чтению, и по записи. С другой стороны, поскольку мы говорим о собственных переменных данной программы, нельзя разрешить нескольким процессам совместно использовать один и тот же сегмент данных (по причине несогласованного изменения одних и тех же переменных разными процессами ни один из них не мог бы успешно завершиться).

Сегмент стека - это область виртуальной памяти, в которой размещаются автоматические переменные программы, явно или неявно в ней присутствующие. Этот сегмент, очевидно, должен быть динамическим (т.е. доступным и по чтению, и по записи), и он, также очевидно, должен быть частным (приватным) сегментом процесса.

Разделяемый сегмент виртуальной памяти образуется при подключении к ней сегмента разделяемой памяти. По определению, такие сегменты предназначены для координированного совместного использования несколькими процессами. Поэтому разделяемый сегмент должен допускать доступ по чтению и по записи и может разделяться несколькими процессами.

Сегменты файлов, отображаемых в виртуальную память, представляют собой разновидность разделяемых сегментов. Разница состоит в том, что если при необходимости освободить оперативную память страницы разделяемых сегментов копируются ("откачиваются") в специальную системную область подкачки (swapping space) на диске, то страницы сегментов файлов, отображаемых в виртуальную память, в случае необходимости откачиваются прямо на свое место в области внешней памяти, занимаемой файлом. Такие сегменты также допускают доступ и по чтению, и по записи и являются потенциально совместно используемыми.

На аппаратно-независимом уровне сегментная организация виртуальной памяти каждого процесса описывается структурой as, которая содержит указатель на список описателей сегментов, общий текущий размер виртуальной памяти (т.е. суммарный размер всех существующих сегментов), текущий размер физической памяти, которую процесс занимает в данный момент времени, и наконец, указатель на некоторую аппаратно-зависимую структуру, данные которой используются при отображении виртуальных адресов в физические. Описатель каждого сегмента (несколько огрубляя) содержит индивидуальные характеристики сегмента, в том числе, виртуальный адрес начала сегмента (каждый сегмент занимает некоторую непрерывную область виртуальной памяти), размер сегмента в байтах, список операций, которые можно выполнять над данным сегментом, статус сегмента (например, в каком режиме к нему возможен доступ, допускается ли совместное использование и т.д.), указатель на таблицу описателей страниц сегмента и т.д. Кроме того,

описатель каждого сегмента содержит прямые и обратные ссылки по списку описателей сегментов данной виртуальной памяти и ссылку на общий описатель виртуальной памяти as.

На уровне страниц поддерживается два вида описательных структур. Для каждой страницы физической оперативной памяти существует описатель, входящий в один из трех списков. Первый список включает описатели страниц, не допускающих модификации или отображаемых в область внешней памяти какого-либо файла (например, страницы сегментов программного кода или страницы сегмента файла, отображаемого в виртуальную память). Для таких страниц не требуется пространство в области подкачки системы; они либо вовсе не требуют откачки (перемещения копии во внешнюю память), либо откачка производится в другое место. Второй список - это список описателей свободных страниц, т.е. таких страниц, которые не подключены ни к одной виртуальной памяти. Такие страницы свободны для использования и могут быть подключены к любой виртуальной памяти. Наконец, третий список страниц включает описатели так называемых анонимных страниц, т.е. таких страниц, которые могут изменяться, но для которых нет "родного" места во внешней памяти.

В любом описателе физической страницы сохраняются копии признаков обращения и модификации страницы, вырабатываемых конкретной используемой аппаратурой.

Для каждого сегмента поддерживается таблица отображения, связывающая адреса входящих в него виртуальных страниц с описателями соответствующих им физических страниц из первого или третьего списков описателей физических страниц для виртуальных страниц, присутствующих в основной памяти, или с адресами копий страниц во внешней памяти для виртуальных страниц, отсутствующих в основной памяти. (Правильнее сказать, что поддерживается отдельная таблица отображения для каждого частного сегмента и одна общая таблица отображения для каждого разделяемого сегмента.)

Введение подобной обобщенной модели организации виртуальной памяти и тщательное продумывание связи аппаратно-независимой и аппаратно-зависимой частей подсистемы управления виртуальной памятью позволило добиться того, что обращения к памяти, не требующие вмешательства операционной системы, производятся, как и полагается, напрямую с использованием конкретных аппаратных средств. Вместе с тем, все наиболее ответственные действия операционной системы, связанные с управлением виртуальной памятью, выполняются в аппаратно-независимой части с необходимыми взаимодействиями с аппаратно-зависимой частью.

Конечно, в результате сложность переноса той части ОС UNIX, которая относится к управлению виртуальной памятью, определяется сложностью написания аппаратно-зависимой части. Чем ближе архитектура аппаратуры, поддерживающей виртуальную память, к абстрактной модели виртуальной памяти ОС UNIX, тем проще перенос. Для справедливости заметим, что в подавляющем большинстве современных компьютеров аппаратура выполняет функции, существенно превышающие потребности модели UNIX, так что создание новой аппаратно-зависимой части подсистемы управления виртуальной памятью ОС UNIX в большинстве случаев не является чрезмерно сложной задачей.

Защита адресного пространства задач в многозадачных ОС

Модель без защиты – системное и пользовательское адресные пространства не защищены друг от друга, используется два сегмента памяти: для кода и для данных; при этом от системы не требуется никакого управления памятью, не требуется MMU (memory management unit – специальное аппаратное устройство для поддержки управления виртуальной памятью).

Модель защиты система/пользователь – системное адресное пространство защищено от адресного пространства пользователя, системные и пользовательские процессы выполняются в общем виртуальном адресном пространстве, при этом требуется MMU. Защита обеспечивается страничным механизмом защиты. Различаются системные и пользовательские страницы. Пользовательские приложения никак не защищены друг от друга. Процессор находится в режиме супервизора, если текущий сегмент имеет уровень 0, 1 или 2. Если уровень сегмента – 3, то процессор находится в пользовательском режиме. В этой модели необходимы четыре сегмента – два сегмента на уровне 0 (для кода и данных) и два сегмента на уровне 3. Механизм страничной защиты не добавляет накладных расходов, т.к. защита проверяется одновременно с преобразованием адреса, которое выполняет MMU; при этом ОС не нуждается в управлении памятью.

Модель защиты пользователь/пользователь – к модели система/пользователь добавляется защита между пользовательскими процессами; требуется MMU. Как и в предыдущей модели, используется механизм страничной защиты. Все страницы помечаются как привилегированные, за исключением страниц текущего процесса, которые помечаются как пользовательские. Таким образом, выполняющийся поток не может обратиться за пределы своего адресного пространства. ОС отвечает за обновление флага привилегированности для конкретной страницы в таблице страниц при переключении процесса. Как и в предыдущей модели используются четыре сегмента.

Модель защиты виртуальной памяти – каждый процесс выполняется в своей собственной виртуальной памяти, требуется MMU. У каждого процесса имеются свои собственные сегменты и, следовательно, своя таблица описателей. ОС несет ответственность за поддержку таблиц описателей. Адресуемое пространство может превышать размеры физической памяти, если используется страничная организация памяти совместно с подкачкой. Однако в системах реального времени подкачка обычно не применяется из-за ее непредсказуемости. Для решения этой проблемы доступная память разбивается на фиксированное число логических адресных пространств равного размера. Число одновременно выполняющихся процессов в системе становится ограниченным.

1. Основные принципы организации ввода/вывода в ОС.
2. Классификация устройств в/в. Контроллеры устройств в/в.
3. Адресация устройств в/в. Режимы управления вводом/выводом.

Физические принципы организации ввода-вывода

Существует много разнообразных устройств, которые могут взаимодействовать с процессором и памятью: таймер, жесткие диски, клавиатура, дисплеи, мышь, модемы и т. д., вплоть до устройств отображения и ввода информации в авиационно-космических тренажерах. Часть этих устройств может быть встроена внутрь корпуса компьютера, часть – вынесена за его пределы и общаться с компьютером через различные линии связи: кабельные, оптоволоконные, радиорелейные, спутниковые и т. д. Конкретный набор устройств и способы их подключения определяются целями функционирования вычислительной системы, желаниями и финансовыми возможностями пользователя. Несмотря на все многообразие устройств, управление их работой и обмен информацией с ними строятся на относительно небольшом наборе принципов, которые мы постараемся разобрать в этом разделе.

Общие сведения об архитектуре компьютера

В простейшем случае процессор, память и многочисленные внешние устройства связаны большим количеством электрических соединений – линий, которые в совокупности

принято называть локальной магистралью компьютера. Внутри локальной магистрали линии, служащие для передачи сходных сигналов и предназначенные для выполнения сходных функций, принято группировать в шины. При этом понятие шины включает в себя не только набор проводников, но и набор жестко заданных протоколов, определяющий перечень сообщений, который может быть передан с помощью электрических сигналов по этим проводникам. В современных компьютерах выделяют как минимум три шины:

- шину данных, состоящую из линий данных и служащую для передачи информации между процессором и памятью, процессором и устройствами ввода-вывода, памятью и внешними устройствами;
- адресную шину, состоящую из линий адреса и служащую для задания адреса ячейки памяти или указания устройства ввода-вывода, участвующих в обмене информацией;
- шину управления, состоящую из линий управления локальной магистралью и линий ее состояния, определяющих поведение локальной магистрали. В некоторых архитектурных решениях линии состояния выносятся из этой шины в отдельную шину состояния.

Количество линий, входящих в состав шины, принято называть разрядностью (шириной) этой шины. Ширина адресной шины, например, определяет максимальный размер оперативной памяти, которая может быть установлена в вычислительной системе. Ширина шины данных определяет максимальный объем информации, которая за один раз может быть получена или передана по этой шине.

Операции обмена информацией осуществляются при одновременном участии всех шин. Рассмотрим, к примеру, действия, которые должны быть выполнены для передачи информации из процессора в память. В простейшем случае необходимо выполнить три действия.

1. На адресной шине процессор должен выставить сигналы, соответствующие адресу ячейки памяти, в которую будет осуществляться передача информации.
2. На шину данных процессор должен выставить сигналы, соответствующие информации, которая должна быть записана в память.
3. После выполнения действий 1 и 2 на шину управления выставляются сигналы, соответствующие операции записи и работе с памятью, что приведет к занесению необходимой информации по нужному адресу.

Естественно, что приведенные выше действия являются необходимыми, но недостаточными при рассмотрении работы конкретных процессоров и микросхем памяти. Конкретные архитектурные решения могут требовать дополнительных действий: например, выставления на шину управления сигналов частичного использования шины данных (для передачи меньшего количества информации, чем позволяет ширина этой шины); выставления сигнала готовности магистрали после завершения записи в память, разрешающего приступить к новой операции, и т. д. Однако общие принципы выполнения операции записи в память остаются неизменными.

В то время как память легко можно представить себе в виде последовательности пронумерованных адресами ячеек, локализованных внутри одной микросхемы или набора микросхем, к устройствам ввода-вывода подобный подход неприменим. Внешние устройства разнесены пространственно и могут подключаться к локальной магистрали в одной точке или множестве точек, получивших название портов ввода-вывода. Тем не менее, точно так же, как ячейки памяти взаимно однозначно отображались в адресное пространство памяти, порты ввода-вывода можно взаимно однозначно отобразить в другое адресное пространство –

адресное пространство ввода-вывода. При этом каждый порт ввода-вывода получает свой номер или адрес в этом пространстве. В некоторых случаях, когда адресное пространство памяти (размер которого определяется шириной адресной шины) задействовано не полностью (остались адреса, которым не соответствуют физические ячейки памяти) и протоколы работы с внешним устройством совместимы с протоколами работы с памятью, часть портов ввода - вывода может быть отображена непосредственно в адресное пространство памяти (так, например, поступают с видеопамятью дисплеев), правда, тогда эти порты уже не принято называть портами. Надо отметить, что при отображении портов в адресное пространство памяти для организации доступа к ним в полной мере могут быть задействованы существующие механизмы защиты памяти без организации специальных защитных устройств.

В ситуации прямого отображения портов ввода-вывода в адресное пространство памяти действия, необходимые для записи информации и управляющих команд в эти порты или для чтения данных из них и их состояний, ничем не отличаются от действий, производимых для передачи информации между оперативной памятью и процессором, и для их выполнения применяются те же самые команды. Если же порт отображен в адресное пространство ввода-вывода, то процесс обмена информацией инициируется специальными командами ввода-вывода и включает в себя несколько другие действия. Например, для передачи данных в порт необходимо выполнить следующее.

- На адресной шине процессор должен выставить сигналы, соответствующие адресу порта, в который будет осуществляться передача информации, в адресном пространстве ввода-вывода.
- На шину данных процессор должен выставить сигналы, соответствующие информации, которая должна быть передана в порт.
- После выполнения действий 1 и 2 на шину управления выставляются сигналы, соответствующие операции записи и работе с устройствами ввода-вывода (переключение адресных пространств!), что приведет к передаче необходимой информации в нужный порт.

Существенное отличие памяти от устройств ввода-вывода заключается в том, что занесение информации в память является окончанием операции записи, в то время как занесение информации в порт зачастую представляет собой инициализацию реального совершения операции ввода-вывода. Что именно должны делать устройства, приняв информацию через свой порт, и каким именно образом они должны поставлять информацию для чтения из порта, определяется электронными схемами устройств, получившими название контроллеров. Контроллер может непосредственно управлять отдельным устройством (например, контроллер диска), а может управлять несколькими устройствами, связываясь с их контроллерами посредством специальных шин ввода-вывода (шина IDE, шина SCSI и т. д.).

Современные вычислительные системы могут иметь разнообразную архитектуру, множество шин и магистралей, мосты для перехода информации от одной шины к другой и т. п. Для нас сейчас важными являются только следующие моменты.

- Устройства ввода-вывода подключаются к системе через порты.
- Могут существовать два адресных пространства: пространство памяти и пространство ввода-вывода.
- Порты, как правило, отображаются в адресное пространство ввода-вывода и иногда – непосредственно в адресное пространство памяти.
- Использование того или иного адресного пространства определяется типом команды, выполняемой процессором, или типом ее операндов.

- Физическим управлением устройством ввода-вывода, передачей информации через порт и выставлением некоторых сигналов на магистрали занимается контроллер устройства.

Именно единообразие подключения внешних устройств к вычислительной системе является одной из составляющих идеологии, позволяющих добавлять новые устройства без перепроектирования всей системы.

Структура контроллера устройства

Контроллеры устройств ввода-вывода весьма различны как по своему внутреннему строению, так и по исполнению (от одной микросхемы до специализированной вычислительной системы со своим процессором, памятью и т. д.), поскольку им приходится управлять совершенно разными приборами. Не вдаваясь в детали этих различий, мы выделим некоторые общие черты контроллеров, необходимые им для взаимодействия с вычислительной системой. Обычно каждый контроллер имеет по крайней мере четыре внутренних регистра, называемых регистрами состояния, управления, входных данных и выходных данных. Для доступа к содержимому этих регистров вычислительная система может использовать один или несколько портов, что для нас не существенно. Для простоты изложения будем считать, что каждому регистру соответствует свой порт.

Регистр состояния содержит биты, значение которых определяется состоянием устройства ввода-вывода и которые доступны только для чтения вычислительной системой. Эти биты индицируют завершение выполнения текущей команды на устройстве (бит занятости), наличие очередного данного в регистре выходных данных (бит готовности данных), возникновение ошибки при выполнении команды (бит ошибки) и т. д.

Регистр управления получает данные, которые записываются вычислительной системой для инициализации устройства ввода-вывода или выполнения очередной команды, а также изменения режима работы устройства. Часть битов в этом регистре может быть отведена под код выполняемой команды, часть битов будет кодировать режим работы устройства, бит готовности команды свидетельствует о том, что можно приступить к ее выполнению.

Регистр выходных данных служит для помещения в него данных для чтения вычислительной системой, а регистр входных данных предназначен для помещения в него информации, которая должна быть выведена на устройство. Обычно емкость этих регистров не превышает ширину линии данных (а чаще всего меньше ее), хотя некоторые контроллеры могут использовать в качестве регистров очередь FIFO для буферизации поступающей информации.

Разумеется, набор регистров и составляющих их битов приблизителен, он призван послужить нам моделью для описания процесса передачи информации от вычислительной системы к внешнему устройству и обратно, но в том или ином виде он обычно присутствует во всех контроллерах устройств.

Опрос устройств и прерывания. Исключительные ситуации и системные вызовы

Построив модель контроллера и представляя себе, что скрывается за словами "прочитать информацию из порта" и "записать информацию в порт", мы готовы к рассмотрению процесса взаимодействия устройства и процессора. Как и в предыдущих случаях, примером нам послужит команда записи, теперь уже записи или вывода данных на внешнее устройство. В нашей модели для вывода информации, помещающейся в регистр входных данных, без проверки успешности вывода процессор и контроллер должны связываться следующим образом.

1. Процессор в цикле читает информацию из порта регистра состояний и проверяет значение бита занятости. Если бит занятости установлен, то это означает, что устройство еще не завершило предыдущую операцию, и процессор уходит на новую итерацию цикла. Если бит занятости сброшен, то устройство готово к выполнению новой операции, и процессор переходит на следующий шаг.
2. Процессор записывает код команды вывода в порт регистра управления.
3. Процессор записывает данные в порт регистра входных данных.
4. Процессор устанавливает бит готовности команды. В следующих шагах процессор не задействован.
5. Когда контроллер замечает, что бит готовности команды установлен, он устанавливает бит занятости.
6. Контроллер анализирует код команды в регистре управления и обнаруживает, что это команда вывода. Он берет данные из регистра входных данных и инициирует выполнение команды.
7. После завершения операции контроллер обнуляет бит готовности команды.
8. При успешном завершении операции контроллер обнуляет бит ошибки в регистре состояния, при неудачном завершении команды – устанавливает его.
9. Контроллер сбрасывает бит занятости.

При необходимости вывода новой порции информации все эти шаги повторяются. Если процессор интересует, корректно или некорректно была выведена информация, то после шага 4 он должен в цикле считывать информацию из порта регистра состояний до тех пор, пока не будет сброшен бит занятости устройства, после чего проанализировать состояние бита ошибки.

Как видим, на первом шаге (и, возможно, после шага 4) процессор ожидает освобождения устройства, непрерывно опрашивая значение бита занятости. Такой способ взаимодействия процессора и контроллера получил название polling или, в русском переводе, способа опроса устройств. Если скорости работы процессора и устройства ввода-вывода примерно равны, то это не приводит к существенному уменьшению полезной работы, совершаемой процессором. Если же скорость работы устройства существенно меньше скорости процессора, то указанная техника резко снижает производительность системы и необходимо применять другой архитектурный подход.

Для того чтобы процессор не дожидался состояния готовности устройства ввода-вывода в цикле, а мог выполнять в это время другую работу, необходимо, чтобы устройство само умело сигнализировать процессору о своей готовности. Технический механизм, который позволяет внешним устройствам оповещать процессор о завершении команды вывода или команды ввода, получил название механизма прерываний.

В простейшем случае для реализации механизма прерываний необходимо к имеющимся у нас шинам локальной магистрали добавить еще одну линию, соединяющую процессор и устройства ввода-вывода – линию прерываний. По завершении выполнения операции внешнее устройство выставляет на эту линию специальный сигнал, по которому процессор после выполнения очередной команды (или после завершения очередной итерации при выполнении цепочечных команд, т. е. команд, повторяющихся циклически со сдвигом по памяти) изменяет свое поведение. Вместо выполнения очередной команды из потока команд он частично сохраняет содержимое своих регистров и переходит на выполнение программы обработки прерывания, расположенной по заранее оговоренному адресу. При наличии только одной линии прерываний процессор при выполнении этой программы должен опросить

состояние всех устройств ввода-вывода, чтобы определить, от какого именно устройства пришло прерывание (polling прерываний!), выполнить необходимые действия (например, вывести в это устройство очередную порцию информации или перевести соответствующий процесс из состояния ожидание в состояние готовность) и сообщить устройству, что прерывание обработано (снять прерывание).

В большинстве современных компьютеров процессор стараются полностью освободить от необходимости опроса внешних устройств, в том числе и от определения с помощью опроса устройства, сгенерировавшего сигнал прерывания. Устройства сообщают о своей готовности процессору не напрямую, а через специальный контроллер прерываний, при этом для общения с процессором он может использовать не одну линию, а целую шину прерываний. Каждому устройству присваивается свой номер прерывания, который при возникновении прерывания контроллер прерывания заносит в свой регистр состояния и, возможно, после распознавания процессором сигнала прерывания и получения от него специального запроса выставляет на шину прерываний или шину данных для чтения процессором. Номер прерывания обычно служит индексом в специальной таблице прерываний, хранящейся по адресу, задаваемому при инициализации вычислительной системы, и содержащей адреса программ обработки прерываний – векторы прерываний. Для распределения устройств по номерам прерываний необходимо, чтобы от каждого устройства к контроллеру прерываний шла специальная линия, соответствующая одному номеру прерывания. При наличии множества устройств такое подключение становится невозможным, и на один проводник (один номер прерывания) подключается несколько устройств. В этом случае процессор при обработке прерывания все равно вынужден заниматься опросом устройств для определения устройства, выдавшего прерывание, но в существенно меньшем объеме. Обычно при установке в систему нового устройства ввода-вывода требуется аппаратно или программно определить, каким будет номер прерывания, вырабатываемый этим устройством.

Рассматривая кооперацию процессов и взаимоисключения, мы говорили о существовании критических секций внутри ядра операционной системы, при выполнении которых необходимо исключить всякие прерывания от внешних устройств. Для запрещения прерываний, а точнее, для невосприимчивости процессора к внешним прерываниям обычно существуют специальные команды, которые могут маскировать (запрещать) все или некоторые из прерываний устройств ввода-вывода. В то же время определенные кризисные ситуации в вычислительной системе (например, неустранимый сбой в работе оперативной памяти) должны требовать ее немедленной реакции. Такие ситуации вызывают прерывания, которые невозможно замаскировать или запретить и которые поступают в процессор по специальной линии шины прерываний, называемой линией немаскируемых прерываний (NMI – Non-Maskable Interrupt).

Не все внешние устройства являются одинаково важными с точки зрения вычислительной системы ("все животные равны, но некоторые равнее других"). Соответственно, некоторые прерывания являются более существенными, чем другие. Контроллер прерываний обычно позволяет устанавливать приоритеты для прерываний от внешних устройств. При почти одновременном возникновении прерываний от нескольких устройств (во время выполнения одной и той же команды процессора) процессору сообщается номер наиболее приоритетного прерывания для его обслуживания в первую очередь. Менее приоритетное прерывание при этом не пропадает, о нем процессору будет доложено после обработки более приоритетного прерывания. Более того, при обработке возникшего

прерывания процессор может получить сообщение о возникновении прерывания с более высоким приоритетом и переключиться на его обработку.

Механизм обработки прерываний, по которому процессор прекращает выполнение команд в обычном режиме и, частично сохранив свое состояние, отвлекается на выполнение других действий, оказался настолько удобен, что зачастую разработчики процессоров используют его и для других целей. Хотя эти случаи и не относятся к операциям ввода-вывода, мы вынуждены упомянуть их здесь, для того чтобы их не путали с прерываниями. Похожим образом процессор обрабатывает исключительные ситуации и программные прерывания.

Для внешних прерываний характерны следующие особенности.

- Внешнее прерывание обнаруживается процессором между выполнением команд (или между итерациями в случае выполнения цепочечных команд).
- Процессор при переходе на обработку прерывания сохраняет часть своего состояния перед выполнением следующей команды.
- Прерывания происходят асинхронно с работой процессора и непредсказуемо, программист никоим образом не может предугадать, в каком именно месте работы программы произойдет прерывание.

Исключительные ситуации возникают во время выполнения процессором команды. К их числу относятся ситуации переполнения, деления на ноль, обращения к отсутствующей странице памяти (см. часть III) и т. д. Для исключительных ситуаций характерно следующее.

- Исключительные ситуации обнаруживаются процессором во время выполнения команд.
- Процессор при переходе на выполнение обработки исключительной ситуации сохраняет часть своего состояния перед выполнением текущей команды.
- Исключительные ситуации возникают синхронно с работой процессора, но непредсказуемо для программиста, если только тот специально не заставил процессор делить некоторое число на ноль.

Программные прерывания возникают после выполнения специальных команд, как правило, для выполнения привилегированных действий внутри системных вызовов. Программные прерывания имеют следующие свойства.

- Программное прерывание происходит в результате выполнения специальной команды.
- Процессор при выполнении программного прерывания сохраняет свое состояние перед выполнением следующей команды.
- Программные прерывания, естественно, возникают синхронно с работой процессора и абсолютно предсказуемы программистом.

Надо сказать, что реализация похожих механизмов обработки внешних прерываний, исключительных ситуаций и программных прерываний лежит целиком на совести разработчиков процессоров. Существуют вычислительные системы, где все три ситуации обрабатываются по-разному.

Прямой доступ к памяти (Direct Memory Access – DMA)

Использование механизма прерываний позволяет разумно загружать процессор в то время, когда устройство ввода-вывода занимается своей работой. Однако запись или чтение большого количества информации из адресного пространства ввода-вывода (например, с диска) приводят к большому количеству операций ввода-вывода, которые должен выполнять процессор. Для освобождения процессора от операций последовательного вывода данных из оперативной памяти или последовательного ввода в нее был предложен механизм прямого

доступа внешних устройств к памяти – ПДП или Direct Memory Access – DMA. Давайте кратко рассмотрим, как работает этот механизм.

Для того чтобы какое-либо устройство, кроме процессора, могло записать информацию в память или прочитать ее из памяти, необходимо чтобы это устройство могло забрать у процессора управление локальной магистралью для выставления соответствующих сигналов на шины адреса, данных и управления. Для централизации эти обязанности обычно возлагаются не на каждое устройство в отдельности, а на специальный контроллер – контроллер прямого доступа к памяти. Контроллер прямого доступа к памяти имеет несколько спаренных линий – каналов DMA, которые могут подключаться к различным устройствам. Перед началом использования прямого доступа к памяти этот контроллер необходимо запрограммировать, записав в его порты информацию о том, какой канал или каналы предполагается задействовать, какие операции они будут совершать, какой адрес памяти является начальным для передачи информации и какое количество информации должно быть передано. Получив по одной из линий – каналов DMA, сигнал запроса на передачу данных от внешнего устройства, контроллер по шине управления сообщает процессору о желании взять на себя управление локальной магистралью. Процессор, возможно, через некоторое время, необходимое для завершения его действий с магистралью, передает управление ею контроллеру DMA, известив его специальным сигналом. Контроллер DMA выставляет на адресную шину адрес памяти для передачи очередной порции информации и по второй линии канала прямого доступа к памяти сообщает устройству о готовности магистрали к передаче данных. После этого, используя шину данных и шину управления, контроллер DMA, устройство ввода-вывода и память осуществляют процесс обмена информацией. Затем контроллер прямого доступа к памяти извещает процессор о своем отказе от управления магистралью, и тот берет руководящие функции на себя. При передаче большого количества данных весь процесс повторяется циклически.

При прямом доступе к памяти процессор и контроллер DMA по очереди управляют локальной магистралью. Это, конечно, несколько снижает производительность процессора, так как при выполнении некоторых команд или при чтении очередной порции команд во внутренний кэш он должен поджидать освобождения магистрали, но в целом производительность вычислительной системы существенно возрастает.

При подключении к системе нового устройства, которое умеет использовать прямой доступ к памяти, обычно необходимо программно или аппаратно задать номер канала DMA, к которому будет приписано устройство. В отличие от прерываний, где один номер прерывания мог соответствовать нескольким устройствам, каналы DMA всегда находятся в монопольном владении устройств.

Логические принципы организации ввода-вывода

Рассмотренные в предыдущем разделе физические механизмы взаимодействия устройств ввода-вывода с вычислительной системой позволяют понять, почему разнообразные внешние устройства легко могут быть добавлены в существующие компьютеры. Все, что необходимо сделать пользователю при подключении нового устройства, – это отобразить порты устройства в соответствующее адресное пространство, определить, какой номер будет соответствовать прерыванию, генерируемому устройством, и, если нужно, закрепить за устройством некоторый канал DMA. Для нормального функционирования hardware этого будет достаточно. Однако мы до сих пор ничего не сказали о том, как должна быть построена подсистема управления вводом-выводом в операционной системе для легкого и

безболезненного добавления новых устройств и какие функции вообще обычно на нее возлагаются.

Структура системы ввода-вывода

Если поручить неподготовленному пользователю сконструировать систему ввода-вывода, способную работать со всем множеством внешних устройств, то, скорее всего, он окажется в ситуации, в которой находились биологи и зоологи до появления трудов Линнея [Linnaes, 1789]. Все устройства разные, отличаются по выполняемым функциям и своим характеристикам, и кажется, что принципиально невозможно создать систему, которая без больших постоянных переделок позволяла бы охватывать все многообразие видов. Вот перечень лишь нескольких направлений (далеко не полный), по которым различаются устройства.

- Скорость обмена информацией может варьироваться в диапазоне от нескольких байтов в секунду (клавиатура) до нескольких гигабайтов в секунду (сетевые карты).
- Одни устройства могут использоваться несколькими процессами параллельно (являются разделяемыми), в то время как другие требуют монопольного захвата процессом.
- Устройства могут запоминать выведенную информацию для ее последующего ввода или не обладать этой функцией. Устройства, запоминающие информацию, в свою очередь, могут дифференцироваться по формам доступа к сохраненной информации: обеспечивать к ней последовательный доступ в жестко заданном порядке или уметь находить и передавать только необходимую порцию данных.
- Часть устройств умеет передавать данные только по одному байту последовательно (символьные устройства), а часть устройств умеет передавать блок байтов как единое целое (блочные устройства).
- Существуют устройства, предназначенные только для ввода информации, устройства, предназначенные только для вывода информации, и устройства, которые могут выполнять и ввод, и вывод.

В области технического обеспечения удалось выделить несколько основных принципов взаимодействия внешних устройств с вычислительной системой, т. е. создать единый интерфейс для их подключения, возложив все специфические действия на контроллеры самих устройств. Тем самым конструкторы вычислительных систем переложили все хлопоты, связанные с подключением внешней аппаратуры, на разработчиков самой аппаратуры, заставляя их придерживаться определенного стандарта.

Похожий подход оказался продуктивным и в области программного подключения устройств ввода-вывода. Подобно тому как Линнею удалось заложить основы систематизации знаний о растительном и животном мире, разделив все живое в природе на относительно небольшое число классов и отрядов, мы можем разделить устройства на относительно небольшое число типов, отличающихся по набору операций, которые могут быть ими выполнены, считая все остальные различия несущественными. Мы можем затем специфицировать интерфейсы между ядром операционной системы, осуществляющим некоторую общую политику ввода-вывода, и программными частями, непосредственно управляющими устройствами, для каждого из таких типов. Более того, разработчики операционных систем получают возможность освободиться от написания и тестирования этих специфических программных частей, получивших название драйверов, передав эту деятельность производителям самих внешних устройств. Фактически мы приходим к использованию принципа уровневого или слоеного построения системы управления вводом-выводом для операционной системы (см. рис. 12.2).

Два нижних уровня этой слоеной системы составляет hardware: сами устройства, непосредственно выполняющие операции, и их контроллеры, служащие для организации совместной работы устройств и остальной вычислительной системы. Следующий уровень составляют драйверы устройств ввода-вывода, скрывающие от разработчиков операционных систем особенности функционирования конкретных приборов и обеспечивающие четко определенный интерфейс между hardware и вышележащим уровнем – уровнем базовой подсистемы ввода-вывода, которая, в свою очередь, предоставляет механизм взаимодействия между драйверами и программной частью вычислительной системы в целом.

В последующих разделах мы подробнее рассмотрим организацию и функции набора драйверов и базовой подсистемы ввода-вывода.

Систематизация внешних устройств и интерфейс между базовой подсистемой ввода-вывода и драйверами

Как и система видов Линнея, система типов устройств является далеко не полной и не строго выдержанной. Устройства обычно принято разделять по преобладающему типу интерфейса на следующие виды:

- символьные (клавиатура, модем, терминал и т. п.);
- блочные (магнитные и оптические диски и ленты, и т. д.);
- сетевые (сетевые карты);
- все остальные (таймеры, графические дисплеи, телевизионные устройства, видеокамеры и т. п.).

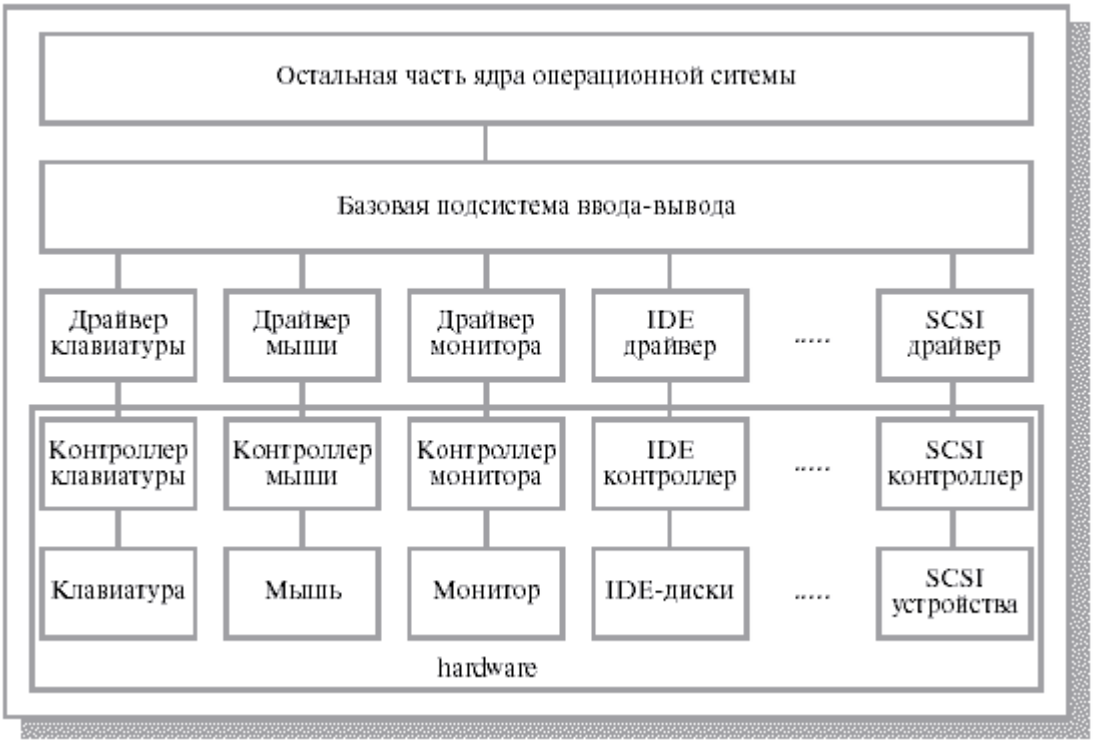


Рисунок 12.2 — Структура системы ввода-вывода

Такое деление является весьма условным. В одних операционных системах сетевые устройства могут не выделяться в отдельную группу, в некоторых других – отдельные группы составляют звуковые устройства и видеоустройства и т. д. Некоторые группы в свою очередь могут разбиваться на подгруппы: подгруппа жестких дисков, подгруппа мышек, подгруппа принтеров. Нас такие детали не интересуют. Мы не ставим перед собой цель осуществить

систематизацию всех возможных устройств, которые могут быть подключены к вычислительной системе. Единственное, для чего нам понадобится эта классификация, так это для иллюстрации того положения, что устройства могут быть разделены на группы по выполняемым ими функциям, и для понимания функций драйверов, и интерфейса между ними и базовой подсистемой ввода-вывода.

Для этого мы рассмотрим только две группы устройств: символьные и блочные. Как уже упоминалось в предыдущем разделе, символьные устройства – это устройства, которые умеют передавать данные только последовательно, байт за байтом, а блочные устройства – это устройства, которые могут передавать блок байтов как единое целое.

К символьным устройствам обычно относятся устройства ввода информации, которые спонтанно генерируют входные данные: клавиатура, мышь, модем, джойстик. К ним же относятся и устройства вывода информации, для которых характерно представление данных в виде линейного потока: принтеры, звуковые карты и т. д. По своей природе символьные устройства обычно умеют совершать две общие операции: ввести символ (байт) и вывести символ (байт) – get и put.

Для блочных устройств, таких как магнитные и оптические диски, ленты и т. п. естественными являются операции чтения и записи блока информации – read и write, а также, для устройств прямого доступа, операция поиска требуемого блока информации – seek.

Драйверы символьных и блочных устройств должны предоставлять базовой подсистеме ввода-вывода функции для осуществления описанных общих операций. Помимо общих операций, некоторые устройства могут выполнять операции специфические, свойственные только им – например, звуковые карты умеют увеличивать или уменьшать среднюю громкость звучания, дисплеи умеют изменять свою разрешающую способность. Для выполнения таких специфических действий в интерфейс между драйвером и базовой подсистемой ввода-вывода обычно входит еще одна функция, позволяющая непосредственно передавать драйверу устройства произвольную команду с произвольными параметрами, что позволяет задействовать любую возможность драйвера без изменения интерфейса. В операционной системе Unix такая функция получила название ioctl (от input-output control).

Помимо функций read, write, seek (для блочных устройств), get, put (для символьных устройств) и ioctl, в состав интерфейса обычно включают еще следующие функции.

- Функцию инициализации или повторной инициализации работы драйвера и устройства – open.
- Функцию временного завершения работы с устройством (может, например, вызывать отключение устройства) – close.
- Функцию опроса состояния устройства (если по каким-либо причинам работа с устройством производится методом опроса его состояния, например, в операционных системах Windows NT и Windows 9x так построена работа с принтерами через параллельный порт) – poll.
- Функцию останова драйвера, которая вызывается при останове операционной системы или выгрузке драйвера из памяти, halt.

Существует еще ряд действий, выполнение которых может быть возложено на драйвер, но поскольку, как правило, это действия базовой подсистемы ввода-вывода, мы поговорим о них в следующем разделе. Приведенные выше названия функций, конечно, являются условными и могут меняться от одной операционной системы к другой, но действия, выполняемые драйверами, характерны для большинства операционных систем, и соответствующие функции присутствуют в интерфейсах к ним.

Функции базовой подсистемы ввода-вывода

Базовая подсистема ввода-вывода служит посредником между процессами вычислительной системы и набором драйверов. Системные вызовы для выполнения операций ввода-вывода трансформируются ею в вызовы функций необходимого драйвера устройства. Однако обязанности базовой подсистемы не сводятся к выполнению только действий трансляции общего системного вызова в обращение к частной функции драйвера. Базовая подсистема предоставляет вычислительной системе такие услуги, как поддержка блокирующихся, неблокирующихся и асинхронных системных вызовов, буферизация и кэширование входных и выходных данных, осуществление spooling'a и монопольного захвата внешних устройств, обработка ошибок и прерываний, возникающих при операциях ввода-вывода, планирование последовательности запросов на выполнение этих операций. Давайте остановимся на этих услугах подробнее.

Блокирующиеся, неблокирующиеся и асинхронные системные вызовы

Все системные вызовы, связанные с осуществлением операций ввода-вывода, можно разбить на три группы по способам реализации взаимодействия процесса и устройства ввода-вывода.

- К первой, наиболее привычной для большинства программистов группе относятся блокирующиеся системные вызовы. Как следует из самого названия, применение такого вызова приводит к блокировке инициировавшего его процесса, т. е. процесс переводится операционной системой из состояния исполнение в состояние ожидание. Завершив выполнение всех операций ввода-вывода, предписанных системным вызовом, операционная система переводит процесс из состояния ожидание в состояние готовность. После того как процесс будет снова выбран для исполнения, в нем произойдет окончательный возврат из системного вызова. Типичным для применения такого системного вызова является случай, когда процессу необходимо получить от устройства строго определенное количество данных, без которых он не может выполнять работу далее.
- Ко второй группе относятся неблокирующиеся системные вызовы. Их название не совсем точно отражает суть дела. В простейшем случае процесс, применивший неблокирующийся вызов, не переводится в состояние ожидание вообще. Системный вызов возвращается немедленно, выполнив предписанные ему операции ввода-вывода полностью, частично или не выполнив совсем, в зависимости от текущей ситуации (состояния устройства, наличия данных и т. д.). В более сложных ситуациях процесс может блокироваться, но условием его разблокирования является завершение всех необходимых операций или окончание некоторого промежутка времени. Типичным случаем применения неблокирующегося системного вызова может являться периодическая проверка на поступление информации с клавиатуры при выполнении трудоемких расчетов.
- К третьей группе относятся асинхронные системные вызовы. Процесс, использовавший асинхронный системный вызов, никогда в нем не блокируется. Системный вызов инициирует выполнение необходимых операций ввода-вывода и немедленно возвращается, после чего процесс продолжает свою регулярную деятельность. Об окончании завершения операции ввода-вывода операционная система впоследствии информирует процесс изменением значений некоторых переменных, передачей ему сигнала или сообщения или каким-либо иным способом. Необходимо четко понимать разницу между неблокирующимися и асинхронными вызовами. Неблокирующийся системный вызов для выполнения операции read вернется немедленно, но может прочитать запрошенное количество байтов, меньшее количество или вообще ничего. Асинхронный системный вызов для этой

операции также вернется немедленно, но требуемое количество байтов рано или поздно будет прочитано в полном объеме.

Буферизация и кэширование

Под буфером обычно понимается некоторая область памяти для запоминания информации при обмене данных между двумя устройствами, двумя процессами или процессом и устройством. Обмен информацией между двумя процессами относится к области кооперации процессов, и мы подробно рассмотрели его организацию в соответствующей лекции. Здесь нас будет интересовать использование буферов в том случае, когда одним из участников обмена является внешнее устройство. Существует три причины, приводящие к использованию буферов в базовой подсистеме ввода-вывода.

- Первая причина буферизации – это разные скорости приема и передачи информации, которыми обладают участники обмена. Рассмотрим, например, случай передачи потока данных от клавиатуры к модему. Скорость, с которой поставляет информацию клавиатура, определяется скоростью набора текста человеком и обычно существенно меньше скорости передачи данных модемом. Для того чтобы не занимать модем на все время набора текста, делая его недоступным для других процессов и устройств, целесообразно накапливать введенную информацию в буфере или нескольких буферах достаточного размера и отсылать ее через модем после заполнения буферов.
- Вторая причина буферизации – это разные объемы данных, которые могут быть приняты или получены участниками обмена единовременно. Возьмем другой пример. Пусть информация поставляется модемом и записывается на жесткий диск. Помимо обладания разными скоростями совершения операций, модем и жесткий диск представляют собой устройства разного типа. Модем является символьным устройством и выдает данные байт за байтом, в то время как диск является блочным устройством и для проведения операции записи для него требуется накопить необходимый блок данных в буфере. Здесь также можно применять более одного буфера. После заполнения первого буфера модем начинает заполнять второй, одновременно с записью первого на жесткий диск. Поскольку скорость работы жесткого диска в тысячи раз больше, чем скорость работы модема, к моменту заполнения второго буфера операция записи первого будет завершена, и модем снова сможет заполнять первый буфер одновременно с записью второго на диск.
- Третья причина буферизации связана с необходимостью копирования информации из приложений, осуществляющих ввод-вывод, в буфер ядра операционной системы и обратно. Допустим, что некоторый пользовательский процесс пожелал вывести информацию из своего адресного пространства на внешнее устройство. Для этого он должен выполнить системный вызов с обобщенным названием write, передав в качестве параметров адрес области памяти, где расположены данные, и их объем. Если внешнее устройство временно занято, то возможна ситуация, когда к моменту его освобождения содержимое нужной области окажется испорченным (например, при использовании асинхронной формы системного вызова). Чтобы избежать возникновения подобных ситуаций, проще всего в начале работы системного вызова скопировать необходимые данные в буфер ядра операционной системы, постоянно находящийся в оперативной памяти, и выводить их на устройство из этого буфера.

Под словом кэш (cash – "наличные"), этимологию которого мы не будем здесь рассматривать, обычно понимают область быстрой памяти, содержащую копию данных, расположенных где-либо в более медленной памяти, предназначенную для ускорения работы вычислительной системы. Мы с вами сталкивались с этим понятием при рассмотрении

иерархии памяти. В базовой подсистеме ввода-вывода не следует смешивать два понятия, буферизацию и кэширование, хотя зачастую для выполнения этих функций отводится одна и та же область памяти. Буфер часто содержит единственный набор данных, существующий в системе, в то время как кэш по определению содержит копию данных, существующих где-нибудь еще. Например, буфер, используемый базовой подсистемой для копирования данных из пользовательского пространства процесса при выводе на диск, может в свою очередь применяться как кэш для этих данных, если операции модификации и повторного чтения данного блока выполняются достаточно часто.

Функции буферизации и кэширования не обязательно должны быть локализованы в базовой подсистеме ввода-вывода. Они могут быть частично реализованы в драйверах и даже в контроллерах устройств, скрытно по отношению к базовой подсистеме.

Spooling и захват устройств

О понятии spooling мы говорили в первой лекции нашего курса, как о механизме, впервые позволившем совместить реальные операции ввода-вывода одного задания с выполнением другого задания. Теперь мы можем определить это понятие более точно. Под словом spool мы подразумеваем буфер, содержащий входные или выходные данные для устройства, на котором следует избегать чередования его использования (возникновения interleaving – см. раздел "Interleaving, race condition и взаимоисключения" лекции 5) различными процессами. Правда, в современных вычислительных системах spool для ввода данных практически не используется, а в основном предназначен для накопления выходной информации.

Рассмотрим в качестве внешнего устройства принтер. Хотя принтер не может печатать информацию, поступающую одновременно от нескольких процессов, может оказаться желательным разрешить процессам совершать вывод на принтер параллельно. Для этого операционная система вместо передачи информации напрямую на принтер накапливает выводимые данные в буферах на диске, организованных в виде отдельного spool-файла для каждого процесса. После завершения некоторого процесса соответствующий ему spool-файл ставится в очередь для реальной печати. Механизм, обеспечивающий подобные действия, и получил название spooling.

В некоторых операционных системах вместо использования spooling для устранения race condition применяется механизм монопольного захвата устройств процессами. Если устройство свободно, то один из процессов может получить его в монопольное распоряжение. При этом все другие процессы при попытке осуществления операций над этим устройством будут либо блокированы (переведены в состояние ожидание), либо получат информацию о невозможности выполнения операции до тех пор, пока процесс, захвативший устройство, не завершится или явно не сообщит операционной системе о своем отказе от его использования.

Обеспечение spooling и механизма захвата устройств является прерогативой базовой подсистемы ввода-вывода.

Обработка прерываний и ошибок

Если при работе с внешним устройством вычислительная система не пользуется методом опроса его состояния, а задействует механизм прерываний, то при возникновении прерывания, как мы уже говорили раньше, процессор, частично сохранив свое состояние, передает управление специальной программе обработки прерывания. Мы уже рассматривали действия операционной системы над процессами, происходящими при возникновении прерывания, в разделе "Переключение контекста" лекции 2, где после возникновения прерывания осуществлялись следующие действия: сохранение контекста, обработка

прерывания, планирование использования процессора, восстановление контекста. Тогда мы обращали больше внимания на действия, связанные с сохранением и восстановлением контекста и планированием использования процессора. Теперь давайте подробнее остановимся на том, что скрывается за словами "обработка прерывания".

Одна и та же процедура обработки прерывания может применяться для нескольких устройств ввода-вывода (например, если эти устройства используют одну линию прерываний, идущую от них к контроллеру прерываний), поэтому первое действие собственно программы обработки состоит в определении того, какое именно устройство выдало прерывание. Зная устройство, мы можем выявить процесс, который инициировал выполнение соответствующей операции. Поскольку прерывание возникает как при удачном, так и при неудачном ее выполнении, следующее, что мы должны сделать, – это определить успешность завершения операции, проверив значение бита ошибки в регистре состояния устройства. В некоторых случаях операционная система может предпринять определенные действия, направленные на компенсацию возникшей ошибки. Например, в случае возникновения ошибки чтения с гибкого диска можно попробовать несколько раз повторить выполнение команды. Если компенсация ошибки невозможна, то операционная система впоследствии известит об этом процесс, запросивший выполнение операции, (например, специальным кодом возврата из системного вызова). Если этот процесс был заблокирован до выполнения завершившейся операции, то операционная система переводит его в состояние готовность. При наличии других неудовлетворенных запросов к освободившемуся устройству операционная система может инициировать выполнение следующего запроса, одновременно известив устройство, что прерывание обработано. На этом, собственно, обработка прерывания заканчивается, и система может приступать к планированию использования процессора.

Действия по обработке прерывания и компенсации возникающих ошибок могут быть частично переложены на плечи соответствующего драйвера. Для этого в состав интерфейса между драйвером и базовой подсистемой ввода-вывода добавляют еще одну функцию – функцию обработки прерывания intr.

Планирование запросов

При использовании неблокирующегося системного вызова может оказаться, что нужное устройство уже занято выполнением некоторых операций. В этом случае неблокирующийся вызов может немедленно вернуться, не выполнив запрошенных команд. При организации запроса на совершение операций ввода-вывода с помощью блокирующегося или асинхронного вызова занятость устройства приводит к необходимости постановки запроса в очередь к данному устройству. В результате с каждым устройством оказывается связан список неудовлетворенных запросов процессов, находящихся в состоянии ожидания, и запросов, выполняющихся в асинхронном режиме. Состояние ожидание расщепляется на набор очередей процессов, дожидающихся различных устройств ввода-вывода.

После завершения выполнения текущего запроса операционная система (по ходу обработки возникшего прерывания) должна решить, какой из запросов в списке должен быть удовлетворен следующим, и инициировать его исполнение. Точно так же, как для выбора очередного процесса на исполнение из списка готовых нам приходилось осуществлять краткосрочное планирование процессов, здесь нам необходимо осуществлять планирование применения устройств, пользуясь каким-либо алгоритмом этого планирования. Критерии и цели такого планирования мало отличаются от критериев и целей планирования процессов.

Задача планирования использования устройства обычно возлагается на базовую подсистему ввода-вывода, однако для некоторых устройств лучшие алгоритмы планирования

могут быть тесно связаны с деталями их внутреннего функционирования. В таких случаях операция планирования переносится внутрь драйвера соответствующего устройства, так как эти детали скрыты от базовой подсистемы. Для этого в интерфейс драйвера добавляется еще одна специальная функция, которая осуществляет выбор очередного запроса, – функция strategy.

ПРАКТИЧЕСКИЙ РАЗДЕЛ

ЛАБОРАТОРНАЯ РАБОТА №1 «ИНТЕРФЕЙС. ФАЙЛЫ. КОМАНДЫ»

Часть 1. CLI — Command-Line Interface

Для его изучения включите терминал (Приложения > Стандартные > Терминал).

Работу ОС LINUX можно представить в виде функционирования множества взаимосвязанных процессов. При загрузке системы сначала запускается ядро (процесс 0).

Взаимодействие пользователя с системой LINUX происходит в интерактивном режиме посредством командного языка. Оболочка операционной системы – shell – интерпретирует вводимые команды, запускает соответствующие программы (процессы), формирует и выводит ответные сообщения.

Shell - это интерфейс, обеспечивающий взаимодействие между ядром и пользователем. Интерфейс shell очень прост. Обычно он состоит из приглашения, по которому пользователь вводит команды и нажимает клавишу Enter. Строка, в которой вы набираете команду, называется командной строкой. Shell не только интерпретирует команды, но и создает среду, которую вы можете конфигурировать и программировать. У shell есть свой язык программирования, который позволяет писать программы, содержащие достаточно сложные последовательности команд Linux. Язык программирования shell обладает многими свойствами обычного языка программирования, в частности в нем предусмотрено использование циклов и условных переходов. Каждому пользователю системы Linux предоставляется свой собственный пользовательский интерфейс, или shell. Пользователи могут модифицировать свои shell в соответствии с конкретными потребностями. В этом смысле shell пользователя функционирует скорее как операционная среда, которой пользователь может управлять по своему усмотрению.

За последние годы разработано несколько разновидностей shell. Сейчас используются в основном три варианта: Bourne, Korn и C-shell. Bourne-shell был разработан в Bell Labs для System V. C-shell разработан для версии BSD. Kom-shell - это усовершенствованный вариант Bourne-shell. В современных версиях Unix, включая Linux, представлены все три вышеназванных shell, что дает пользователю возможность выбора. В Linux, однако, используются расширенные или общедоступные версии этих shell: Bourne Again, TC-shell и Public Domain Korn. При запуске ОС Linux активизируется Bourne Again Shell, модифицированная версия Bourne. Отсюда можно переключаться в другие shell.

Файловая структура: каталоги и файлы

В операционной системе Linux все файлы организованы в каталоги, которые, в свою очередь, иерархически соединены друг с другом, образуя одну общую файловую структуру. При обращении к файлу необходимо указывать не только его имя, но и место, которое он занимает в этой файловой структуре. Можно создавать любое количество новых каталогов, добавляя их к файловой структуре. Команды работы с файлами ОС Linux могут выполнять сложные операции, например, перемещение и копирование целых каталогов вместе с их подкаталогами. Такие команды, как find, cp, mv и ln, позволяют находить файлы, копировать их и перемещать из одного каталога в другой, а также создавать ссылки.

Файлы в операционной системе Linux организованы в иерархическую систему каталогов. Каталог может содержать файлы и другие каталоги. В этом смысле каталоги выполняют две важные функции. Во-первых, в каталоге хранятся файлы, подобно папкам в ящике картотеки, а во-вторых, каталог соединяется с другими каталогами, как ветвь

дерева соединяется с другими ветвями. По отношению к файлам каталоги выполняют роль ящиков картотеки, в каждом из которых хранится несколько папок. Для того чтобы взять одну из них, нужно открыть ящик. Следует отметить, однако, что, в отличие от ящиков картотеки, каталоги могут содержать не только файлы, но и другие каталоги. Именно таким образом каталог может соединяться с другим каталогом. Из-за сходства с деревом такую структуру часто называют древовидной структурой. Если быть более точным, то эта

структура скорее похожа не на дерево, а на перевернутый вверх корнями куст. Ствола здесь нет, и изображается дерево перевернутым, при этом корень находится наверху. Вниз от корня отходят ветви. Каждая ветвь отходит только от одной ветви, а от нее самой может отходить множество ветвей нижнего уровня. В этом смысле данную структуру можно назвать структурой "родители-потомки".

Аналогичным образом любой каталог является подкаталогом другого каталога. Каждый каталог может содержать множество подкаталогов, но сам должен быть потомком только одного родительского каталога. Вверху файловой системы находится корневой каталог (обозначается символом "косая черта"), от которого ответвляются другие каталоги. Каждый каталог может содержать несколько других каталогов или

файлов, но родительский каталог у него всегда бывает только один.

В каталоге chris, например, организованы два подкаталога, reports и programs. Сам же каталог chris соединен только с одним родительским каталогом, home. Файловая структура ОС Linux разветвляется на несколько каталогов, начиная с корневого, /. В корневом каталоге имеется несколько системных каталогов, которые содержат файлы и программы, относящиеся к самой ОС Linux. Корневой каталог, кроме того, содержит каталог home, который может содержать начальные каталоги всех пользователей системы. Начальный каталог каждого пользователя, в свою очередь, будет включать в себя каталоги, который пользователь создает для своих нужд. Каждый из этих каталогов тоже может содержать каталоги. Все эти вложенные каталоги ответвляются от начального каталога пользователя.

Получить доступ к каталогу можно либо по имени, либо сделав его каталогом по умолчанию. Каждому каталогу при создании присваивается имя. Этим именем можно пользоваться для доступа к файлам, находящимся в данном каталоге. Если при проведении какой-либо операции над файлами имена каталогов не указываются, то используется каталог по умолчанию, который называют рабочим каталогом. В этом смысле рабочий каталог - это каталог, в котором вы в данный момент работаете. При регистрации в системе в качестве рабочего принимается ваш начальный каталог, имя которого обычно совпадает с вашим регистрационным именем.

Рабочий каталог можно заменить с помощью команды cd. В процессе замены рабочего каталога вы переходите из одного каталога в другой. Каталог можно рассматривать как коридор, в который выходит множество дверей с табличками. Некоторые двери ведут в комнаты, а некоторые - в другие коридоры. Двери, ведущие в комнаты, - это файлы, находящиеся в каталоге, а двери, ведущие в коридоры, - это другие каталоги. Переходя из одного коридора в другой, вы меняете рабочий каталог. Проходя по нескольким коридорам, вы перемещаетесь по нескольким каталогам.

Путевые имена

Имя, которое дается каталогу или файлу при его создании, не является полным. Полным именем каталога является его путевое имя. Иерархические связи, существующие между каталогами, образуют пути, и эти пути можно использовать для однозначного указания каталога или файла и обращения к нему. Можно сказать, что каждый каталог в файловой структуре имеет собственный уникальный путь. Фактическое имя, которым система обозначает каталог, всегда начинается с корневого каталога и состоит из имен всех каталогов, ведущих к данному каталогу.

В ОС Linux путевое имя каталога состоит из имен всех каталогов, образующих путь. Эти имена отделяются друг от друга символами "косая черта". Косая черта перед первым каталогом пути обозначает корневой каталог(/).

Путевые имена могут быть абсолютными и относительными. Абсолютное путевое имя - это полное имя файла или каталога, начинающееся символом корневого каталога. Относительное путевое имя начинается символом рабочего каталога и представляет собой обозначение пути к файлу относительно вашего рабочего каталога.

Системные каталоги

Корневой каталог, являющийся началом файловой структуры ОС Linux, содержит ряд системных каталогов. Системные каталоги содержат файлы и программы, служащие для управления системой и ее сопровождения. Многие из этих каталогов содержат подкаталоги с программами, предназначенными для выполнения конкретных задач.

/bin

bin - это сокращенно от `binaries' (т.е. двоичные или выполняемые файлы). Здесь находится много важных системных программ. Большинство основных команд Unix находятся в этом каталоге.

/dev

"Файлы" в dev известны как драйверы устройств - они используются для доступа к устройствам и ресурсам системы, таким как диски, модемы, память и т.д. Например, вы можете читать данные из файла, точно также вы можете читать входные сигналы от мыши, имея доступ к /dev/mouse. Имена файлов, начинающиеся на fd - это дисководы гибких дисков. fd0 - первый дисковод, fd1 - второй.

/etc

etc содержит файлы конфигурации системы. Например /etc/passwd(файл паролей), /etc/group (файл групп), /etc/rc (командный файл инициализации) и т.д.

/sbin

В sbin находятся важные исполняемые системные файлы, используемые системным администратором.

/home

home содержит домашние каталоги пользователей.

/lib

lib содержит образы разделяемых библиотек (shared library images). Эти файлы содержат код, который могут использовать многие программы. Вместо того, чтобы каждая программа имела свою собственную копию этих выполняемых файлов, они хранятся в одном общедоступном месте – в /lib. Это позволяет сделать выполняемые файлы меньше и сэкономит место в системе.

Команды

Прежде чем перейти к рассмотрению конкретных команд, дадим определение команде. Пользователям, вышедшим из среды DOS, это понятие знакомо: команда - основа главных функций операционной системы. Из команд DIR, COPY или ATTRIB составляются довольно сложные процедуры, оформляемые в виде bat-файлов (командных файлов).

Однако в DOS, как и в других операционных системах, количество команд ограниченно и статично — пользователь не может вводить собственные команды. В мире Unix (следовательно, и Linux) понятие команды несколько иное. Здесь команда - это любой выполняемый файл. Командой является любой файл, предназначенный для выполнения, а не для хранения данных или конфигурационных параметров. Любой выполняемый файл, записанный в систему, становится ее командой. Коротко перечислим средства группирования команд:

- *cmd1 arg ...; cmd2 arg ...; ... cmdN arg ...* - последовательное выполнение команд;
- *cmd1 arg ...& cmd2 arg ...& ... cmdN arg ...* - асинхронное выполнение команд;
- *cmd1 arg ... && cmd2 arg ...* - зависимость последующей команды от предыдущей таким образом, что последующая команда выполняется, если предыдущая выдала нулевое значение;
- *cmd1 arg ...* || *cmd2 arg ...* - зависимость последующей команды от предыдущей таким образом, что последующая команда выполняется, если предыдущая выдала ненулевое значение.

Управление каталогами

Создание и удаление каталогов: mkdir и rmdir

Каталоги создаются и удаляются соответственно командой mkdir и командой rmdir. В том и другом случае можно использовать путевые имена каталогов. В следующем примере пользователь сначала создает каталог reports, а затем, используя абсолютное путевое имя - каталог letters. (Здесь и далее символ $ означает приглашение к вводу и не относится к командам).

```
$ mkdir reports
$ mkdir /home/chris/letters
```

Для удаления каталога нужно дать команду rmdir с именем этого каталога. В приведенном ниже примере пользователь сперва удаляет командой rmdir каталог reports, а затем - указав абсолютное путевое имя - каталог letters.

```
$ rmdir reports
$ rmdir /home/chris/letters
```

Просмотр содержимого каталогов: ls

Чтобы с помощью команды ls можно получить список файлов и каталогов, находящихся в рабочем каталоге, необходимо выполнить следующую команду.

```
$ ls
```

Для того чтобы имена файлов и имена каталогов различались между собой, эту команду нужно дать с опцией -F. В этом случае после каждого имени каталога в списке ставится косая черта.

```
$ ls
weather reports letters
$ ls -F
weather reports/ letters/
```

В качестве аргумента команда ls может использовать имя или путевое имя каталога. Это позволяет получить список файлов любого каталога, не переходя в него. В следующем примере команда ls использует в качестве аргумента имя каталога reports. Затем она выполняется еще раз, но уже с абсолютным путевым именем этого каталога.

```
$ ls reports
monday tuesday
$ ls /home/chris/reports
monday tuesday
```

Переход в другой каталог: команда cd

Переход из одного каталога в другой осуществляется командой cd. Переход в каталог делает его рабочим. Файловые команды, например Ls, будут манипулировать файлами, находящимися именно в рабочем каталоге, если иного не указано.В качестве аргумента команда cd использует имя каталога, в который вы хотите перейти.

```
$ cd имя_каталога
```

Все создаваемые каталоги будут находиться в рабочем каталоге. Рабочий каталог

является для вновь созданного каталога родительским. Для обозначения родительского каталога можно пользоваться двумя точками (..). Этот специальный символ обозначает путевое имя родительского каталога. Его допускается использовать в команде cd для перехода обратно в родительский каталог, таким образом вновь делая этот каталог рабочим.

Если вы хотите вернуться в начальный каталог, нужно ввести команду cd без аргумента. Вы вернетесь прямо в свой начальный каталог, и он вновь станет рабочим.

Путевые имена: команда pwd

В каждом каталоге можно создавать другие каталоги, осуществляя, по сути дела, вложение одного каталога в другой. Команда cd позволяет переходить из одного каталога в другой, однако, никакого указателя на то, в каком каталоге вы в данный момент находитесь, нет. Для того чтобы определить, в какой каталог вы перешли, дайте команду pwd, которая сообщит абсолютное путевое имя рабочего каталога, как показано в следующем примере. Путевое имя состоит из имен рабочего каталога dylan и каталога, частью которого он является, home. Имена каталогов разделены косой чертой. Корневой каталог обозначен первой косой чертой.

```
$ pwd
/home/dylan
```

Обращение к рабочему и родительскому каталогам: . и ..

Каждый каталог обязательно имеет родительский каталог (за исключением, естественно, корневого каталога). При создании каталога в нем сразу же делаются две записи. Одна из них будет представлена точкой (.), а вторая - двумя точками (..). Точка обозначает путевое имя данного каталога, а две точки - путевое имя его родительского каталога. Две точки, используемые как аргумент команды, обозначают родительский каталог. Одна точка обозначает рабочий каталог.

Точка используется для обозначения рабочего каталога вместо указания его путевого имени. Например, для копирования файла в рабочий каталог с сохранением имени файла можно вместо путевого имени рабочего каталога поставить точку. В этом смысле точка - еще одно имя рабочего каталога.

Символ .. часто используется для обозначения файлов родительского каталога. Используя команду cd с символом .., можно возвращаться из каталога нижнего уровня, последовательно переходя в родительские каталоги по дереву каталогов.

Во многих случаях в команде допускается использование обоих символов. Например, если letters - рабочий каталог и нужно скопировать в него файл weather, то каталог chris можно обозначить двумя точками, а каталог letters - одной:

```
$ cp ../weather .
```

Использование абсолютных и относительных путевых имен: ~

Как упоминалось выше, файлы и каталога можно обозначать абсолютными и относительными путевыми именами. У обоих вариантов, однако, есть свои недостатки. Абсолютное путевое имя пригодно для обозначения любого файла и каталога, но такие имена, как правило, очень длинные и сложные, что затрудняет работу с ними. Относительное путевое имя короче и проще в работе, но число файлов, которые им можно обозначить, ограниченно. Как правило, относительные путевые имена нужно использовать при каждой возможности, а абсолютные - только в случае необходимости. В некоторых shell предусмотрена возможность сокращения абсолютных путевых имен. Относительные путевые имена применяют для обозначения только файлов, находящихся в подкаталогах рабочего каталога. Этих подкаталогов, вложенных один в другой, может быть сколь угодно много, но их пути должны ответвляться от рабочего каталога. Допустим, вам нужно обратиться к каталогу, расположенному по дереву каталогов выше рабочего или в другой ветви, тогда необходимо

использовать абсолютное путевое имя.

Диалоговое руководство

В системе Linux используются различные утилиты, среди которых редакторы, программы-почтальоны и руководства. Эти утилиты представляют собой отдельные программы, имеющие собственные интерфейсы с собственными наборами команд. Примером такой утилиты является диалоговое руководство man, которое позволяет пользователю получить информацию о любой команде и программе ОС Linux. Для обращения к диалоговому руководству введите команду man и имя команды, информация о которой вам нужна. Ниже приведен пример, в котором пользователь вызывает из диалогового руководства информацию о команде ls.

```
$ man ls
```

Команды whatis и apropos

Команды whatis и apropos обеспечивают поиск в базе данных заголовков man страниц и выдают все найденные заголовки с кратким описанием каждого. Команда whatis позволяет искать заголовки по целым словам. Например, если вы хотите увидеть все пункты руководства, в которых есть отдельно стоящая буква x, необходимо дать следующую команду (она выдаст вам все пункты руководства, в названиях которых упоминается X Window):

```
$ whatis x
X (3) - a portable, network-transparent window system
X Consortium (3) - X Consortium information
X Standards (3) - X Consortium Standards
X security (3) - x display access control
x (3) - a Portable, network-transparent window system
X Consortium (3) - X Consortium information
X Standards (3) - x Consortium Standards
X security (3) - X display access control
(END)
$
```

С помощью этих команд осуществляется постраничный вывод результатов поиска. Если выводимые данные занимают несколько страниц можно перемещаться по ним с помощью клавиш f и Ь. Допускается и выполнение поиска по образцу (клавишами / и ?). Для выхода нажмите клавишу q Лишь после этого вы вернетесь в командную строку.Команда apropos выполняет ту же задачу, что и команда whatis но поиск выполняется по образцу, а не по целым словам. Скажем, команда apropos x выдаст несколько страниц с данными, в которых будут перечислены все пункты руководства, начинающиеся с буквы x, например xwpe и xloadimage. В следующем примере выдается перечень пунктов руководства начинающихся с комбинации ls. Сюда входит команда ls и другие команды, например lseek и lsearch.

```
$ apropos ls
ls, dir, vdir (1) - list contents of directories lsattr (1) - list file attributes on a Linux second extended file system
lsearch (n) - see if a list contains a particular element
lseek (2) - reposition read/write file offset lsort (n) - sort the elements of a list lsattr (1) - list file attributes on a Linux second extended file system
lsearch (n) - See if a list contains a particular element
lseek (2) - reposition read/write file offset lsort (n) - Sort the elements of a list (END)
$
```

Отображение файлов: cat и more

Во многих случаях бывает необходимо просматривать содержимое файла. Команды cat и more выводят содержимое файла на экран. Название команды cat образовано путем сокращения слова concatenate. Это очень сложная и универсальная команда. В нижеследующем примере употребляется в очень узких рамках, только для вывода текста файла на экран:

```
$ cat mydata
computers
```

Команда cat выводит на экран сразу весь текст файла. Если файл имеет большой размер, то текст очень быстро мелькает на экране. Для устранения этого недостатка служит команда more, с помощью которой текст на экран можно выводить порциями. Эта команда вызывается с именем файла, который вы хотите просмотреть:

```
$ more mydata
```

Когда more вызывает файл, отображается его первый фрагмент, умещающийся на экране. Для отображения следующего фрагмента нажимается клавиша f или клавиша пробела. Для возврата к предыдущему тексту используется клавиша b. Нажав клавишу q, можно в любой момент выйти из данной программы.

Операции с файлами и каталогами: find, cp, mv, rm, ln

По мере создания файлов возникает необходимость снятия с них резервных копий, изменения их имен, удаления некоторых из них и даже присваивания им дополнительных имен. В ОС Linux предусмотрен набор команд, которые обеспечивают поиск, копирование, переименование и удаление файлов. Команды представляют собой сокращенную форму слов, состоящую из двух символов. Команда cp обозначает "copy" и позволяет копировать файл, mv обозначает "move" и дает возможность перемещать либо переименовывать файл, rm обозначает "remove" и приводит к удалению файла и, наконец. Ln обозначает "link" и позволяет дать файлу еще одно имя. Исключение из этого правила - команда find, с помощью которой осуществляется поиск файла в списке имен.

Поиск в каталогах: команда find

Если вы используете много файлов, разбросанных по разным каталогам, то для выявления одного из них или нескольких файлов определенного типа можно провести поиск. Эта функция осуществляется с помощью команды find. В качестве аргументов в ней используются имена каталогов, за которыми следуют несколько опций, задающих тип и критерии поиска. Команда find позволяет производить поиск в перечисленных каталогах и их подкаталогах, отыскивая файлы, соответствующие указанным критериям. Команда find дает возможность искать файлы по имени, типу, владельцу и даже по времени последнего изменения.

```
$ find список каталогов -опция критерии
```

Копирование файлов

Для того чтобы создать копию файла, нужно указать команде cp два имени файла. Первое из них - имя копируемого файла, который уже существует. Этот файл часто называют исходным. Второе - имя, которое вы хотите присвоить копии. Это будет новый файл, содержащий копию всех данных исходного файла. Его часто называют выходным файлом. Команда cp имеет следующий синтаксис:

$ cp исходный файл выходной файл

В следующем примере пользователь копирует файл proposal в новый файл, oldprop.

$ cp proposal oldprop

Когда пользователь запросит перечень файлов, содержащихся в каталоге, среди них будет новая копия.

$ ls proposal oldprop

Может случиться так, что при копировании файла с помощью команды cp вы непреднамеренно разрушите другой файл. При создании копии посредством этой команды сначала создается файл, а затем в него копируются данные. Если какой-нибудь файл уже имеет то имя, которое вы указали для выходного файла, первый из них разрушается и создается новый файл с этим именем. В некотором смысле можно сказать, что файл-оригинал перезаписывается новой копией. В следующем примере файл proposal перезаписывается новой копией (потому что файл с таким именем уже существовал).

$ cp newprop proposal

Чтобы выявить подобные случаи лучше пользоваться командой cp с опцией -i. Такая команда сначала проверяет, существует ли файл под указанным именем. Если да, то программа спросит у вас, хотите ли вы перезаписать этот файл. Если вы ответите y, то существующий файл будет разрушен, и программа создаст новый файл в качестве его копии. Если вы дадите другой ответ, он будет считаться отрицательным и выполнение команды cp будет прервано, а файл-оригинал сохранен.

```
$ cp -i newprop proposal
Overwrite proposal? n
$
```

Копирование файлов в каталоги

Для того чтобы скопировать файл из рабочего каталога в другой каталог, нужно указать имя этого каталога команде cp в качестве второго аргумента. Имя новой копии будет таким же, как у оригинала, но находиться она будет в другом каталоге. Файлы в разных каталогах могут иметь одинаковые имена.

$ cp имена_файлов имя_каталога

Для того чтобы скопировать файл из начального каталога в подкаталог, просто укажите имя этого каталога. В следующем примере файл newprop копируется из рабочего каталога в каталог props.

$ cp newprop props

Команда cp может использовать в качестве аргументов имена многих файлов, заданные в виде списка, поэтому можно одновременно копировать в каталог несколько файлов. Введите имена этих файлов в командной строке, причем имя каталога должно быть последним аргументом. Все эти файлы копируются в указанный каталог. В следующем примере пользователь копирует файлы preface и doc1 в каталог props. Обратите внимание:

props - последний аргумент.

$ cp preface doc1 props

При создании списка имен файлов для команды cp или команды mv можно использовать любые специальные символы. Пусть, например, вам нужно скопировать в заданный каталог все файлы с исходными текстами программ, написанными на языке С. Вместо того чтобы указывать в командной строке все эти файлы, можно ввести специальный символ * с расширением.с, обозначая тем самым все файлы с расширением.с (т.е. все файлы исходных текстов С-программ) и формируя их список. В следующем примере пользователь копирует все файлы исходных текстов программ из текущего каталога в каталог sourcebks.

$ cp *. c sourcebks

При копировании файла можно дать копии имя, отличное от имени оригинала. Для этого нужно поместить новое имя файла после косой черты, следующей вслед за именем каталога.

$ cp имя_файла имя_каталога/новое_имя файла

В следующем примере файл newprop копируется в каталог props и копии присваивается имя version1. Затем пользователь переходит в каталог props и получает список файлов. В нем имеется только один файл, который называется version1.

$ cd newprop props/version1
$ cd props
$ ls version1

Если нужно скопировать файл из дочернего каталога, например, из props, в родительский каталог, нужно указать имя этого дочернего каталога. Первый аргумент команды cp - имя копируемого файла. Перед ним должно через косую черту стоять имя дочернего каталога. Второй аргумент - имя, которое файл будет иметь в родительском каталоге.

$ cp имя_дочернего_каталога/имя_файла новое_имя_файла

В следующем примере файл version1 копируется из каталога props в начальный каталог:

$ cp props/version1 version1

Предположим теперь, что вы хотите скопировать файл из дочернего каталога в родительский. Вам нужно как-то указать на этот родительский каталог. Это можно сделать двумя точками, которые обозначают путевое имя родительского каталога:

$ cp имя файла ..
$ cp имя_файла .. /новое__имя_файла

Перемещение файлов

С помощью команды mv можно либо изменить имя файла, либо переместить файл из одного каталога в другой. Используя mv для переименования файла, в качестве второго аргумента нужно указать новое имя файла. Первый аргумент - текущее имя файла.

$ mv текущее имя_файла новое_имя_файла

В следующем примере имя файла proposal меняется на version1.

$ mv proposal version1

Как и при использовании команды cp, здесь можно очень просто совершить ошибку, удалив нужный файл. Переименовывая файл, вы можете выбрать имя, которое уже носит другой файл, и этот файл будет удален. Команда mv тоже имеет опцию -i, которая сначала проверяет, существует ли файл с указанным именем. Если да, программа спросит, хотите ли вы перезаписать его. В следующем примере файл с именем version1 уже существует. Программа обнаруживает, что будет осуществлена перезапись, и спрашивает, хотите вы это сделать или нет.

```
$ ls
proposal version1
$ mv -i version1 proposal
Overwrite proposal? n
$
```

Файл можно перенести из одного каталога в другой. Для этого нужно в качестве второго аргумента в команде mv поставить имя каталога. В данном случае можно считать, что команда mv не переименовывает файл, а просто перемещает его из одного каталога в другой.

После перемещения файла у него останется то имя, которое он носил в исходном каталоге (если вы не укажете иного).

$ mv имя_файла имя_каталога

В следующем примере файл newprop перемещается из начального каталога в каталог props.

$ mv newprop props

Если при перемещении файла вы хотите переименовать его, укажите новое имя файла после имени каталога. Имя каталога отделяется от нового имени файла косой чертой. В следующем примере файл newprop перемещается в каталог props и получает имя version1.

```
$ mv newprops props/version1
$ cd props
$ ls version1
```

Перемещение и копирование каталогов

Система Linux позволяет копировать и перемещать целые каталоги. В качестве первого аргумента команды cp и mv могут использовать имя каталога, позволяя копировать и перемещать подкаталоги из одного каталога в другой. Первый аргумент - имя перемещаемого или копируемого каталога, а второй - имя каталога, в который он будет помещен. При перемещении и копировании каталогов действует та же структура путевых имен, что и при соответствующих операциях с файлами.

Подкаталоги можно так же легко, как и файлы, копировать из одного каталога в другой. Для копирования каталога команду cp необходимо использовать с опцией -r (сокращение от recursive, т.е. "рекурсивный"). Эта опция дает команде cp указание копировать

каталог вместе со всеми его подкаталогами. Другими словами, копируется все поддерево каталогов, начиная с указанного. В следующем примере каталог thankyou копируется в каталог oldletters. После завершения этой операции начинают равноправно сосуществовать два подкаталога thankyou: один в каталоге letters, другой в oldletters.

```
$ cp -r letters/thankyou oldletters
$ ls -F letters
thankyou/
$ ls -F oldletters
thankyou/
```

Предположим, вы хотите скопировать не каталог, делая его тем самым подкаталогом другого каталога, а только все его файлы. Для копирования всех файлов из одного каталога в другой нужно указать имена этих файлов. Специальный символ * обозначает имена всех файлов и каталогов в данном каталоге. Для того чтобы скопировать все файлы из каталога letters в каталог oldletters, нужно в качестве первого аргумента поставить звездочку, и программа создаст список всех имен файлов, имеющихся в каталоге letters. Если нужно указать путевое имя первого аргумента, сделайте это, а звездочку поставьте в конце. В следующем примере все файлы из каталога letters копируются в каталог oldletters. Для letters указано путевое имя, а звездочка в конце этого имени обозначает все файлы в данном каталоге.

```
$ cp letters/* oldletters
```

Если вы хотите, чтобы операция копирования осуществлялась и над подкаталогами, нужно указать опцию -r.

```
$ cp -r letters/* oldletters
```

Специальный символ ~

Вы уже знаете, как обозначать тильдой абсолютное путевое имя начального каталога. Например, при копировании файла из нижестоящего каталога в начальный каталог тильдой можно обозначить абсолютное путевое имя начального каталога. В приведенном ниже примере пользователь переходит в каталог reports, а затем копирует из него файл monday в начальный каталог.

```
$ cd reports
$ cp monday ~
```

Для того чтобы при копировании файла в начальный каталог дать ему новое имя, поставьте новое имя после пары символов ~/. В следующем примере файл monday копируется в начальный каталог, и копия получает имя today.

```
$ cp monday ~/today
```

Удаление файла: команда rm

В процессе работы с ОС Linux число используемых файлов будет стремительно возрастать. Появляются новые файлы в этой системе очень часто. Многие из них создаются при работе различных приложений, скажем, редакторов, и с помощью команд, например cp. Постепенно некоторые из этих файлов устаревают. Их можно удалить посредством команды rm. В следующем примере пользователь удаляет файл oldprop.

```
$ rm oldprop
```

Команда rm может быть использована с любым числом аргументов, что позволяет одновременно удалить несколько файлов. Имена этих файлов указываются в командной строке после имени команды.

```
$ rm proposal version1 version2
```

Командой rm следует пользоваться осторожно, так как отменить ее действие нельзя. Если файл удален, восстановить его не удастся.

Предположим, что вы случайно ввели эту команду вместо какой-то другой, например, cp или mv. Когда вы опомнитесь, будет слишком поздно - файлы пропали. Для того чтобы избежать таких ошибок, используйте команду rm с опцией -i, которая инициирует выдачу запроса на подтверждение удаления. Теперь перед удалением каждого файла система будет спрашивать, действительно ли вы хотите удалить его.

Если вы введете y, файл будет удален. При любом ином ответе файл не удаляется. В следующем примере посредством команды rm система получает указание удалить файлы proposal и oldprop, а затем запрашивает подтверждение по каждому из них. Пользователь решает удалить oldprop, а proposal оставить.

```
$ rm -i proposal oldprop
Remove proposal? n
Remove oldprop? y
$
```

Управление файлами

В операционной системе Linux используются разнообразные средства управления файлами и каталогами. Пользователь имеет возможность получить подробную информацию о файлах. Он может, например, узнать, когда они в последний раз корректировались и сколько на них имеется ссылок. Пользователь может управлять доступом к своим файлам. С каждым файлом в ОС Linux связаны права доступа, которые определяют круг лиц, имеющих к нему доступ, и вид доступа. Вы можете разрешить доступ к файлам другим пользователям либо не разрешить такового.

Файлы располагаются на физических устройствах - жестких дисках, flash память, ssd - и на каждом устройстве организуются в файловую систему. Для того чтобы получить доступ к файлам, находящимся на каком-либо устройстве, необходимо присоединить его файловую систему к определенному каталогу. Эта операция называется монтированием файловой системы. Например, для работы с файлами, расположенными на флешке, нужно сначала смонтировать ее файловую систему в определенном каталоге.

В данной главе рассказывается о том, как работать с компакт дисками и разделами жестких дисков. Можно даже обращаться к разделу жесткого диска Microsoft, а также к файловым системам, находящимся на удаленном сервере. В системе предусмотрена возможность создания резервных архивов файлов и передачи архивов по сети в другие системы. Файлы можно сжимать, что позволяет повысить эффективность передачи и обеспечивает экономию дискового пространства.

Архивирование и сжатие широко применяются при получении программных пакетов из удаленных источников. Сжатый заархивированный пакет программ переписывается на жесткий диск, а затем распаковывается и разархивируется. После этого его можно инсталлировать в систему. Именно таким образом пользователи получают большинство новых программных средств ОС Linux.

Вывод информации о файлах: команда ls -l

Команда ls -l позволяет получить подробную информацию о файле.

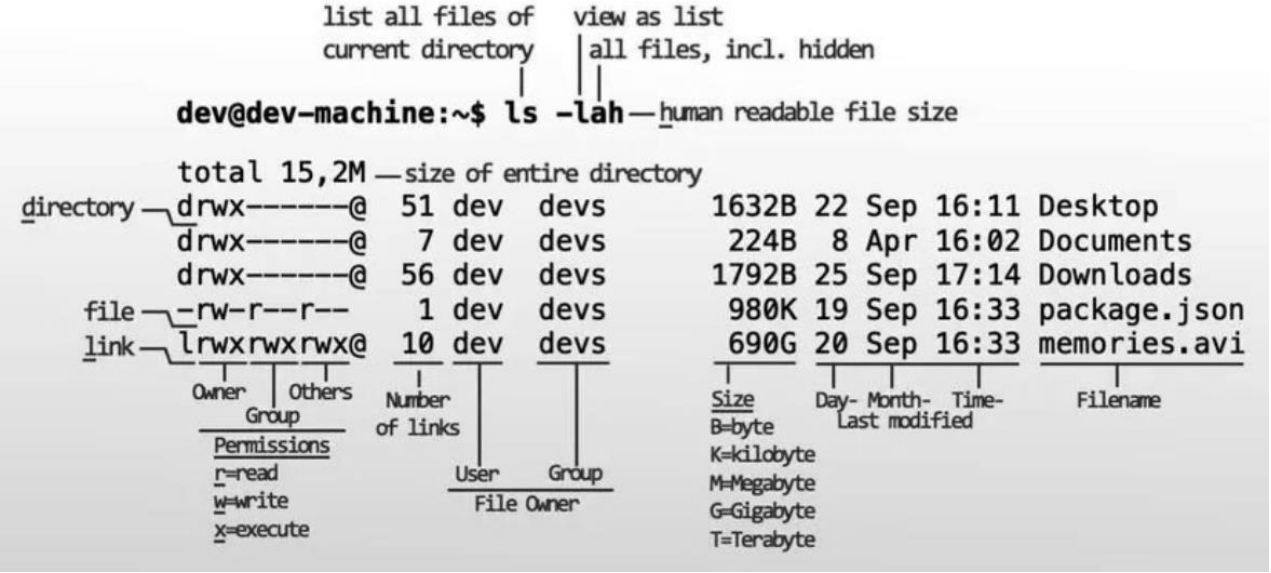
Сначала указываются права доступа, затем количество ссылок, имя владельца файла, имя группы, к которой он относится, размер файла в байтах, дата и время последнего изменения и, наконец, имя файла. Имя группы обозначает группу, которой предоставляется доступ по категории "группа". Например (см. ниже), тип файла mydata - обычный файл. У него всего одна ссылка - это говорит о том, что у файла нет других имен. Имя владельца - chris, оно совпадает с регистрационным именем данного пользователя. Имя группы - weather.

Вероятно, есть и другие пользователи, входящие в эту группу. Размер файла - 207 байт. Последний раз его корректировали 20 февраля в 11:55. Имя файла - mydata.

-rw-r--r-- 1 chris weather 207 Feb 20 11:55 mydata

Если вы хотите получить подобную информацию обо всех файлах, находящихся в данном каталоге, дайте команду ls -l без аргумента:

$ ls -l
-rw-r--r-- chris weather 207 Feb 20 11:55 mydata
-rw-rw-r-- chris weather 568 Feb 14 10:30 today
-rw-rw-r-- chris weather 308 Feb 17 12:40 monday



Часть 2. GUI — Graphic User Interface

Linux чрезвычайно гибок и поддерживает использование множества различных рабочих сред. KDE — самая популярная рабочая среда в мире Linux и занимает это место по достоинству. Она изящно выглядит, весьма удобна и проворна, солидна и обладает богатыми возможностями. Вы можете положиться на нее в деле отправки и получения электронной почты

Задание для выполнения

Часть 1.

1. Определить путевое имя рабочего (начального) каталога. Как обозначается

корневой каталог? Какое путевое имя получили (относительное или абсолютное)?

2. Создать в домашнем каталоге два подкаталога. Просмотреть содержимое рабочего каталога. Просмотреть содержимое родительского каталога, не переходя в него.

3. Используя относительные имена каталогов, выполнить следующие задания. Перейти в системный каталог. Просмотреть его содержимое. Просмотреть содержимое начального каталога. Вернуться в домашний каталог.

4. Удалить созданные ранее подкаталоги одной командой.

5. Получить информацию по командам ls и rm с помощью утилиты man. Изучить структуру man-документа. Опишите структуру документа.

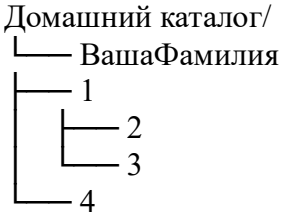
6. Получить информацию по командам ls и rm с помощью утилиты info. Изучить структуру документа. Опишите структуру документа. В чем различие между info и man.

7. Получить краткую информацию по командам ls и cd с помощью команды whatis и apropos. В чем различие?

8. Изучить команду whereis. Применить команду с параметрами ls, cd. Сделать выводы.

Часть 2.

1. Создайте в домашнем каталоге следующую структуру подкаталогов (существующие каталоги не удаляйте!)



2. Скопируйте файл /etc/group в каталог 1 используя абсолютные имена копируемого файла и каталога назначения.

3. Скопируйте файл /etc/group в каталог 2 используя абсолютное имя копируемого файла и относительное имя каталога назначения, задавая ему новое имя ФИО_group.

4. Скопируйте файл /etc/group в каталог 3 используя относительные имена копируемого файла и каталога назначения.

5. Скопируйте файл /etc/group в каталог 4 используя абсолютные имена копируемого файла и относительное имя каталога назначения с использованием специального символа ~.

6. При помощи одной команды зайдите в каталог 3.

7. Удалите файл group из каталога 4 при помощи одной команды.

8. Переименуйте файл group в ФИО_group и создайте новый файл group, используя команду touch.

9. Перейдите в свой домашний каталог. Удалите каталоги 1 и 4.

10. Выведите первые и последние 5 строк файла /etc/group.

Отчет

В этой и последующих работах отчет по проделанной работе представляется преподавателю в стандартной форме: на листах формата А4, с титульным листом (включающим тему, фио, номер зачетки и пр.), целью, ходом работы и выводами по выполненной работе. Каждое задание должно быть отражено в отчете следующим образом: 1) что надо было сделать, 2) как это сделали, 3) что получилось, и, в зависимости от задания – 4) почему получилось именно так, а не иначе.

ЛАБОРАТОРНАЯ РАБОТА №2. «ССЫЛКА. ПРАВА ДОСТУПА»

Часть 1. Ссылки на файлы и каталоги

Виды файлов

С точки зрения пользователя в системе UNIX существует два типа объектов: файлы и процессы. Все данные хранятся в виде файлов. Работа с различными ресурсами организована через файлы. в системе существует 6 различных типов:

- обычный файл;
- каталог - это файл с именами находящихся в нем файлов и их индексных дескрипторов (inode), любой процесс может прочитать каталог при наличии прав, а записать в него может только ядро;
- специализированный файл устройства необходим для доступа к физическому устройству. Устройства делятся на символьные и блочные;
- именованный канал служит для связи между процессами;
- символическая связь (ссылка) - это еще одно обозначение путевого имени файла;
- сокет служит для обмена между процессами.

При выполнении команды ls с ключом -l выдается полная информация по файлам каталога. Первый символ в строке для любого файла определяет его тип. Команда dd предназначена для чтения данных из файла.

Жесткие и символические ссылки.

Индексный дескриптор хранит информацию о файле (атрибуты, число жестких связей, идентификаторы владельца и групп,размер файла и т.п.), кроме его имени. Между именем и inode устанавливается жесткая связь, число этих связей может быть более одной. Все жесткие связи равноправны. Изменение атрибутов или данных по одному имени автоматически распространяется на все. при удалении имени удаляется только связь, если связей больше нет удаляется весь файл.

Если необходимо создать ссылку на файл в каталоге другого пользователя, жесткая ссылка может не сработать. Это обусловлено тем, что файловую структуру ОС Linux можно физически сегментировать на файловые системы. Файловая система может располагаться на любых физических запоминающих устройствах - от дискеты до жестких дисков. Несмотря на то что файлы и каталоги во всех файловых системах присоединены к одному общему дереву каталогов, каждая файловая система физически управляет своими файлами и каталогами. Это значит, что Файл одной файловой системы нельзя связать прямой ссылкой с файлом, принадлежащим другой файловой системе.

Для решения этой проблемы применяются символические ссылки. Символическая (косвенная) ссылка содержит путевое имя файла, для которого она создается. Чтобы удалить файл, нужно удалить только прямые ссылки Если остались символические ссылки, доступ к файлу по ним будет невозможен.

При создании сиволической ссылки создается файл, имеющий тип "связь" и свой inode. В отличие от прямых ссылок, символические ссылки можно использовать для создания ссылок на каталоги. По сути дела, можно создать еще одно имя для обращения к каталогу. При этом следует помнить, что команда pwd всегда выдает фактическое имя каталога, а не символическое.

Ссылки. Команда ln

С помощью команды ln файлам можно присваивать дополнительные имена. Это нужно для того, чтобы иметь возможность обращаться к файлу по разным именам из разных каталогов. Дополнительные имена часто называют ссылками.

Команда ln использует два аргумента: имя исходного файла и новое, дополнительное имя файла. В операции ls указываются оба имени, но в действительности существует лишь один физический файл.

$ ln исходное_имя_файла дополнительное_имя_файла

В следующем примере файлу today присваивается дополнительное имя weather.

```
$ ls
today
$ ln today weather
$ ls
today weather
```

Файлу можно дать несколько имен, применив по отношению к нему несколько команд ln. В следующем примере файлу today присваиваются дополнительные имена weather и weekend.

```
$ ln today weather
$ ln today weekend
$ ls
today weather weekend
```

Используя команду ls с опцией -l, можно выяснить, есть ли у файла ссылки. Эта команда позволяет получить следующую информацию: права доступа, количество ссылок, размер файла и дату последнего изменения. Первое число перед именем владельца файла - это количество ссылок. Число перед датой - размер файла. В следующем примере пользователь получает полную информацию о файлах today и

weather. Обратите внимание: число ссылок у обоих файлов равно двум. Более того, совпадают их размеры и даты создания. Это еще раз показывает, что указанные файлы - просто разные имена одного и того же файла.

```
$ ls -l today weather
-rw-rw-r-- 2 chris group 563 Feb 14 10:30 today
-rw-rw-r-- 2 chris group 563 Feb 14 10:30 weather
```

Данные сведения, однако, не позволяют утверждать наверняка, что имена этих файлов связаны ссылками. Вы можете просто посмотреть, совпадают ли число ссылок, размеры и даты модификации двух файлов, как в случае с файлами today и weather. Для того чтобы знать это наверняка, нужно дать команду ls с опцией -i. Эта команда

сообщает имя файла и его индексный дескриптор. Индексный дескриптор - это уникальный номер, которым система обозначает конкретный файл. Если индексные дескрипторы двух имен файлов совпадают, это значит, что они относятся к одному и тому же файлу. В следующем примере пользователь получает информацию о файлах today, weather и larisa. Обратите внимание: today и weather имеют один и тот же индексный дескриптор.

```
$ ls -i today weather larlsa
1234 today 1234 weather 3976 larisa
```

Дополнительные имена, или ссылки, созданные командой ln, часто используются для обращения к одному файлу из разных каталогов. Файл, находящийся в одном каталоге, может быть связан ссылкой с другим каталогом и использоваться из него. Предположим, нужно обратиться к файлу, находящемуся в начальном каталоге, из другого каталога. Для этого нужно создать ссылку из этого каталога на файл, находящийся в начальном каталоге. Эта ссылка будет просто еще одним именем файла. Поскольку ссылка находится в другом каталоге, она может иметь то же имя, что и оригинал. Для того чтобы создать ссылку на файл, находящийся в начальном каталоге, из другого каталога, нужно в качестве второго аргумента команды ln задать имя этого каталога.

```
$ ln имя_файла имя_каталога
```

Аналогично тому как это делалось при использовании команд cp и mv, ссылке можно дать другое имя. Для этого новое имя нужно указать через косую черту после имени каталога. В следующем примере в каталоге reports создается ссылка на файл today с именем wednesday. Фактически существует один файл, оригинал с именем today в каталоге chris, но теперь файл today связан с каталогом reports ссылкой wednesday. В этом смысле данный файл получил новое имя. В каталоге reports файл today проходит под именем wednesday.

```
$ ln today reports/wednesday
$ ls
today reports
$ ls reports
wednesday
```

Создавать ссылки на файлы можно с помощью путевых имен. В следующем примере для файла monday в каталоге reports создается ссылка в каталоге chris. Второй аргумент команды здесь - абсолютное путевое имя.

```
$ ln monday /home/chris
```

Для того чтобы удалить файл, нужно удалить все его ссылки. Имя файла фактически рассматривается как ссылка на этот файл. То есть с помощью команды rm удаляется именно ссылка на тот или иной файл. Если ссылок было несколько и одна из них удаляется, к файлу можно обращаться по оставшимся даже в случае удаления исходной ссылки - первоначального имени файла. В следующем примере файл today удаляется командой rm, но остается ссылка на этот файл с именем weather. К файлу можно обращаться по этому имени.

```
$ ln today weather
$ rm today
$ cat weather
The storm broke today
and the sun came out.
```

ОС Linux поддерживает так называемые символические ссылки. Ссылки, которые мы рассматривали до сих пор, называются прямыми ссылками (или жесткими ссылками - hard link). В принципе, в большинстве случаев удобно использовать именно прямые ссылки, но им присущ один серьезный недостаток: если вы попытаетесь создать ссылку на файл в каталоге другого пользователя, прямая ссылка может не сработать. Это обусловлено тем, что файловую структуру ОС Linux можно физически сегментировать на файловые системы. Файловая система может располагаться на любых физических запоминающих устройствах - от дискеты до комплекта жестких дисков. Несмотря на то, что файлы и каталоги во всех файловых системах присоединены к одному общему дереву каталогов, каждая файловая система физически управляет своими файлами и каталогами. Это значит, что файл одной файловой системы нельзя связать прямой ссылкой с файлом, принадлежащим другой файловой системе. Если вы попробуете создать ссылку на файл, находящийся в каталоге другого пользователя и принадлежащий другой файловой системе, прямая ссылка не сработает.

Для решения этой проблемы применяются символические ссылки. Символическая (или косвенная) ссылка содержит путевое имя файла, для которого она создается. Это не прямая ссылка, а скорее информация о том, как найти конкретный файл. Вместо того чтобы

регистрировать еще одно имя файла, как это делает прямая ссылка, символическая ссылка позволяет создать еще одно обозначение путевого имени файла.

Символические ссылки создаются командой ln с опцией -s. В следующем примере пользователь создает ссылку lunch на файл /home/george/veglist.

```
$ ln -s /home/george/veglist lunch
```

Нетрудно убедиться в различии между символической ссылкой и файлом, на который она указывает. В приведенном ниже примере пользователь получает полную информацию о lunch и /home/george/veglist с помощью команды ls -l. Первый символ в строке обозначает тип файла. Символические ссылки имеют собственный тип, обозначенный как l. Тип файла для lunch - l, т.е. это символическая ссылка, а не обычный файл. Число, стоящее после group - это размер файла. Обратите внимание: размеры разные. Размер файла lunch составляет всего 20 байтов. Это обусловлено тем, что lunch - всего лишь символическая ссылка, хранящая путевое имя реально существующего файла, занимающее всего несколько байтов. Это не прямая ссылка на файл veglist.

```
$ ls -I /home/george/vegllst lunch
lrw-rw-r-- 1 chris group 20 Feb 14 10:30 lunch-> /home/george/veglist
-rw-rw-r-- 1 george group 793 Feb 14 10:30 veglist
```

Для того чтобы удалить файл, нужно удалить только прямые ссылки. Если остались символические ссылки, доступ к файлу по ним будет невозможен. В данном случае в символической ссылке содержится путевое имя более не существующего файла.

В отличие от прямых ссылок, символические ссылки можно использовать для создания ссылок на каталоги. По сути дела, можно создать еще одно имя для обращения к каталогу. При этом следует помнить, что команда pwd всегда выдает фактическое имя каталога, а не символическое. В следующем примере пользователь создает для каталога thankyou символическую ссылку gifts. Используя ссылку gifts в команде cd, пользователь переходит в каталог thankyou.

Часть 2. Права доступа к файлам и каталогам

Категории пользователей и действия над файлами

Для каждого файла и каталога в ОС Linux задаются права доступа, которые определяют, кто и какие операции может проводить над данным файлом. Существуют три категории пользователей, которые могут иметь доступ к файлу или каталогу: "владелец", "группа" и "прочие". Владелец - это пользователь, создавший файл. Часто пользователи объединяются в группы Например, системный администратор может объединить в группу пользователей, работающих над одним проектом. Пользователи не входящие в группу относятся к категории "прочие".

Для каждой категории пользователей существует отдельный набор прав доступа на чтение, запись и выполнение. Первый набор управляет доступом самого пользователя к его файлам. Второй набор управляет доступом группы к файлам этого пользователя. Третий набор управляет доступом прочих пользователей к файлам данного пользователя. Эти три набора прав доступа на чтение запись и выполнение для трех категорий - владельца, группы и прочих -образуют в совокупности девять типов разрешений на действия с файлом.

Команда ls с опцией -l выдает подробную информацию о файле, включая права доступа к нему.

Отсутствие права доступа обозначается дефисом, -. Право на чтение обозначается буквой r, право на запись - буквой w, право на выполнение - буквой x. Первый трехсимвольный набор - это права доступа к файлу для категории "владелец". Второй трехсимвольный набор -

это права доступа к файлу для категории "группа". Третий трехсимвольный набор - это права доступа к файлу для категории "прочие".

Для директорий параметры те же, но обозначают немного другое: просмотр директории (r), создание папок / файлов (w) внутри директории, переход в директорию (x).

Для создания различных конфигураций прав доступа применяется команда chmod. В качестве аргументов в этой команде используются два списка: список изменений прав доступа и список имен файлов. Список изменений прав доступа можно задавать двумя способами. В первом, который называется символическим методом, используются символы r, w, x. Во втором, который называют абсолютным методом, применяется так называемая двоичная маска.

Установление прав доступа: символы прав доступа

В символическом методе права доступа на чтение, запись и выполнение обозначаются соответственно символами r, w и x. Любое из этих разрешений можно добавлять и удалять. Символом добавления права доступа является знак плюс, +. Символом отмены является знак минус, -. Категории пользователей "владелец", "группа" и "прочие" обозначаются соответственно символами u, g и o. Символ категории ставится перед символами, устанавливающими права на чтение, запись и выполнение. Если символа категории нет, то подразумеваются все категории и указанные права устанавливаются для пользователя, группы и прочих. Есть еще один символ разрешения, a (от all), который обозначает все категории. Он действует по умолчанию.

Помимо прав на чтение, запись и выполнение можно устанавливать для пользователей право владения программами на время их выполнения. Как правило, программа во время выполнения обладает правами того пользователя, который ее запустил, даже если сам файл программы принадлежит другому пользователю. Установка бита "смены идентификатора пользователя" позволяет остальным пользователям выполнять программу с правами ее настоящего владельца. Например, многими программами в системе владеет пользователь root, тогда как выполняют их обычные пользователи. Иногда при работе таких программ возникает необходимость изменения файлов, принадлежащих пользователю root. В этом случае обычному пользователю нужно запустить эту программу с сохранением права пользователя root, чтобы она получила право изменять принадлежащие ему файлы. Бит "смены идентификатора группы" имеет то же значение, что и бит "смены идентификатора пользователя", но только для групп. Пользователи выполняют программу с правами той группы, к которой принадлежит ее владелец. Такая программа может изменять файлы, принадлежащие группе.

Для установки бита смены идентификатора пользователя или группы используется опция s. Бит смены идентификатора пользователя или группы обозначается буквой s в позиции "выполнение" категории "владелец" или "группа". Это право фактически является вариантом права выполнения, x.

Есть еще одно специальное право доступа, которое повышает эффективность программ. Так называемый sticky-бит дает системе указание оставить программу в памяти после завершения ее выполнения. Это полезно для небольших программ, которые часто используются многими пользователями. На наличие sticky-бита указывает буква t в позиции "выполнение" категории "прочие". У программы с правом доступа на чтение и выполнение и установленным sticky-битом права доступа обозначаются как r-t.

Установление прав доступа: двоичные маски

Вместо символов разрешений многие пользователи предпочитают применять абсолютный метод. Абсолютный метод позволяет изменять сразу все права доступа. Здесь используется двоичная маска, которая обозначает все разрешения в каждой категории. Эти три категории, по три разрешения в каждой, представлены в восьмеричном формате. В восьмеричной системе счисления все числа имеют основание 8. При преобразовании в

двоичный формат каждый восьмеричный разряд превращается в три двоичных. Три восьмеричных разряда числа преобразуются в три набора по три двоичных разряда в каждом. Итого получается девять цифр, что в точности соответствует количеству разрешений доступа к файлу.

Восьмеричные цифры можно использовать как маску для установления различных прав доступа к файлу. Каждая восьмеричная цифра относится к одной из категорий пользователей. При этом категории нумеруются слева направо, начиная с категории "владелец". Первая восьмеричная цифра относится к владельцу, вторая к группе, а третья - к прочим пользователям.

Чтобы обозначить бит смены иднтификатора пользователя и sticky-бит нужно ввести перед этими восьмеричными цифрами еще одну. Бит смены идентификатора пользователя обозначается цифрой 4 (100); бит смены идентификатора группы обозначается цифрой 2 (010); sticky-бит обозначается цифрой 1 (001).

Права доступа к каталогам

Права доступа можно устанавливать и для каталогов. Если для каталога установлено право на чтение, пользователи могут получать список файлов, находящихся в данном каталоге. Право на выполнение позволяет переходить в этот каталог. Право на запись дает возможность пользователю создавать и удалять файлы в этом каталоге. При создании каталога для него автоматически устанавливаются права на чтение, запись и выполнение для владельца. Это позволяет получать список файлов, содержащихся в каталоге, переходить в него и создавать в нем файлы.

Как и в случае с файлами, права доступа к каталогам задаются для владельца, группы и прочих пользователей. Также при установлении прав доступа к каталогу можно пользоваться символами прав доступа и двоичными масками.

Команда ls с опцией -ld выдает полную информацию только о каталоге. Если у пользователя есть файлы, доступ к которым он хотел бы предоставить другим пользователям, то нужно установить права доступа не только для этих файлов, но и для каталога, в котором они находятся. Другой пользователь, желающий получить доступ к файлу, должен сначала войти в каталог, где этот файл находится. Это же касается и родительских каталогов.

Даже если к каталогу имеют право доступа другие пользователи, они смогут попасть в него лишь в том случае, если имеют право доступа и к родительскому каталогу данного каталога. Поэтому необходимо внимательно следить за деревом каталогов. Чтобы пользователь мог работать в каталоге, ему должны быть доступны все остальные каталоги, стоящие в дереве выше этого каталога.

Изменение владельца и группы: команды chown и chgrp Доступ к файлу могут иметь все пользователи, но права доступа к нему может менять только владелец. Если же необходимо передать контроль над правами доступа к файлу другому пользователю, надо заменить владельца файла. Контроль над файлом передается другому пользователю с помощью команды chown. В качестве первого аргумента в этой команде указывается имя пользователя, которому передается контроль. После него можно дать список файлов, которые вы передаете.

С помощью команды chgrp можно изменить группу, владеющую файлом. В качестве первого аргумента эта команда принимает имя новой группы для файла или файлов. После имени группы можно дать список файлов, которые передаются в эту группу.

Задание для выполнения

Часть 1.

1. Изучить назначение и ключи команды ln.

- создать жесткую ссылку на файл. Просмотреть содержимое файла, используя ссылку. Удалить файл. Просмотреть содержимое файла. Объяснить результат;

- создать жесткую ссылку на каталог. Объяснить результат;

2. Выполнить все задания пункта 1, создавая не жесткие, а символьные ссылки.

3. Создать жесткую и символьную ссылки на файл. С помощью команды ls просмотреть inod файла и ссылок. Объяснить результат.

Часть 2.

1. Изучите при помощи man опцию -l команды ls. Просмотрите права каталогов /etc, /bin и домашнего каталога. Просмотрите права файлов, содержащиеся в этих каталогов. Выявите тенденции (файлов с какими правами в каких каталогах больше). Сделайте вывод.

2. Изучите материал, посвященный пользователям и группам пользователей. Изучите руководство по командам chown и chgrp. Приведите пример изменения владельца на class и группы на room для файла fail (не выполнять в терминале!). Выясните, кто является владельцем и к какой группе владельцов принадлежат каталоги и файлы домашнего каталога, каталогов /etc, /root, /bin и /dev.

3. Определите атрибуты файлов /etc/shadow и /etc/passwd, попробуйте вывести на экран содержимое этих файлов. Объясните результат.

4. Изучите команду chmod. Создайте в домашнем каталоге любые четыре файла. Какие права устанавливаются на новые файлы? Установите при помощи восмеричных масок на каждый из них в отдельности следующие права:

- для себя чтение, для группы и остальных - никаких;
- для себя все, для группы чтение и выполнение, для остальных запись;
- для себя чтение и запись, для группы никаких, для остальных выполнение;
- для себя запись, для группы все, для остальных чтение.

5. Выполните задание предыдущего пункта, используя в команде chmod только символы прав доступа.

6. Переведите номер своей зачетной книжки в восьмеричную систему счисления, разбейте полученное значение на группы по 3 цифры и создайте файлы с правами доступа, выраженными полученными масками. Сопоставьте данные маски с символами прав доступа и объясните, какие операции с данными файлами доступны каким субъектам системы.

7. В домашнем каталоге создайте каталог, в нем создайте два файла. Установите на каталог права так, чтобы его можно было только читать. Выведите содержимое каталога. Создайте в каталоге еще один файл. Выведите содержимое каталога. Зайдите в каталог. Объясните результат .

8. Скопируйте в свой домашний каталог файл ls из каталога /bin. Запретите выполнение этого файла и попробуйте выполнить именно его, а не исходный(!). Объясните результат.

ЛАБОРАТОРНАЯ РАБОТА №3. «BASH: ПОТОКИ ДАННЫХ. ПРОГРАММИРОВАНИЕ»

Часть 1. Потоки ввода и вывода данных

Потоки и файлы

Логически все файлы в системе Linux организованы в непрерывный поток байтов. Любой файл можно свободно копировать и добавлять к другому файлу, так как все файлы организованы одинаково.

Эта логическая организация файлов распространяется на операции ввода и вывода. Данные в операциях ввода и вывода организованы аналогично файлам. Данные, вводимые с клавиатуры, направляются в поток данных, организованный как непрерывная совокупность байтов. Данные, выводимые из команды или программы, также направляются в поток, организованный как непрерывная совокупность байтов. Входной поток данных в ОС Linux называется стандартным вводом, а выходной поток данных - стандартным выводом.

Поскольку стандартный ввод и стандартный вывод организованы так же, как файл, они свободно могут взаимодействовать с файлами. В ОС Linux широко используется переадресация, которая позволяет перемещать данные в файлы и из файлов.

Переадресация стандартного вывода: > и >>

Когда выполняется команда ОС Linux, дающая какую-либо выходную информацию, эта информация направляется в поток данных стандартного вывода. По умолчанию в качестве пункта назначения данных стандартного вывода используется какое-либо устройство, в данном случае экран. Устройства, такие как клавиатура и экран, тоже рассматриваются как файлы. Они принимают и отправляют потоки байтов, имеющих такую же организацию, как и файлы байтового потока. Экран - это устройство, на котором отображается непрерывный поток байтов. По умолчанию стандартный вывод посылает свои данные на экран, где они отображаются.

Для направления стандартного вывода в файл, а не на экран, необходимо использовать оператор переадресации вывода: >. С помощью операции переадресации создается новый файл-адресат. Если он уже существует, то система заменит его содержимое данными стандартного вывода. Для того чтобы этого не произошло, можно установить для shell режим noclobber:

```
set -o noclobber
```

В этом случае операция переадресации существующего файла выполнена не будет. Отменить режим noclobber можно, поставив после оператора переадресации восклицательный знак:

```
cat file1 >! file2
```

Хотя оператор переадресации и имя файла ставятся после команды, перенаправление стандартного потока выполняется не после выполнения команды, а до него. С помощью оператора переадресации создается файл и переадресация организуется до того, как начинают поступать данные со стандартного вывода. Если файл уже существует, он будет разрушен и заменен новым файлом под тем же именем. Команда, генерирующая выходные данные, выполняется только после создания файла переадресации.

Если пользователь попытается использовать одно и то же имя для входного файла команды и переадресованного файла, возникнет ошибка. Так как операция переадресации выполняется первой, входной файл, поскольку он существует, разрушается и заменяется файлом с тем же именем. Когда команда начинает выполняться, она обнаруживает пустой

входной файл. Для добавления стандартного вывода к существующему файлу служит оператор переадресации >>.

Переадресация стандартного ввода: < и <<

Многие команды ОС Linux могут принимать данные со стандартного ввода. Сам стандартный ввод получает данные из устройства или из файла. По умолчанию в качестве устройства для стандартного ввода используется клавиатура. Символы, набираемые на клавиатуре, подаются на стандартный ввод, который затем направляется в команду.

Во многих Linux-системах применяется метод буферизации строк. При буферизации строк, информация посылается на стандартный ввод только после того, как пользователь ввел всю строку.

Стандартный ввод можно переадресовать так же, как и стандартный вывод. Стандартный ввод может приниматься не с клавиатуры, а из файла. Оператор переадресации стандартного ввода: <. Кроме этого для переадресации стандартного ввода применяется механизм "файл здесь":

```
cat << 'слово-признак конца ввода'
> 'текст'
> 'текст'
> 'слово-признак конца ввода'
```

Операции переадресации стандартного ввода и стандартного вывода можно объединять.

Переадресация и пересылка по каналу стандартного потока ошибок: >&, 2>.

При выполнении команд иногда происходят ошибки. Например, пользователь указал неверное количество аргументов или возникла какая-то системная ошибка. Когда возникает ошибка, система выдает специальное сообщение. Как правило, такие сообщения об ошибках отображаются на экране вместе со стандартным выводом. ОС Linux, однако, различает стандартный вывод и сообщения об ошибках. Сообщения об ошибках помещаются еще в один стандартный байтовый поток, который называется стандартным потоком ошибок (диагностики).

Так как сообщения об ошибках направляются в поток, отдельный от стандартного вывода, то в случае переадресации стандартного вывода в файл они все равно появляются на экране.

Стандартный поток ошибок можно переадресовать так же, как и стандартный вывод. Например, сообщения об ошибках можно сохранить в файле для справок. Как и в случае стандартного вывода, пунктом назначения стандартного потока ошибок по умолчанию является экран, однако с помощью специальных операторов переадресации его можно переадресовать в любой файл или устройство.

Для переадресации стандартного потока ошибок в shell предусмотрена специальная возможность. Все стандартные байтовые потоки в операциях переадресации можно обозначать номерами (дескрипторами). Номера 0, 1 и 2 обозначают соответственно стандартный ввод, стандартный вывод и стандартный поток ошибок. Оператор переадресации вывода, >, по умолчанию действует на стандартный вывод, 1. Чтобы переадресовать стандартный поток ошибок, нужно поставить перед оператором переадресации вывода цифру 2.

Стандартный поток ошибок можно дописать в файл, используя цифру 2 и оператор добавления: >>. Для того чтобы переадресовать и стандартный вывод, и стандартный поток ошибок, нужны две операции переадресации и два файла.

В BASH можно ссылаться на стандартный поток по его номеру со знаком "&": &1 обозначает стандартный вывод. Это обозначение можно использовать в операции

переадресации для того, чтобы сделать стандартный вывод файлом назначения. Операция переадресации 2>&1 переадресует стандартный поток ошибок на стандартный вывод. В результате стандартный вывод становится файлом назначения для стандартного потока ошибок. Операция переадресации 1>&2 переадресует стандартный ввод в стандартный поток ошибок. По умолчанию входным потоком операции >& является стандартный поток ошибок, а выходным потоком - стандартный вывод. Поэтому, если его использовать в команде, все сообщения о ошибках будут перенаправляться на стандартный вывод.

Программные каналы: |

Иногда возникают ситуации, когда нужно передать данные из одной команды в другую. Другими словами, необходимо послать стандартный вывод одной команды на стандартный ввод другой, а не в файл. Для образования такого соединения в ОС Linux используется так называемый канал. Оператор канала, | (вертикальная черта), помещенный между двумя командами, связывает их стандартные потоки. Стандартный вывод одной команды становится стандартным вводом другой. Выходная информация команды, стоящей перед оператором канала, передается в качестве входной в команду, стоящую за оператором канала.

Программные каналы можно объединять с другими средствами shell, например со специальными символами, проводя таким образом специализированные операции.

Если операция переадресации позволяет просто направлять выходную информацию в файл, то каналы обеспечивают ее пересылку в другую команду Linux. Следует помнить о различии между файлом и командой. Файл - это носитель данных. Вы можете хранить или читать из него данные. Команда - это программа, которая исполняет инструкции. Команда может читать данные из файла и сохранять данные в файле, но во время выполнения ее нельзя рассматривать как файл. По этой причине операция переадресации выполняется с файлами, а не с командами. В процессе переадресации данные посылаются из программы в файл, а не в другую программу. Пунктом назначения операции переадресации могут быть только файлы, но не программы.

Можно, тем не менее, смоделировать процесс конвейерной пересылки с помощью нескольких операций переадресации. Выходная информация одной команды посылается в файл. Команда, записанная в следующей строке, использует этот файл как переадресованный ввод.

Каналы работают со стандартным выводом команды независимо от того, что подается на этот вывод. Пересылаться по каналу из одной команды в другую может содержимое целого файла и даже нескольких файлов.

Стандартным вводом, посылаемым по каналу в команду, можно более эффективно управлять с помощью аргумента стандартного ввода, −. Дефис, используемый в команде в качестве аргумента, обозначает стандартный ввод.

Допустим, нужно напечатать файл с именем его каталога в начале списка. Команда pwd выдает имя каталога, а команда cat выдает содержимое файла. В данном случае команде cat нужно использовать в качестве входной информации и файл, и стандартный ввод, пересланный по каналу из команды pwd. Команда cat будет иметь два аргумента: стандартный ввод, обозначенный дефисом, и имя выводимого на печать файла.

```
pwd | cat - file | lpr
```

Каналы и переадресация: команда tee

Для того, чтобы переадресовать стандартный вывод в файл и одновременно воспроизвести эту информацию на экране необходимо использовать команду tee. Команда tee копирует стандартный вывод в файл. В качестве аргумента она использует имя нового файла, в который копируется стандартный вывод. Это все равно что скопировать содержимое

стандартного вывода и один его экземпляр переадресовать в файл, а другой отправить дальше (часто - на экран).

Переадресацией в сочетании с каналами необходимо пользоваться осторожно. Переадресация стандартного вывода задает файл назначения для него. Стандартный вывод записывается и сохраняется в этом файле. После записи никакой информации для пересылки по каналу в другую команду не остается. Переадресация может производиться в конце последовательности программных каналов, но не внутри этой последовательности.

Задание для выполнения

1. Вывести любое сообщение с помощью команды echo перенаправив вывод:
- в несуществующий файл с помощью символа >;
- в несуществующий файл с помощью символа >>;
- в существующий файл с помощью символа >;
- в существующий файл с помощью символа >>;

Объяснить результаты.

2. Переадресовать стандартный ввод для команды cat на файл с применением механизм "файл здесь".

3. Вывести сообщение с помощью команды echo в канал ошибок.

Создать файл myscript:

```
#!/bin/sh
echo stdout
echo stderr>&2
exit 0
```

Запустить его (одно задание- одна команда):
- без перенаправления (sh myscript);
- перенаправив стандартный вывод в файл, просмотреть содержимое файла (sh myscript > file1);
- перенаправить стандартный канал ошибок в существующий и несуществующий файлы с помощью символов > и >> ;
- перенаправив стандартный вывод в файл 1, стандартный канал ошибок - в файл 2;
- перенаправив стандартный вывод и стандартный канал ошибок в файл 3;
- перенаправив стандартный вывод в файл 4 с помощью символа >, а стандартный канал ошибок в файл 4 с помощью символа >>;

Объяснить результаты.

4. Вывести 2 и 7-10 строку из последних пятнадцати строк отсортированного в обратном порядке файла /etc/group, используя команду sed.

5. Подсчитать при помощи конвейера команд количество файлов и каталогов в системном каталоге /etc, используя команду grep.

6. Написать скрипт, выводящий на консоль все аргументы командной строки, переданные данному скрипту. Привести **различные варианты запуска** данного скрипта, в том числе без непосредственного вызова интерпретатора в командной строке.

ЛАБОРАТОРНАЯ РАБОТА №4 «GCC. ПРОЦЕССЫ»

Знакомство с компилятором GCC

Средствами, традиционно используемыми для создания программ для открытых операционных систем, являются инструменты разработчика GNU. Сделаем маленькую историческую справку. Проект GNU был основан в 1984 году Ричардом Столлманом. Его необходимость была вызвана тем, что в то время сотрудничество между программистами было затруднено, так как владельцы коммерческого программного обеспечения чинили многочисленные препятствия такому сотрудничеству. Целью проекта GNU было создание комплекта программного обеспечения под единой лицензией, которая не допускала бы возможности присваивания кем– то эксклюзивных прав на это ПО. Частью этого комплекта и является набор инструментов для разработчика, которым мы будем пользоваться, и который должен входить во все дистрибутивы Linux.

Одним из этих инструментов является компилятор GCC. Первоначально эта аббревиатура расшифровывалась, как GNU C Compiler. Сейчас она означает – GNU Compiler Collection.

Файлы с исходными кодами программ, которые мы будем создавать, это обычные текстовые файлы, и создавать их можно с помощью любого текстового редактора (например, GEdit KWrite, Kate, а также более традиционные для пользователей Linux – vi и emacs).

Помимо текстовых редакторов, существуют специализированные среды разработки со своими встроенными редакторами. Одним из таких средств является KDevelop. Интересно, что в нём есть встроенный редактор и встроенная консоль, расположенная прямо под редактором. Так что можно прямо в одной программе, не переключаясь между окнами, и редактировать код и давать консольные команды.

Создайте отдельный каталог hello. Это будет каталог нашего первого проекта. В нём создайте текстовый файл hello.c со следующим текстом:

```
#include <stdio.h>
int main(void){
  printf("Hello world!\n");
  return(0);
}
```

Затем в консоли зайдите в каталог проекта. Наберите команду:

```
gcc hello.c
```

Теперь посмотрите внимательно, что произошло. В каталоге появился новый файл a.out. Это и есть исполняемый файл. Запустим его. Наберите в консоли:

```
./a.out
```

Программа должна запуститься, то есть должен появиться текст:

```
Hello world!
```

Компилятор gcc по умолчанию присваивает всем созданным исполняемым файлам имя a.out. Если хотите назвать его по– другому, нужно к команде на компиляцию добавить флаг – o и имя, которым вы хотите его назвать. Давайте наберём такую команду:

```
gcc hello.c -o hello
```

Мы видим, что в каталоге появился исполняемый файл с названием hello. Запустим его.

```
./hello
```

Как видите, получился точно такой же исполняемый файл, только с удобным для нас названием.

Флаг -o является лишь одним из многочисленных флагов компилятора gcc. Некоторые другие флаги мы рассмотрим позднее. Чтобы просмотреть все возможные флаги, можно воспользоваться справочной системой man. Наберите в командной строке:

```
man gcc
```

Перед вами предстанет справочная система по этой программе. Просмотрите, что означает каждый флаг. С некоторыми из них мы скоро встретимся. Выход из справочной системы осуществляется с помощью клавиши q.

Вы, конечно, обратили внимание, что, когда мы запускаем программу из нашего каталога разработки, мы перед названием файла набираем точку и слэш. Зачем же мы это делаем?

Дело в том, что, если мы наберём только название исполняемого файла, операционная система будет искать его в каталогах /usr/bin и /usr/local/bin, и, естественно, не найдёт. Каталоги /usr/bin и /usr/local/bin – системные каталоги размещения исполняемых программ. Первый из них предназначен для размещения стабильных версий программ, как правило, входящих в дистрибутив Linux. Второй – для программ, устанавливаемых самим пользователем (за стабильность которых никто не ручается). Такая система нужна, чтобы отделить их друг от друга. По умолчанию при сборке программы устанавливаются в каталог /usr/local/bin. Крайне нежелательно помещать что-либо лишнее в /usr/bin или удалять что-то оттуда вручную, потому что это может привести к краху системы. Там должны размещаться программы, за стабильность которых отвечают разработчики дистрибутива.

Чтобы запустить программу, находящуюся в другом месте, надо прописать полный путь к ней, например так:

```
/home/myuser/projects/hello/hello
```

Или другой вариант: прописать путь относительно текущего каталога, в котором вы в данной момент находитесь в консоли. При этом одна точка означает текущий каталог, две точки – родительский. Например, команда ./hello запускает программу hello, находящуюся в текущем каталоге, команда ../hello – программу hello, находящуюся в родительском каталоге, команда ./projects/hello/hello – программу во вложенных каталогах, находящихся внутри текущего.

Есть возможность добавлять в список системных путей к программам дополнительные каталоги. Для этого надо добавить новый путь в системную переменную PATH. Но давайте пока не будем отвлекаться от главной темы. Переменные окружения – это отдельный разговор.

Теперь рассмотрим, что же делает программа gcc. Её работа включает три этапа: обработка препроцессором, компиляция и компоновка (или линковка).

Препроцессор включает в основной файл содержимое всех заголовочных файлов, указанных в директивах #include. В заголовочных файлах обычно находятся объявления функций, используемых в программе, но не определённых в тексте программы. Их определения находятся где-то в другом месте: или в других файлах с исходным кодом или в бинарных библиотеках *(ключ – E)*.

Вторая стадия – компиляция. Она заключается в превращении текста программы на языке C/C++ в набор машинных команд. Результат сохраняется в объектном файле. Разумеется, на машинах с разной архитектурой процессора двоичные файлы получаются в разных форматах, и на одной машине невозможно запустить бинарный файл собранный на другой машине (разве только, если у них одинаковая архитектура процессора и одинаковые операционные системы). Вот почему программы для UNIX– подобных систем распространяются в виде исходных кодов: они должны быть доступны всем пользователям, независимо от того, у кого какой процессор и какая операционная система *(ключ – c)*.

Последняя стадия – компоновка. Она заключается в связывании всех объектных файлов проекта в один, связывании вызовов функций с их определениями, и присоединением библиотечных файлов, содержащих функции, которые вызываются, но не определены в проекте. В результате формируется запускаемый файл – наша конечная цель. Если какая– то функция в программе используется, но компоновщик не найдёт место, где эта функция определена, он выдаст сообщение об ошибке, и откажется создавать исполняемый файл *(ключ – o)*.

Если мы создаём объектный файл из исходного файла, уже обработанного препроцессором (например, такого, какой мы получили выше), то мы должны обязательно указать явно, что компилируемый файл является файлом исходного кода, обработанный препроцессором, и имеющий теги препроцессора. В противном случае он будет обрабатываться, как обычный файл C++, без учёта тегов препроцессора, а, значит, связь с объявленными функциями не будет устанавливаться. Для явного указания на язык и формат обрабатываемого файла служит опция -x. Файл C++, обработанный препроцессором обозначается cpp– output.

```
gcc –x cpp–output –c proga.cpp
```

Наконец, последний этап –компоновка. Получаем из объектного файла исполняемый.

```
gcc proga.o –o proga
```

Можно его запускать.

```
./proga
```

Вы спросите: «Зачем вся эта возня с промежуточными этапами? Не лучше ли просто один раз скомандовать gcc proga.c -o proga?»

Дело в том, что настоящие программы очень редко состоят из одного файла. Как правило, исходных файлов несколько, и они объединены в проект. И в некоторых исключительных случаях программу приходится компоновать из нескольких частей, написанных на разных языка. В этом случае приходится запускать компиляторы разных языков, чтобы каждый получил объектный файл из своего исходника, а затем уже эти полученные объектные файлы компоновать в исполняемую программу.

Процессы

Контекст процесса

Контекст процесса складывается из пользовательского контекста и контекста ядра, как изображено на рисунке.

Под пользовательским контекстом процесса понимают код и данные, расположенные в адресном пространстве процесса. Все данные подразделяются на:

• инициализируемые неизменяемые данные (например, константы);

• инициализируемые изменяемые данные (все переменные, начальные значения которых присваиваются на этапе компиляции);

• неинициализируемые изменяемые данные (все статические переменные, которым не присвоены начальные значения на этапе компиляции);

• стек пользователя;

• данные, расположенные в динамически выделяемой памяти (например, с помощью стандартных библиотечных C функций malloc(), calloc(), realloc()).

Исполняемый код и инициализируемые данные составляют содержимое файла программы, который исполняется в контексте процесса. Пользовательский стек применяется при работе процесса в пользовательском режиме (user-mode).

Под понятием "контекст ядра" объединяются системный контекст и регистровый контекст, рассмотренные на лекции. Мы будем выделять в контексте ядра стек ядра, который используется при работе процесса в режиме ядра (kernel mode), и данные ядра, хранящиеся в структурах, являющихся аналогом блока управления процессом — PCB. Состав данных ядра будет уточняться на последующих семинарах. На этом занятии нам достаточно знать, что в данные ядра входят: идентификатор пользователя — UID, групповой идентификатор пользователя — GID, идентификатор процесса — PID, идентификатор родительского процесса — PPID.

Идентификация процесса

Каждый процесс в операционной системе получает уникальный идентификационный номер – PID (process identificator). При создании нового процесса операционная система пытается присвоить ему свободный номер больший, чем у процесса, созданного перед ним. Если таких свободных номеров не оказывается (например, мы достигли максимально возможного номера для процесса), то операционная система выбирает минимальный номер из всех свободных номеров. В операционной системе Linux присвоение идентификационных номеров процессов начинается с номера 0, который получает процесс kernel при старте операционной системы. Этот номер впоследствии не может быть присвоен никакому другому

процессу. Максимально возможное значение для номера процесса в Linux на базе 32–разрядных процессоров Intel составляет 2^{31}– 1.

Иерархия процессов

В операционной системе UNIX все процессы, кроме одного, создающегося при старте операционной системы, могут быть порождены только какими – либо другими процессами. В качестве прародителя всех остальных процессов в подобных UNIX системах могут выступать процессы с номерами 1 или 0. В операционной системе Linux таким родоначальником, существующим только при загрузке системы, является процесс kernel с идентификатором 0.

Таким образом, все процессы в UNIX связаны отношениями процесс-родитель – процесс-ребенок и образуют генеалогическое дерево процессов. Для сохранения целостности генеалогического дерева в ситуациях, когда процесс-родитель завершает свою работу до завершения выполнения процесса-ребенка, идентификатор родительского процесса в данных ядра процесса-ребенка (PPID – parent process identificator) изменяет свое значение на значение 1, соответствующее идентификатору процесса init, время жизни которого определяет время функционирования операционной системы. Тем самым процесс init как бы усыновляет осиротевшие процессы. Наверное, логичнее было бы заменять PPID не на значение 1, а на значение идентификатора ближайшего существующего процесса-прародителя умершего процесса-родителя, но в UNIX почему-то такая схема реализована не была.

Системные вызовы getppid() и getpid()

Данные ядра, находящиеся в контексте ядра процесса, не могут быть прочитаны процессом непосредственно. Для получения информации о них процесс должен совершить соответствующий системный вызов. Значение идентификатора текущего процесса может быть получено с помощью системного вызова getpid(), а значение идентификатора родительского процесса для текущего процесса – с помощью системного вызова getppid(). Прототипы этих системных вызовов и соответствующие типы данных описаны в системных файлах **<sys/types.h>** и **<unistd.h>**. Системные вызовы не имеют параметров и возвращают идентификатор текущего процесса и идентификатор родительского процесса соответственно.

Системные вызовы getpid() и getppid()
Прототипы системных вызовов

```
#include <sys/types.h>
#include <unistd.h>
pid_t getpid(void);
```

Описание системных вызовов

Системный вызов getpid возвращает идентификатор текущего процесса. Системный вызов getppid возвращает идентификатор процесса-родителя для текущего процесса. Тип данных pid_t является синонимом для одного из целочисленных типов языка С.

Создание процесса в UNIX. Системный вызов fork()

В операционной системе UNIX новый процесс может быть порожден единственным способом – с помощью системного вызова fork(). При этом вновь созданный процесс будет

являться практически полной копией родительского процесса. У порожденного процесса по сравнению с родительским процессом (на уровне уже полученных знаний) изменяются значения следующих параметров: идентификатор процесса – PID; идентификатор родительского процесса – PPID.

Системный вызов для порождения нового процесса
Прототип системного вызова

```
#include <sys/types.h>
#include <unistd.h>
pid_t fork(void);
```

Описание системного вызова

Системный вызов fork служит для создания нового процесса в операционной системе UNIX. Процесс, который инициировал системный вызов fork, принято называть родительским процессом (parent process). Вновь порожденный процесс принято называть процессом-ребенком (child process). Процесс-ребенок является почти полной копией родительского процесса. У порожденного процесса по сравнению с родительским изменяются значения следующих параметров:

• идентификатор процесса;

• идентификатор родительского процесса;

• время, оставшееся до получения сигнала SIGALRM;

• сигналы, ожидавшие доставки родительскому процессу, не будут доставляться порожденному процессу.

При однократном системном вызове возврат из него может произойти дважды: один раз в родительском процессе, а второй раз в порожденном процессе. Если создание нового процесса произошло успешно, то в порожденном процессе системный вызов вернет значение 0, а в родительском процессе – положительное значение, равное идентификатору процесса-ребенка. Если создать новый процесс не удалось, то системный вызов вернет в инициировавший его процесс отрицательное значение.

Системный вызов fork является единственным способом породить новый процесс после инициализации операционной системы UNIX.

В процессе выполнения системного вызова fork() порождается копия родительского процесса и возвращение из системного вызова будет происходить уже как в родительском, так и в порожденном процессах. Этот системный вызов является единственным, который вызывается один раз, а при успешной работе возвращается два раза (один раз в процессе-родителе и один раз в процессе-ребенке)! После выхода из системного вызова оба процесса продолжают выполнение регулярного пользовательского кода, следующего за системным вызовом.

Для того чтобы после возвращения из системного вызова fork() процессы могли определить, кто из них является ребенком, а кто родителем, и, соответственно, по-разному организовать свое поведение, системный вызов возвращает в них разные значения. При успешном создании нового процесса в процесс-родитель возвращается положительное значение, равное идентификатору процесса-ребенка. В процесс-ребенок же возвращается значение 0. Если по какой-либо причине создать новый процесс не удалось, то системный

вызов вернет в инициировавший его процесс значение -1. Таким образом, общая схема организации различной работы процесса-ребенка и процесса-родителя выглядит так:

```
pid = fork();
if(pid == -1){
...
/* ошибка */
...
} else if (pid == 0){
...
/* ребенок */
...
} else {
...
/* родитель */
...
}
```

Завершение процесса. Функция exit()

Существует два способа корректного завершения процесса в программах, написанных на языке С. Первый способ мы использовали до сих пор: процесс корректно завершался по достижении конца функции main() или при выполнении оператора return в функции main(), второй способ применяется при необходимости завершить процесс в каком-либо другом месте программы. Для этого используется функция exit() из стандартной библиотеки функций для языка С. При выполнении этой функции происходит сброс всех частично заполненных буферов ввода-вывода с закрытием соответствующих потоков, после чего инициируется системный вызов прекращения работы процесса и перевода его в состояние **закончил исполнение**. Возврата из функции в текущий процесс не происходит и функция ничего не возвращает. Значение параметра функции exit() – кода завершения процесса – передается ядру операционной системы и может быть затем получено процессом, породившим завершившийся процесс. На самом деле при достижении конца функции main() также неявно вызывается эта функция со значением параметра 0.

Функция для нормального завершения процесса
Прототип функции

```
#include <stdlib.h>
void exit(int status);
```

Описание функции

Функция exit служит для нормального завершения процесса. При выполнении этой функции происходит сброс всех частично заполненных буферов ввода-вывода с закрытием соответствующих потоков (файлов, pipe, FIFO, сокетов), после чего инициируется системный вызов прекращения работы процесса и перевода его в состояние закончил исполнение. Возврата из функции в текущий процесс не происходит, и функция ничего не возвращает.

Значение параметра status (кода завершения процесса) передается ядру операционной системы и может быть затем получено процессом, породившим

завершившийся процесс. При этом используются только младшие 8бит параметра, так что для кода завершения допустимы значения от 0 до 255. По соглашению, код завершения 0означает безошибочное завершение процесса.

Если процесс завершает свою работу раньше, чем его родитель, и родитель явно не указал, что он не хочет получать информацию о статусе завершения порожденного процесса, то завершившийся процесс не исчезает из системы окончательно, а остается в состоянии **закончил исполнение** либо до завершения процесса-родителя, либо до того момента, когда родитель получит эту информацию. Процессы, находящиеся в состоянии **закончил исполнение**, в операционной системе UNIX принято называть процессами-зомби (zombie, defunct).

Изменение пользовательского контекста процесса. Семейство функций для системного вызова exec().

Для изменения пользовательского контекста процесса применяется системный вызов exec(), который пользователь не может вызвать непосредственно. Вызов exec() заменяет пользовательский контекст текущего процесса на содержимое некоторого исполняемого файла и устанавливает начальные значения регистров процессора (в том числе устанавливает программный счетчик на начало загружаемой программы). Этот вызов требует для своей работызадания имени исполняемого файла, аргументов командной строки и параметров окружающей среды. Для осуществления вызова программист может воспользоваться одной из шести функций: execlp(), execvp(), execl() и, execv(), execle(), execve(), отличающихся друг от друга представлением параметров, необходимых для работы системного вызова exec().

Функции изменения пользовательского контекста процесса
Прототипы функций

```
#include <unistd.h>
int execlp(const char *file,
const char *arg0,
... const char *argN,(char *)NULL)
int execvp(const char *file, char *argv[])
int execl(const char *path,
const char *arg0,
... const char *argN,(char *)NULL)
int execv(const char *path, char *argv[])
int execle(const char *path,
const char *arg0,
... const char *argN,(char *)NULL,
char * envp[])
int execve(const char *path, char *argv[],
char *envp[])
```

Описание функций

Для загрузки новой программы в системный контекст текущего процесса используется семейство взаимосвязанных функций, отличающихся друг от друга формой представления параметров.

Аргумент file является указателем на имя файла, который должен быть загружен. Аргумент path –это указатель на полный путь к файлу, который должен быть загружен.

Аргументы arg0, ..., argN представляют собой указатели на аргументы командной строки. Заметим, что аргумент arg0 должен указывать на имя загружаемого файла. Аргумент argv представляет собой массив из указателей на аргументы командной строки. Начальный элемент массива должен указывать на имя загружаемой программы, а заканчиваться массив должен элементом, содержащим указатель NULL.

Аргумент envp является массивом указателей на параметры окружающей среды, заданные в виде строк "переменная=строка". Последний элемент этого массива должен содержать указатель NULL.

Поскольку вызов функции не изменяет системный контекст текущего процесса, загруженная программа унаследует от загрузившего ее процесса следующие атрибуты:

- идентификатор процесса;
- идентификатор родительского процесса;
- групповой идентификатор процесса;
- идентификатор сеанса;
- время, оставшееся до возникновения сигнала SIGALRM;
- текущую рабочую директорию;
- маску создания файлов;
- идентификатор пользователя;
- групповой идентификатор пользователя;
- явное игнорирование сигналов;
- таблицу открытых файлов (если для файлового дескриптора не устанавливался признак "закрыть файл при выполнении exec()").

В случае успешного выполнения возврата из функций в программу, осуществившую вызов, не происходит, а управление передается загруженной программе. В случае неудачного выполнения в программу, инициировавшую вызов, возвращается отрицательное значение.

Поскольку системный контекст процесса при вызове exec() остается практически неизменным, большинство атрибутов процесса, доступных пользователю через системные вызовы (PID, UID, GID, PPID и другие, смысл которых станет понятен по мере углубления наших знаний на дальнейших занятиях), после запуска новой программы также не изменяется.

Важно понимать разницу между системными вызовами *fork()* и *exec()*. Системный вызов *fork()* создает новый процесс, у которого пользовательский контекст совпадает с пользовательским контекстом процесса-родителя. Системный вызов *exec()* изменяет пользовательский контекст текущего процесса, не создавая новый процесс.

Задание для выполнения

Написать программу, которая будет реализовывать следующие функции:

- сразу после запуска получает и сообщает свой ID и ID родительского процесса;
- перед каждым выводом сообщения об ID процесса и родительского процесса эта информация получается заново;

• порождает процессы, формируя генеалогическое дерево согласно варианту, сообщая, что "процесс с ID таким-то породил процесс с таким-то ID";

• перед завершением процесса сообщить, что "процесс с таким-то ID и таким-то ID родителя завершает работу";

• один из процессов должен вместо себя запустить программу, указанную в варианте задания.

На основании выходной информации программы предыдущего пункта изобразить генеалогическое дерево процессов (с указанием идентификаторов процессов). Объяснить каждое выведенное сообщение и их порядок в предыдущем пункте.

Варианты индивидуальных заданий

В столбце **fork** описано генеалогическое древо процессов: каждая цифра указывает на относительный номер (не путать с pid) процесса, являющегося родителем для данного процесса. Например, строка

0 1 1 1 3

означает, что первый процесс не имеет родителя среди ваших процессов (порождается и запускается извне), второй, третий и четвертый – порождены первым, пятый – третьим.

В столбце exec указан номер процесса, выполняющего вызов **exec**, команды для которого указаны в последнем столбце. Запускайте команду обязательно с какими-либо параметрами. Если в команде присутствуют квадратные скобки, то это значит, что у вас есть дополнительные параметры, которые вы выбираете самостоятельно для запуска команды.

№	fork	exec	
1	0 1 1 1 3 3 5	1	ls -l
2	0 1 2 2 3 4 6	2	ps -ax
3	0 1 1 2 2 5 6	3	cat [file1] [file 2]
4	0 1 1 1 2 5 5	4	sort -r [file]
5	0 1 1 2 2 3 3	5	df -h
6	0 1 1 2 4 4 4	6	touch [file1]
7	0 1 1 1 3 3 5	7	ps –u [user]
8	0 1 2 2 3 4 5	1	pwd
9	0 1 1 2 2 5 6	2	whoami
10	0 1 1 1 2 5 5	3	whereis [команд]

ЛАБОРАТОРНАЯ РАБОТА №5. «ВВОД/ВЫВОД»

Знакомимся. Понятие о потоке ввода-вывода

Среди всех категорий средств коммуникации наиболее употребительными являются каналы связи, обеспечивающие достаточно безопасное и достаточно информативное взаимодействие процессов.

Существует две модели передачи данных по каналам связи 0 поток ввода-вывода и сообщения. Из них более простой является потоковая модель, в которой операции передачи/приема информации вообще не интересуются содержимым того, что передается или принимается. Вся информация в канале связи рассматривается как непрерывный поток байт, не обладающий никакой внутренней структурой.

Понятие о работе с файлами через системные вызовы и стандартную библиотеку ввода-вывода для языка С

Потоковая передача информации может осуществляться не только между процессами, но и между процессом и устройством ввода-вывода, например между процессом и диском, на котором данные представляются в виде файла. Поскольку понятие файла должно быть знакомо изучающим этот курс, а системные вызовы, использующиеся для потоковой работы с файлом, во многом соответствуют системным вызовам, применяемым для потокового общения процессов, мы начнем наше рассмотрение именно с механизма потокового обмена между процессом и файлом.

Как мы надеемся, из курса программирования на языке С вам известны функции работы с файлами из стандартной библиотеки ввода-вывода, такие как fopen(), fread(), fwrite(), fprintf(), fscanf(), fgets() и т.д. Эти функции входят как неотъемлемая часть в стандарт ANSI 2 на язык С и позволяют программисту получать информацию из файла или записывать ее в файл при условии, что программист обладает определенными знаниями о содержимом передаваемых данных. Так, например, функция fgets() используется для ввода из файла последовательности символов, заканчивающейся символом '\n' – перевод каретки. Функция fscanf() производит ввод информации, соответствующей заданному формату, и т. д. С точки зрения потоковой модели операции, определяемые функциями стандартной библиотеки ввода-вывода, не являются потоковыми операциями, так как каждая из них требует наличия некоторой структуры передаваемых данных.

В операционной системе UNIX эти функции представляют собой надстройку, сервисный интерфейс, над системными вызовами, осуществляющими прямые потоковые операции обмена информацией между процессом и файлом и не требующими никаких знаний о том, что она содержит. Чуть позже мы кратко познакомимся с системными вызовами open(), read(), write() и close(), которые применяются для такого обмена, но сначала нам нужно ввести еще одно понятие – понятие файлового дескриптора.

Файловый дескриптор

Информация о файлах, используемых процессом, входит в состав его системного контекста и хранится в его блоке управления PCB. В операционной системе UNIX можно упрощенно полагать, что информация о файлах, с которыми процесс осуществляет операции потокового обмена, наряду с информацией о потоковых линиях связи, соединяющих процесс с другими процессами и устройствами ввода-вывода, хранится в некотором массиве, получившем название таблицы открытых файлов или таблицы файловых дескрипторов.

Индекс элемента этого массива, соответствующий определенному потоку ввода-вывода, получил название файлового дескриптора для этого потока. Таким образом, файловый дескриптор представляет собой небольшое целое неотрицательное число, которое для текущего процесса в данный момент времени однозначно определяет некоторый действующий канал ввода- вывода.

Некоторые файловые дескрипторы на этапе старта любой программы ассоциируются со стандартными потоками ввода-вывода. Так, например, файловый дескриптор «0» соответствует стандартному потоку ввода, файловый дескриптор «1» стандартному потоку вывода, файловый дескриптор «2» стандартному потоку для вывода ошибок. В нормальном интерактивном режиме работы стандартный поток ввода связывает процесс с клавиатурой, а стандартные потоки вывода и вывода ошибок с текущим терминалом. Более детально строение структур данных, содержащих информацию о потоках ввода-вывода, ассоциированных с процессом, мы будем рассматривать позже, при изучении организации файловых систем в UNIX.

Открытие файла. Системный вызов open()

Файловый дескриптор используется в качестве параметра, описывающего поток ввода- вывода, для системных вызовов, выполняющих операции над этим потоком. Поэтому прежде чем совершать операции чтения данных из файла и записи их в файл, мы должны поместить информацию о файле в таблицу открытых файлов и определить соответствующий файловый дескриптор. Для этого применяется процедура открытия файла, осуществляемая системным вызовом open().

Прототип системного вызова

```
#include <fcntl.h>

int open(char *path, int flags);
int open(char *path, int flags, int mode);
```

Описание системного вызова

Системный вызов open предназначен для выполнения операции открытия файла и, в случае ее удачного осуществления, возвращает файловый дескриптор открытого файла (небольшое неотрицательное целое число, которое используется в дальнейшем для других операций с этим файлом).

Параметр path является указателем на строку, содержащую полное или относительное имя файла.

Параметр flags может принимать одно из следующих трех значений:

O_RDONLY – если над файлом в дальнейшем будут совершаться только операции чтения;

O_WRONLY – если над файлом в дальнейшем будут осуществляться только операции записи;

O_RDWR – если над файлом будут осуществляться и операции чтения, и операции записи.

Каждое из этих значений может быть скомбинировано посредством операции "побитовое или (|)" с одним или несколькими флагами:

O_CREAT – если файла с указанным именем не существует, он должен быть создан;

O_EXCL – применяется совместно с флагом O_CREAT. При совместном их использовании и существовании файла с указанным именем, открытие файла не производится и констатируется ошибочная ситуация;

O_NDELAY – запрещает перевод процесса в состояние ожидание при выполнении операции открытия и любых последующих операциях над этим файлом;

O_APPEND – при открытии файла и перед выполнением каждой операции записи (если она, конечно, разрешена) указатель текущей позиции в файле устанавливается на конец файла;

O_TRUNC – если файл существует, уменьшить его размер до 0, с сохранением существующих атрибутов файла, кроме, быть может, времен последнего доступа к файлу и его последней модификации.

Кроме того, в некоторых версиях операционной системы UNIX могут применяться дополнительные значения флагов:

O_SYNC – любая операция записи в файл будет блокироваться (т. е. процесс будет переведен в состояние ожидание) до тех пор, пока записанная информация не будет физически помещена на соответсвующий нижележащий уровень hardware;

O_NOCTTY – если имя файла относится к терминальному устройству, оно не становится управляющим терминалом процесса, даже если до этого процесс не имел управляющего терминала.

Параметр mode устанавливает атрибуты прав доступа различных категорий пользователей к новому файлу при его создании. Он обязателен, если среди заданных флагов присутствует флаг O_CREAT, и может быть опущен в противном случае. Этот параметр задается как сумма следующих восьмеричных значений:

0400 – разрешено чтение для пользователя, создавшего файл;
0200 – разрешена запись для пользователя, создавшего файл;
0100 – разрешено исполнение для пользователя, создавшего файл;
0040 – разрешено чтение для группы пользователя, создавшего файл;
0020 – разрешена запись для группы пользователя, создавшего файл;
0010 – разрешено исполнение для группы пользователя, создавшего файл;
0004 – разрешено чтение для всех остальных пользователей;
0002 – разрешена запись для всех остальных пользователей;
0001 – разрешено исполнение для всех остальных пользователей.

При создании файла реально устанавливаемые права доступа получаются из стандартной комбинации параметра mode и маски создания файлов текущего процесса umask, а именно 0 они равны mode & ~umask.

При открытии файлов типа FIFO системный вызов имеет некоторые особенности поведения по сравнению с открытием файлов других типов. Если FIFO открывается только для чтения, и не задан флаг O_NDELAY, то процесс, осуществивший системный вызов, блокируется до тех пор, пока какой-либо другой процесс не откроет FIFO на запись. Если флаг O_NDELAY задан, то возвращается значение файлового дескриптора, ассоциированного с FIFO. Если FIFO открывается только для записи, и не задан флаг O_NDELAY, то процесс, осуществивший системный вызов, блокируется до тех пор, пока какой-либо другой процесс

не откроет FIFO на чтение. Если флаг O_NDELAY задан, то констатируется возникновение ошибки и возвращается значение -l.

Возвращаемое значение

Системный вызов возвращает значение файлового дескриптора для открытого файла при нормальном завершении и значение -1 при возникновении ошибки.

Системный вызов open() использует набор флагов для того, чтобы специфицировать операции, которые предполагается применять к файлу в дальнейшем или которые должны быть выполнены непосредственно в момент открытия файла. Из всего возможного набора флагов на текущем уровне знаний нас будут интересовать только флаги O_RDONLY, O_WRONLY, O_RDWR, O_CREAT и O_EXCL. Первые три флага являются взаимоисключающими: хотя бы один из них должен быть применен и наличие одного из них не допускает наличия двух других. Эти флаги описывают набор операций, которые, при успешном открытии файла, будут разрешены над файлом в дальнейшем: только чтение, только запись, чтение и запись.

Как вам известно, у каждого файла существуют атрибуты прав доступа для различных категорий пользователей. Если файл с заданным именем существует на диске, и права доступа к нему для пользователя, от имени которого работает текущий процесс, не противоречат запрошенному набору операций, то операционная система сканирует таблицу открытых файлов от ее начала к концу в поисках первого свободного элемента, заполняет его и возвращает индекс этого элемента в качестве файлового дескриптора открытого файла. Если файла на диске нет, не хватает прав или отсутствует свободное место в таблице открытых файлов, то констатируется возникновение ошибки.

В случае, когда мы допускаем, что файл на диске может отсутствовать, и хотим, чтобы он был создан, флаг для набора операций должен использоваться в комбинации с флагом O_CREAT. Если файл существует, то все происходит по рассмотренному выше сценарию. Если файла нет, сначала выполняется создание файла с набором прав, указанным в параметрах системного вызова. Проверка соответствия набора операций объявленным правам доступа может и не производиться. В случае, когда мы требуем, чтобы файл на диске отсутствовал и был создан в момент открытия, флаг для набора операций должен использоваться в комбинации с флагами O_CREAT и O_EXCL.

Системные вызовы read(), write(), close()

Для совершения потоковых операций чтения информации из файла и ее записи в файл применяются системные вызовы read() и write().

Прототипы системных вызовов

```
#include <sys/types.h>
#include <unistd.h>
size_t read(int fd, void *addr, size_t nbytes);
size_t write(int fd, void *addr,size_t nbytes);
```

Описание системных вызовов Системные вызовы read и write предназначены для осуществления потоковых операций ввода (чтения) и вывода (записи) информации над

каналами связи, описываемыми файловыми дескрипторами, т.е. для файлов, pipe, FIFO и socket.

Параметр fd является файловым дескриптором созданного ранее потокового канала связи, через который будет отсылаться или получаться информация, т. е. значением, которое вернул один из системных вызовов open(), pipe() или socket().

Параметр addr представляет собой адрес области памяти, начиная с которого будет браться информация для передачи или размещаться принятая информация.

Параметр nbytes для системного вызова write определяет количество байт, которое должно быть передано, начиная с адреса памяти addr. Параметр nbytes для системного вызова read определяет количество байт, которое мы хотим получить из канала связи и разместить в памяти, начиная с адреса addr.

Возвращаемые значения

В случае успешного завершения системный вызов возвращает количество реально посланных или принятых байт. Заметим, что это значение (большее или равное 0) может не совпадать с заданным значением параметра nbytes, а быть меньше, чем оно, в силу отсутствия места на диске или в линии связи при передаче данных или отсутствия информации при ее приеме. При возникновении какой-либо ошибки возвращается отрицательное значение.

Особенности поведения при работе с файлами

При работе с файлами информация записывается в файл или читается из файла, начиная с места, определяемого указателем текущей позиции в файле. Значение указателя увеличивается на количество реально прочитанных или записанных байт. При чтении информации из файла она не пропадает из него. Если системный вызов read() возвращает значение 0, то это означает, что файл прочитан до конца.

Мы сейчас не акцентируем внимание на понятии указателя текущей позиции в файле и взаимном влиянии значения этого указателя и поведения системных вызовов. После завершения потоковых операций процесс должен выполнить операцию закрытия потока ввода-вывода, во время которой произойдет окончательный сброс буферов на линии связи, освободятся выделенные ресурсы операционной системы, и элемент таблицы открытых файлов, соответствующий файловому дескриптору, будет отмечен как свободный. За эти действия отвечает системный вызов close(). Надо отметить, что при завершении работы процесса с помощью явного или неявного вызова функции exit() происходит автоматическое закрытие всех открытых потоков ввода-вывода.

Системный вызов close

Прототип системного вызова

```
#include <unistd.h>
int close(int fd);
```

Описание системного вызова

Системный вызов close предназначен для корректного завершения работы с файлами и другими объектами ввода-вывода, которые описываются в операционной системе через файловые дескрипторы: pipe, FIFO, socket. Параметр fd является дескриптором

соответствующего объекта, т. е. значением, которое вернул один из системных вызовов open(), pipe() или socket().

Возвращаемые значения

Системный вызов возвращает значение 0 при нормальном завершении и значение -1 при возникновении ошибки.

Код программы для записи информации в файл

Для иллюстрации сказанного давайте рассмотрим следующую программу:

```
/*Программа, иллюстрирующая использование системных вызовов open(), write() и close() для записи информации в файл */

#include <sys/types.h>
#include <fcntl.h>
#include <stdio.h>
int main(){
 int fd;
 size_t size;
 char string[] = "Hello, world!";
 /* Обнуляем маску создания файлов текущего процесса для того, чтобы права доступа у создаваемого файла точно соответствовали параметру вызова open() */
 (void)umask(0);
 /* Попытаемся открыть файл с именем myfile в текущей директории только для операций вывода. Если файла не существует, попробуем  его создать с правами доступа 0666, т. е. read-write для всех категорий пользователей */
 if((fd = open("myfile", O_WRONLY | O_CREAT, 0666)) < 0){
 /* Если файл открыть не удалось, печатаем об этом сообщение и прекращаем  работу */
        printf("Can\'t open file\n");
        exit(-1);
 }
 /* Пробуем записать в файл 14 байт из нашего массива, т.е. всю строку "Hello, world!" вместе с признаком конца строки */
 size = write(fd, string, 14);
 if(size != 14){
 /* Если записалось меньшее количество байт, сообщаем об ошибке */
        printf("Can\'t write all string\n");
        exit(-1);
 }
 /* Закрываем файл */
 if(close(fd) < 0){
        printf("Can\'t close file\n");
 }
 return 0;
}
```

Обратите внимание на использование системного вызова **umask()** с параметром **0** для того, чтобы права доступа к созданному файлу точно соответствовали указанным в системном вызове open().

Если вам необходимо считывать стандартный поток ввода, то используйте файловый дескриптор равный файловому дескриптору стандартного потока ввода, то есть **fd=0**. Если вам

необходимо вывести информацию на экран (стандартный поток вывода), то используйте **fd=1** и системный вызов **write().**

С помощью конвейера можно перенаправить поток вывода команды на поток ввода пользовательской программы следующим образом:

```
user@ubuntu:~$ gcc lab5.c –o lab5
user@ubuntu:~$ ls -l|./lab5.out
```

Задание для выполнения

Часть I

Ознакомиться с руководством по системным вызовам **open**, **read**, **write**, **close**. Вспомнить, что такое *конвейер* и *перенаправление ввода-вывода* (этот пункт задания может оказаться трудновыполнимым, если соответствующие знания не были приобретены в процессе работы над л/р №1-3). Для выполнения вам необходимо создать две отдельные программы, на вход которых вы будете отправлять через конвейер команду, которая указана первой в комбинации. Протестировать свою программу в различных комбинациях с указанными в задании командами.

Например:

Написать программу для обработки строк файла.Протестировать на различных файлах с использованием конвейеров в различных комбинациях вашей программы и команд **cat, sort, head, tail.**

Запуск программы:

```
user@ubuntu:~$ cat file1|./lab5.out
user@ubuntu:~$ cat file1|sort|head./lab5.out и т.д.
```

Часть II

Вариант 1. Написать программу, который получает на вход список файлов с их размерами, выводит в стандартный поток вывода имена файлов, отсортированные по размеру, а в поток ошибок - общий объем, занимаемый этими файлами. Протестировать с использованием команд **du, sort, awk, head, tail.**

Вариант 2. Написать программу, которая получает со стандартного потока ввода содержимое каталога /var и выводит в стандартный поток вывода информацию только о файлах, группа которых не является система (root). Протестировать на различных файлах с использованием конвейеров в различных комбинациях вашей программы и команд **ls, sort, head, tail.**

Вариант 3. Написать программу, которая получает со стандартного потока ввода содержимое любого текстового файла и выводит в стандартный поток вывода те строки, в которых больше 3 слов. Протестировать на различных файлах с использованием конвейеров в различных комбинациях вашей программы и команд **cat, sort, head, tail.**

Вариант 4. Написать программу, которая получает со стандартного потока ввода права доступа к файлам каталога, и выводит в стандартный поток вывода те из них, у которых

установлен бит запуска владельцем. Протестировать на различных каталогах с использованием конвейеров в различных комбинациях вашей программы и команд **ls, sort, head, tail.**

Вариант 5. Написать программу, которая получает со стандартного потока ввода содержимое любого текстового файла и выводит в стандартный поток вывода те его строки, которые начинаются с цифры, заменив в этих строках все буквы X на Y. Протестировать на различных файлах с использованием конвейеров в различных комбинациях вашей программы и команд **cat, sort, head, tail.**

Вариант 6. Написать программу, которая получает со стандартного потока ввода список активных процессов, и выводит в стандартный поток вывода процессы только с четными PID, добавив к имени процессов любое случайное число. Протестировать с использованием конвейеров в различных комбинациях вашей программы и команд **ps, sort, head, tail.**

Вариант 7. Написать программу, которая получает со стандартного потока ввода содержимое любого текстового файла и выводит его в стандартный поток вывода, поменяв местами слова в середине слов (первая и последняя буквы слов остаются на своих местах). Протестировать на различных файлах с использованием конвейеров в различных комбинациях вашей программы и команд **cat, sort, head, tail.**

Вариант 8. Написать программу, которая получает со стандартного потока ввода содержимое любого текстового файла и выводит в стандартный поток вывода строки, начинающиеся на гласную букву, а в поток ошибки – порядковый номер выведенной строки. Протестировать на различных файлах с использованием конвейеров в различных комбинациях вашей программы и команд **cat, sort, head, tail.**

Вариант 9. Написать программу, которая получает со стандартного потока ввода строки в формате, аналогичном **/etc/passwd** (login : password : UID : GID : GECOS : home : shell), и выводит в стандартный поток вывода строки, с четными значениями UID, а в поток ошибки – с нечетными. Протестировать на различных файлах с использованием конвейеров в различных комбинациях вашей программы и команд **cat, sort, head, tail.**

Вариант 10. Написать программу, которая получает со стандартного потока ввода текст руководства и выводит в стандартный поток вывода его строки, начинающиеся на гласную букву, а в поток ошибки – порядковый номер выведенной строки. Протестировать на различных файлах с использованием конвейеров в различных комбинациях вашей программы и команд **cat, sort, head, tail.**

Часть III

В зависимости от варианта добавьте к своей программе следующую функциональность: В зависимости от варианта добавьте к своей программе следующую функциональность:

Вариант 1. В текущем каталоге создайте два файла, в один из которых выводите нечетные, а во второй - четные порядковые номера и соответствующие имена файлов.

Вариант 2. Создайте отдельные текстовые файлы для каждого пользователя, и поместите туда имена всех созданных им файлов.

Вариант 3. Создайте отдельные текстовые файлы, имеющие числовые значения, и поместите туда все слова, которым соответствует такое количество символов.

Вариант 4. Выведите в отдельные файлы списки запускаемых владельцем, читаемых владельцем и записываемых владельцем файлов.

Вариант 5. Откройте любой другой текстовый файл и выводите в стандартный поток вывода строки по очереди - согласно заданию Части II и из этого файла.

Вариант 6. Откройте любой текстовый файл и добавляйте в стандартном выводе к имени процессов не число, а очередное слово из этого файла.

Вариант 7. В текущем каталоге создайте файл, в котором сформируйте словарь слов с переставленными буквами, в виде: слово – совло, словарь – свлораь,

Вариант 8. В текущем каталоге создайте два файла, в один из которых выводите нечетные, а во второй - четные строки, начинающиеся на согласную букву.

Вариант 9. В текущем каталоге создайте два файла, в один из которых выводите UID и имя пользователя, а в другой – GID и имя пользователя.

Вариант 10. Создайте отдельные текстовые файлы для каждой секции руководства, и поместите туда эти секции.

ЛАБОРАТОРНАЯ РАБОТА №6. «СРЕДСТВА МЕЖПРОЦЕССНОГО ВЗАИМОДЕЙСТВИЯ»

Все процессы в ***Linux*** выполняются в раздельных адресных пространствах и для организации межпроцессного взаимодействия необходимо использовать специальные методы:

- общие файлы;
- общую или разделяемую память;
- очереди сообщений;
- сигналы;
- каналы;
- семафоры (см. ЛР №7).

Теоретические сведения

Общие файлы

При использовании общих файлов оба процесса открывают один и тот же файл, с помощью которого и обмениваются информацией. Для ускорения работы следует использовать файлы, отображаемые в памяти при помощи системного вызова ***mmap()***.

Прототип системного вызова

```
#include <unistd.h>
#include <sys/mman.h>

void * mmap(void *start, size_t length, int prot , int flags, int fd, off_t offset);
```

Описание системного вызова

Функция ***mmap*** отображает ***length*** байтов, начиная со смещения ***offset*** файла, определенного файловым описателем ***fd*** в память, начиная с адреса ***start***. Последний параметр ***offset*** необязателен, и обычно равен ***0***. Настоящее местоположение отраженных данных возвращается самой функцией ***mmap***, и никогда не бывает равным ***0***. Аргумент ***prot*** описывает желаемый режим защиты памяти (он не должен конфликтовать с режимом открытия файла):

- ***PROT_EXEC*** данные в области памяти могут исполняться;
- ***PROT_READ*** данные в области памяти можно читать;
- ***PROT_WRITE*** в область памяти можно записывать информацию;
- ***PROT_NONE*** доступ к этой области памяти запрещен.

Параметр ***fd*** должно быть корректным описателем файла, если только не установлено MAP_ANONYMOUS, так как в этом случае аргумент игнорируется.

Параметр ***offset*** должен быть пропорционален размеру страницы, получаемому при помощи функции ***getpagesize()***.

Параметр ***flags*** задает тип отражаемого объекта, опции отражения и указывает, принадлежат ли отраженные данные только этому процессу или их могут читать другие. Он состоит из комбинации следующих битов:

MAP_FIXED использование этой опции не рекомендуется;

MAP_SHARED разделить использование этого отражения с другими процессами, отражающими тот же объект. Запись информации в эту область памяти будет эквивалентна записи в файл. Файл может не обновляться до вызова функций ***msync()*** или ***munmap()***;

MAP_PRIVATE создать неразделяемое отражение с механизмом ***copy-on-write***. Запись в эту область памяти не влияет на файл. Не определено, являются или нет изменения в файле после вызова ***mmap*** видимыми в отраженном диапазоне.

Вы должны задать либо ***MAP_SHARED***, либо ***MAP_PRIVATE***.

Эти три флага описаны в POSIX.1b (бывшем POSIX.4) and SUSv2. В Linux также анализируются следующие нестандартные флаги:

MAP_NORESERVE (используется вместе с MAP_PRIVATE.) Не выделяет страницы пространства подкачки для этого отображения. Если пространство подкачки выделяется, то это частное пространство копирования-при-записи может быть изменено. Если оно не выделено, то можно получить *SIGSEGV* (*SIG* – общий префикс сигналов (от англ. signal), *SEGV* – англ. segmentation violation – нарушение сегментации.) при записи и отсутствии доступной памяти.

MAP_LOCKED (Linux 2.5.37 и выше). Блокировать страницу или размеченную область в памяти так, как это делает **mlock()**. Этот флаг игнорируется в старых ядрах.

MAP_GROWSDOWN. Используется для стеков. Для VM системы ядра обозначает, что отображение должно распространяться вниз по памяти.

MAP_ANONYMOUS. Отображение не резервируется ни в каком файле; аргументы *fd* и *offset* игнорируются, поэтому использование данного флага не имеет смысла при реализации межпроцессного взаимодействия через общие файлы. Этот флаг вместе с MAP_SHARED реализован с Linux 2.4.

MAP_ANON. Псевдоним для MAP_ANONYMOUS.

MAP_32BIT. Поместить размещение в первые 2Гб адресного пространства процесса. Игнорируется, если указано *MAP_FIXED*. Этот флаг сейчас поддерживается только на x86-64 для 64-битных программ.

Некоторые системы документируют дополнительные флаги MAP_AUTOGROW, MAP_AUTORESRV, MAP_COPY и MAP_LOCAL.

Пример использования системного вызова

```
#include <stdio.h>
#include <sys/mman.h>
```

Разделяемая память

Использование разделяемой памяти заключается в создании специальной области памяти, позволяющей иметь к ней доступ нескольким процессам. Системные вызовы для работы с разделяемой памятью:

Прототип системного вызова

```
#include <sys/mman.h>
int shm_open (const char *name, int oflag, mode_t mode);
int shm_unlink (const char *name);
```

Описание системного вызова

Вызов ***shm_open*** создает и открывает новый (или уже существующий) объект разделяемой памяти. При открытии с помощью функции ***shm_open()*** возвращается файловый дескриптор. Имя ***name*** трактуется стандартным для рассматриваемых средств межпроцессного взаимодействия образом. Посредством аргумента ***oflag*** могут указываться флаги ***O_RDONLY, O_RDWR, O_CREAT, O_EXCL*** и/или ***O_TRUNC***. Если объект создается, то режим доступа к нему формируется в соответствии со значением ***mode*** и маской создания файлов процесса. Функция ***shm_unlink*** выполняет обратную операцию, удаляя объект, предварительно созданный с помощью ***shm_open***.

После подключения сегмента разделяемой памяти к виртуальной памяти процесса этот процесс может обращаться к соответствующим элементам памяти с использованием обычных машинных команд чтения и записи, не прибегая к использованию дополнительных системных вызовов.

Пример использования системного вызова
Очереди сообщений

Очереди сообщений (***queue***) являются более сложным методом связи взаимодействующих процессов по сравнению с программными каналами. С помощью очередей также можно из одной или нескольких задач независимым образом посылать сообщения некоторой задаче-приемнику. При этом только процесс-приемник может читать и удалять сообщения из очереди, а процессы-клиенты имеют право лишь помещать в очередь свои сообщения. Очередь работает только в одном направлении, если необходима двухсторонняя связь, следует создать две очереди. Очереди сообщений предоставляют возможность использовать несколько дисциплин обработки сообщений:

- ***FIFO*** – сообщение, записанное первым, будет прочитано первым;
- ***LIFO*** – сообщение, записанное последним, будет прочитано первым;
- приоритетная – сообщения читаются с учетом их приоритетов;
- произвольный доступ – можно читать любое сообщение, а программный канал обеспечивает только дисциплину ***FIFO***.

Для открытия очереди служит функция ***mq_open()***, которая, по аналогии с файлами, создает описание открытой очереди и ссылающийся на него дескриптор типа ***mqd_t***, возвращаемый в качестве нормального результата.

Прототип системного вызова

```
#include <mqueue.h>

mqd_t mq_open(const char *name, int oflag);
mqd_t mq_open(const char *name, int oflag, mode_t mode,
        struct mq_attr *attr);

ssize_t mq_receive(mqd_t mqdes, char *msg_ptr,
            size_t msg_len, unsigned int *msg_prio);

int mq_send(mqd_t mqdes, const char *msg_ptr,
        size_t msg_len, unsigned int msg_prio);
```

Описание системного вызова mq_open()

Аргумент ***oflag*** может принимать одно из следующих значений: O_RDONLY, O_WRONLY, O_RDWR в сочетании с O_CREAT, O_EXCL, O_NONBLOCK. Все эти флаги описаны в лабораторной работе №5.

При создании новой очереди (указан флаг O_CREAT и очередь сообщений еще не существует) требуется указание аргументов ***mode*** и ***attr***.

Возможные значения аргумента ***mode***:

- S_IRUSR: Владелец — чтение.
- S_IWUSR: Владелец — запись.
- S_IRGRP: Группа — чтение.
- S_IWGRP: Группа — запись.
- S_IROTH: Прочие — чтение.
- S_IWOTH: Прочие — запись.

Аргумент ***attr*** позволяет задать некоторые атрибуты очереди. Если в качестве этого аргумента задать нулевой указатель, очередь будет создана с атрибутами по умолчанию.

```
struct mq_attr {
  long mq_flags;      /* Flags (ignored for mq_open()) */
  long mq_maxmsg;     /* Max. # of messages on queue */
  long mq_msgsize;    /* Max. message size (bytes) */
  long mq_curmsgs;    /* # of messages currently in queue
                         (ignored for mq_open()) */
};
```

Возвращаемое функцией ***mq_open()*** значение называется дескриптором очереди сообщений, но оно не обязательно должно быть (и, скорее всего, не является) небольшим целым числом, как дескриптор файла или программного сокета. Это значение используется в качестве первого аргумента оставшихся семи функций для работы с очередями сообщений.

Описание системных вызовов mq_send() и mq_receive()

Функции ***mq_send()*** помещает сообщение из ***msg_len*** байт, на которое указывает аргумент ***msg_ptr***, в очередь, заданную дескриптором ***mqdes*** (если она не полна), в соответствии с приоритетом ***msg_prio*** (большим значениям msg_prio соответствует более высокий приоритет сообщения ; допустимый диапазон - от 0 до MQ_PRIO_MAX).

Если очередь полна, а флаг O_NONBLOCK не установлен, вызов mq_send() блокируется до появления свободного места.

Для извлечения (разумеется, с удалением) сообщений из очереди служат функции ***mq_receive()***. Извлекается самое старое из сообщений с самым высоким приоритетом и помещается в буфер, на который указывает аргумент ***msg_ptr***. Если размер буфера (значение аргумента ***msg_len***) меньше атрибута очереди ***mq_msgsize***, вызов завершается неудачей. Если значение ***msg_prio*** отлично от NULL, в указуемый объект помещается приоритет принятого сообщения.

Программные каналы (pipe)

Наиболее простым способом для передачи информации с помощью потоковой модели между различными процессами или даже внутри одного процесса в операционной системе UNIX является *pipe* (канал, труба, конвейер).

Важное отличие программного канала от файла заключается в том, что прочитанная информация немедленно удаляется из него и не может быть прочитана повторно. *Pipe* можно представить себе в виде трубы ограниченной емкости, расположенной внутри адресного пространства операционной системы, доступ к входному и выходному отверстию которой осуществляется с помощью системных вызовов. В действительности pipe представляет собой область памяти, недоступную пользовательским процессам напрямую, зачастую организованную в виде кольцевого буфера (хотя существуют и другие виды организации). По буферу при операциях чтения и записи перемещаются два указателя, соответствующие входному и выходному потокам. При этом выходной указатель никогда не может перегнать входной и наоборот.

Для создания нового экземпляра такого кольцевого буфера внутри операционной системы используется системный вызов pipe().

Прототип системного вызова

```
#include <unistd.h>
int pipe(int *fd);
```

Описание системного вызова

Системный вызов ***pipe()*** предназначен для создания программного канала внутри операционной системы. Параметр ***fd*** является указателем на массив из двух целых

переменных. При нормальном завершении вызова в первый элемент массива ***fd[0]*** будет занесен файловый дескриптор, соответствующий выходному потоку данных программного канала и позволяющий выполнять только операцию чтения, а во второй элемент массива ***fd[1]*** будет занесен файловый дескриптор, соответствующий входному потоку данных и позволяющий выполнять только операцию записи.

Системный вызов возвращает значение 0 при нормальном завершении и отрицательное значение при возникновении ошибок.

В процессе работы системный вызов организует выделение области памяти под буфер и указатели и заносит информацию, соответствующую входному и выходному потокам данных, в два элемента таблицы открытых файлов, связывая тем самым с каждым программным каналом два файловых дескриптора. Для одного из них разрешена только операция чтения из программного канала, а для другого только операция записи в pipe. Для выполнения этих операций мы можем использовать те же самые системные вызовы ***read()*** и ***write(),*** что и при работе с файлами.

Естественно, по окончании использования входного или/и выходного потока данных, нужно закрыть соответствующий поток с помощью системного вызова ***close()*** для освобождения системных ресурсов. Необходимо отметить, что, когда все процессы, использующие pipe, закрывают все ассоциированные с ним файловые дескрипторы, операционная система ликвидирует pipe. Таким образом, время существования программного канала в системе не может превышать время жизни процессов, работающих с ним.

Понятно, что если бы все достоинство pip'ов сводилось к замене функции копирования из памяти в память внутри одного процесса на пересылку информации через операционную систему, то овчинка не стоила бы выделки. Однако таблица открытых файлов наследуется процессом-ребенком при порождении нового процесса системным вызовом fork() и входит в состав неизменяемой части системного контекста процесса при системном вызове exec() (за исключением тех потоков данных, для файловых дескрипторов которых был специальными средствами выставлен признак, побуждающий операционную систему закрыть их при выполнении exec(), однако их рассмотрение выходит за рамки нашего курса). Это обстоятельство позволяет организовать передачу информации через pipe между родственными процессами, имеющими общего прародителя, создавшего pipe.

Pipe служит для организации однонаправленной или симплексной связи. Если бы в предыдущем примере мы попытались организовать через pipe двустороннюю связь, когда процесс-родитель пишет информацию в pipe, предполагая, что ее получит процесс-ребенок, а затем читает информацию из pip6 a, предполагая, что ее записал порожденный процесс, то могла бы возникнуть ситуация, в которой процесс-родитель прочитал бы собственную информацию, а процесс-ребенок не получил бы ничего. Для использования одного pip6 a в двух направлениях необходимы специальные средства синхронизации процессов. Более простой способ организации двунаправленной связи между родственными процессами заключается в использовании двух pipe.

Необходимо отметить, что в некоторых UNIX-подобных системах (например, в Solaris2) реализованы полностью дуплексные pip6 ы. В таких системах для обоих файловых дескрипторов, ассоциированных с pip'ом, разрешены и операция чтения, и операция записи. Однако такое поведение не характерно для pip6 ов и не является переносимым.

Системные вызовы read() и write() имеют определенные особенности поведения при работе с pip6 ом, связанные с его ограниченным размером, задержками в передаче данных и возможностью блокирования обменивающихся информацией процессов. Организация запрета блокирования этих вызовов для pipe выходит за рамки нашего курса.

Будьте внимательны при написании программ, обменивающихся большими объемами информации через pipe. Помните, что за один раз из pip6 a может прочитаться меньше информации, чем вы запрашивали, и за один раз в pipe может записаться меньше информации, чем вам хотелось бы. Проверяйте значения, возвращаемые вызовами!

Одна из особенностей поведения блокирующегося системного вызова read() связана с попыткой чтения из пустого pip'a. Если есть процессы, у которых этот pipe открыт для записи, то системный вызов блокируется и ждет появления информации. Если таких процессов нет, он вернет значение 0 без блокировки процесса. Эта особенность приводит к необходимости закрытия файлового дескриптора, ассоциированного с входным концом pip'a, в процессе, который будет использовать pipe для чтения (close(fd*1+) в процессе-ребенке в программе из раздела "Прогон программы для организации однонаправленной связи между родственными процессами через pipe"). Аналогичной особенностью поведения при отсутствии процессов, у которых pipe открыт для чтения, обладает и системный вызов write(), с чем связана необходимость закрытия файлового дескриптора, ассоциированного с выходным концом pip'a, в процессе, который будет использовать pipe для записи (close(fd*0+) в процессе-родителе в той же программе).

Именованные каналы (FIFO)

Как мы выяснили, доступ к информации о расположении pip'a в операционной системе и его состоянии может быть осуществлен только через таблицу открытых файлов процесса, создавшего pipe, и через унаследованные от него таблицы открытых файлов процессов- потомков. Поэтому изложенный выше механизм обмена информацией через pipe справедлив лишь для родственных процессов, имеющих общего прародителя, инициировавшего системный вызов pipe(), или для таких процессов и самого прародителя и не может использоваться для потокового общения с другими процессами. В операционной системе UNIX существует возможность использования pip'a для взаимодействия других процессов, но ее реализация достаточно сложна и лежит далеко за пределами наших занятий.

Для организации потокового взаимодействия любых процессов в операционной системе UNIX применяется средство связи, получившее название FIFO (от First Input First Output) или именованный pipe. FIFO во всем подобен pip6 у, за одним исключением: данные о расположении FIFO в адресном пространстве ядра и его состоянии процессы могут получать не через родственные связи, а через файловую систему. Для этого при создании именованного pip6 а на диске заводится файл специального типа, обращаясь к которому процессы могут получить интересующую их информацию. Для создания FIFO используется системный вызов mknod() или существующая в некоторых версиях UNIX функция mkfifo().

Следует отметить, что при их работе не происходит действительного выделения области адресного пространства операционной системы под именованный pipe, а только заводится файл-метка, существование которой позволяет осуществить реальную организацию FIFO в памяти при его открытии с помощью уже известного нам ситемного вызова open().

После открытия именованный pipe ведет себя точно так же, как и неименованный. Для дальнейшей работы с ним применяются системные вызовы read(), write() и close(). Время существования FIFO в адресном пространстве ядра операционной системы, как и в случае с pip6 ом, не может превышать время жизни последнего из использовавших его процессов. Когда все процессы, работающие с FIFO, закрывают все файловые дескрипторы, ассоциированные с ним, система освобождает ресурсы, выделенные под FIFO. Вся непрочитанная информация теряется. В то же время файл-метка остается на диске и может использоваться для новой реальной организации FIFO в да льнейшем.

```
#include <sys/stat.h>
#include <unistd.h>
int mknod(char *path, int mode, int dev);
```

Нашей целью является не полное описание системного вызова mknod, а только описание его использования для создания FIFO. Поэтому мы будем рассматривать не все возможные варианты задания параметров, а только те из них, которые соответствуют этой специфической деятельности.

Параметр dev является несущественным в нашей ситуации, и мы будем всегда задавать его равным 0. Параметр path является указателем на строку, содержащую полное или относительное имя файла, который будет являться меткой FIFO на диске. Для успешного создания FIFO файла с таким именем перед вызовом существовать не должно. Параметр mode

устанавливает атрибуты прав доступа различных категорий пользователей к FIFO. Этот параметр задается как результат побитовой операции "или" значения S_IFIFO, указывающего, что системный вызов должен создать FIFO, и некоторой суммы следующих восьмеричных значений:

- 0400 0 разрешено чтение для пользователя, создавшего FIFO;
- 0200 0 разрешена запись для пользователя, создавшего FIFO;
- 0040 0 разрешено чтение для группы пользователя, создавшего FIFO;
- 0020 0 разрешена запись для группы пользователя, создавшего FIFO;
- 0004 0 разрешено чтение для всех остальных пользователей;
- 0002 0 разрешена запись для всех остальных пользователей

При создании FIFO реально устанавливаемые права доступа получаются из стандартной комбинации параметра mode и маски создания файлов текущего процесса umask, а именно 0 они равны (0777 & mode) & ~umask.

При успешном создании FIFO системный вызов возвращает значение 0, при неуспешном 0 отрицательное значение.

```
#include <sys/stat.h>
#include <unistd.h>
int mkfifo(char *path, int mode);
```

Функция mkfifo предназначена для создания FIFO в операционной системе. Параметр path является указателем на строку, содержащую полное или относительное имя файла, который будет являться меткой FIFO на диске. Для успешного создания FIFO файла с таким именем перед вызовом функции не должно существовать. Параметр mode устанавливает атрибуты прав доступа различных категорий пользователей к FIFO. Этот параметр задается как некоторая сумма следующих восьмеричных значений:

- 0400 0 разрешено чтение для пользователя, создавшего FIFO;
- 0200 0 разрешена запись для пользователя, создавшего FIFO;
- 0040 0 разрешено чтение для группы пользователя, создавшего FIFO;
- 0020 0 разрешена запись для группы пользователя, создавшего FIFO;
- 0004 0 разрешено чтение для всех остальных пользователей;
- 0002 0 разрешена запись для всех остальных пользователей.

При создании FIFO реально устанавливаемые права доступа получаются из стандартной комбинации параметра mode и маски создания файлов текущего процесса umask, а именно 0 они равны (0777 & mode) & ~umask.

При успешном создании FIFO функция возвращает значение 0, при неуспешном 0 отрицательное значение.

Важно понимать, что файл типа FIFO не служит для размещения на диске информации, которая записывается в именованный pipe. Эта информация располагается внутри адресного пространства операционной системы, а файл является только меткой, создающей предпосылки для ее размещения. Не пытайтесь просмотреть содержимое этого файла с помощью Midnight Commander (mc)!!! Это приведет к его глубокому зависанию!

Системные вызовы read() и write() при работе с FIFO имеют те же особенности поведения, что и при работе с pip6 ом. Системный вызов open() при открытии FIFO также ведет себя несколько иначе, чем при открытии других типов файлов, что связано с возможностью блокирования выполняющих его процессов. Если FIFO открывается только для чтения, и флаг O_NDELAY не задан, то процесс, осуществивший системный вызов, блокируется до тех пор, пока какой-либо другой процесс не откроет FIFO на запись. Если флаг O_NDELAY задан, то возвращается значение файлового дескриптора, ассоциированного с FIFO. Если FIFO открывается только для записи, и флаг O_NDELAY не задан, то процесс, осуществивший системный вызов, блокируется до тех пор, пока какой-либо другой процесс не откроет FIFO на чтение. Если флаг O_NDELAY задан, то констатируется возникновение ошибки и

возвращается значение -1. Задание флага O_NDELAY в параметрах системного вызова open() приводит и к тому, что процессу, открывшему FIFO, запрещается блокировка при выполнении последующих операций чтения из этого потока данных и записи в него.

Обратим внимание, что повторный запуск программы приведет к ошибке при попытке создания FIFO, так как файл с заданным именем уже существует. Здесь нужно либо удалять его перед каждым прогоном программы с диска вручную, либо после первого запуска модифицировать исходный текст, исключив из него все, связанное с системным вызовом mknod().

Сигналы

С точки зрения пользователя получение процессом сигнала выглядит как возникновение прерывания. Процесс прерывает исполнение, и управление передается функции-обработчику сигнала. По окончании обработки сигнала процесс может возобновить регулярное исполнение.

Типы сигналов принято задавать специальными символьными константами. Системный вызов ***kill()*** предназначен для передачи сигнала одному или нескольким специфицированным процессам в рамках полномочий пользователя.

```
#include <sys/types.h>
#include <signal.h>
int kill(pid_t pid, int signal);
```

Послать сигнал (не имея полномочий суперпользователя) можно только процессу, у которого эффективный идентификатор пользователя совпадает с эффективным идентификатором пользователя для процесса, посылающего сигнал. Аргумент ***pid*** указывает процесс, которому посылается сигнал, а аргумент ***sig*** – какой сигнал посылается. В зависимости от значения аргументов:

- ***pid > 0*** сигнал посылается процессу с идентификатором ***pid***;
- ***pid=0*** сигнал посылается всем процессам в группе, к которой принадлежит посылающий процесс;
- ***pid=-1*** и посылающий процесс не является процессом суперпользователя, то сигнал посылается всем процессам в системе, для которых идентификатор пользователя совпадает с эффективным идентификатором пользователя процесса, посылающего сигнал.
- ***pid = -1*** и посылающий процесс является процессом суперпользователя, то сигнал посылается всем процессам в системе, за исключением системных процессов (обычно всем, кроме процессов ***с pid = 0*** и ***pid = 1***).
- ***pid < 0***, но не ***–1***, то сигнал посылается всем процессам из группы, идентификатор которой равен абсолютному значению аргумента ***pid*** (если позволяют привилегии).
- если ***sig = 0***, то производится проверка на ошибку, а сигнал не посылается. Это можно использовать для проверки правильности аргумента ***pid*** (есть ли в системе процесс или группа процессов с соответствующим идентификатором).

Системные вызовы для установки собственного обработчика сигналов:

```
#include <signal.h>
void (*signal (int sig, void (*handler) (int)))(int);
int sigaction(int sig, const struct sigaction *act, struct sigaction *oldact);
```

Структура sigaction имеет следующий формат:

```
struct sigaction {
```

```
void (*sa_handler)(int);
void (*sa_sigaction)(int, siginfo_t *, void *);
sigset_t sa_mask;
int sa_flags;
void (*sa_restorer)(void);
}
```

Системный вызов ***signal*** служит для изменения реакции процесса на какой-либо сигнал. Параметр ***sig*** – это номер сигнала, обработку которого предстоит изменить. Параметр handler описывает новый способ обработки сигнала – это может быть указатель на пользовательскую функцию-обработчик сигнала, специальное значение ***SIG_DFL*** (восстановить реакцию процесса на сигнал ***sig*** по умолчанию) или специальное значение ***SIG_IGN*** (игнорировать поступивший сигнал ***sig***). Системный вызов возвращает указатель на старый способ обработки сигнала, значение которого можно использовать для восстановления старого способа в случае необходимости.

Пример пользовательской обработки сигнала ***SIGUSR1***.

```
void *my_handler(int nsig) { код функции-обработчика сигнала }
int main()
{
(void) signal(SIGUSR1, my_handler);
}
```

Системный вызов ***sigaction*** используется для изменения действий процесса при получении соответствующего сигнала. Параметр ***sig*** задает номер сигнала и может быть равен любому номеру. Если параметр ***act*** не равен нулю, то новое действие, связянное с сигналом ***sig***, устанавливается соответственно ***act***. Если ***oldact*** не равен нулю, то предыдущее действие записывается в ***oldact***.

Задание для выполнения

Ознакомиться с руководством, теоретическими сведениями и лекционным материалом по использованию и функционированию средств взаимодействия.

Написать программу, которая порождает дочерний процесс, и общается с ним через средства взаимодействия согласно варианту, передавая и получая информацию согласно варианту. Передачу и получение информации каждым из процессов сопровождать выводом на экран информации типа "процесс такой-то передал/получил такую-то информацию". **Дочерние процессы начинают операции после получения сигнала SIGUSR1 от родительского процесса.**

Варианты индивидуальных заданий

Вар.	Средство взаимодействия	Задание
1	Программные каналы	Родитель передает потомку три стороны треугольника, потомок возвращает его площадь.
2	Разделяемая память	Родитель передает три строки, потомок возвращает самую длинную из них.
3	Очереди сообщений	Родитель передает 5 случайных чисел, потомок возвращает их сумму и произведение.
4	Общие файлы	Родитель передает потомку две строки S1 и S2, тот возвращает их конкатенацию S2+S1.
5	Именованные каналы	Родитель передает величины катетов прямоугольного треугольника, назад получает величины острых углов.

Вар.	Средство взаимодействия	Задание
6	Программные каналы	Родитель передает три строки, потомок возвращает их отсортировав в лексикографическом порядке.
7	Разделяемая память	Родитель передает потомку три стороны треугольника, потомок возвращает его периметр.
8	Очереди сообщений	Родитель передает три строки, потомок возвращает самую длинную из них.
9	Общие файлы	Родитель передает 5 случайных чисел, потомок возвращает их сумму и произведение.
10	Именованные каналы	Родитель передает потомку две строки S1 и S2, тот возвращает их конкатенацию S2+S1+S2.

ЛАБОРАТОРНАЯ РАБОТА №7. «СЕМАФОРЫ»

Теоретические сведения

Семафоры в ОС Линукс реализованы двух типов: согласно стандарту **System V IPC** и удовлетворяющие стандарту **POSIX-1.2001.**

Семафоры первого типа имеют более широкий функционал по сравнению с классическими семафорами Дейкстры (даже примитивов не два, а больше), однако считаются устаревшими (как и сама System V). Поэтому изучение данного вида семафоров ограничим только вашим знакомством с ними через man (например, посмотрите команды semop, semget и т.д.).

Второй тип семафоров, т.н. POSIX-семафоры, намного проще и намного лучше проработаны. Они реализуют классические семафоры Дейкстры. Именно эти семафоры настоятельно рекомендуется использовать в данной работе. Недостаток у данных семафоров один, но весьма существенный: до версии 2.6 ядра Linux они были реализованы только с поддержкой нескольких нитей одного процесса (были только неименованные семафоры).

Начиная с ядра 2.6 появились именованные семафоры, которые могут использоваться несколькими процессами. Начиная с ядра 2.6, POSIX-семафоры полноценно могут быть вами использованы.

POSIX-семафоры позволяют синхронизировать работу процессам и потокам.

Семафоры

Семафор - целое число, которое не может быть меньше нуля. Две операции могут производиться над семафорами: увеличение семафора на 1 (**sem_post**) и уменьшение на 1 (**sem_wait**). Если значение семафора в данный момент равно 0, то вызов sem_wait переводит процесс (поток) в состояние блокировки до тех пор, пока семафор не станет больше 0.

Начиная с ядра Linux 2.6 POSIX-семафоры существуют двух видов: именованные и неименованные.

Именованные семафоры идентифицируются именем формы somename. Два процесса могут оперировать одним и тем же семафором, передавая его имя в sem_open. Данная функция создает новый именованный семафор либо открывает существующий. В дальнейшем к нему можно применять операции sem_post и sem_wait. Когда процесс завершает использование семафора, тот может быть удален из системы с использованием **sem_unlink** (иначе он будет существовать до завершения работы ОС).

Неименованные семафоры располагаются в области памяти, которая является разделяемой между несколькими потоками одного процесса (a thread-shared semaphore) или несколькими процессами (a process-shared semaphore). Например, в глобальной переменной. Разделяемый между процессами семафор должен располагаться в разделяемой памяти (например, построенной с использованием shm_open). Перед использованием неименованный семафор должен быть проинициализирован с использованием **sem_init**. После с ним также можно оперировать sem_post и sem_wait. Когда он больше не нужен, уничтожается с помощью **sem_destroy**.

Внимание! Программы, использующие POSIX семафоры, должны компилироваться с ключом **-lrt** или **-pthread** , чтобы подключать библиотеку librt.

В файловой системе Линукс именованные семафоры обычно монтируются в /dev/shm, с именем типа sem.name.

Основные системные вызовы по работе с семафорами

```
#include <semaphore.h>
sem_t *sem_open(const char *name, int oflag);
sem_t *sem_open(const char *name, int oflag, mode_t mode, unsigned int value);
```

Описание системного вызова

Открывает и инициализирует именованный семафор с именем ***name***, правами доступа ***mode***, начальным значением ***value***.

Пример:

```
char sem_name1[]="mysemaphore1";
char sem_name2[]="mysemaphore2";
sem_t *s1 = sem_open(sem_name1, O_CREAT);
sem_t s2 = sem_open(sem_name2, O_CREAT, 0644, 10);
```

Открывается семафор с именем mysemaphore1, уже имеющийся в системе, значение и права уже установлены. Открывается семафор с именем mysemaphore2 , правами доступа 0644(rw-r--r--) и начальным значением 10.

```
#include <semaphore.h>
int sem_close(sem_t *sem);
```

Описание системного вызова

Закрывает семафор, но семафор все еще существует в системе в /dev/shm.

Пример:

```
sem_close(s1);
sem_close(&s2);
```

```
#include <semaphore.h>
int sem_post(sem_t *sem);
```

Описание системного вызова

Увеличивает значение семафора на 1, тем самым разблокирует другой процесс, который блокировался на этом семафоре.

Пример:

```
sem_post(s1);
sem_post(&s2);
```

```
#include <semaphore.h>
int sem_wait(sem_t *sem);
int sem_trywait(sem_t *sem);
int sem_timedwait(sem_t *sem, const struct timespec *abs_timeout);
```

Описание системного вызова

Блокирует процесс на семафоре, если тот равен 0, иначе уменьшает его на 1. sem_trywait вместо блокирования процесса при невозможности уменьшения семафора вызывает ошибку. sem_timedwait устанавливает предельное время блокировки.

Пример:
sem_wait(s1);
sem_wait(&s2);

Прототип системного вызова

```
#include <semaphore.h>
int sem_getvalue(sem_t *sem, int *sval);
```

Описание системного вызова

Функция sem_getvalue возвращает текущее значение семафора, помещая его в целочисленную переменную, на которую указывает sval. Если семафор заблокирован, возвращается либо 0, либо отрицательное число, модуль которого соответствует количеству потоков, ожидающих разблокирования семафора.

Пример:

```
int val;
sem_getvalue(sem, &val);
printf("Semaphore value = %d\n", val);
```

Больше информации в man sem_overview и man по соответствующим системным вызовам

Мьютэксы

Мьютекс – примитив синхронизации, обеспечивающий взаимное исключение исполнения критических участков кода. Классический мьютекс отличается от двоичного семафора наличием эксклюзивного владельца, который и должен его освобождать (т.е. переводить в незаблокированное состояние)

Условно классический мьютекс можно представить в виде переменной, которая может находиться в двух состояниях: в заблокированном и в незаблокированном. При входе в свою критическую секцию поток вызывает функцию перевода мьютекса в заблокированное состояние, при этом поток блокируется до освобождения мьютекса, если другой поток уже владеет им. При выходе из критической секции поток вызывает функцию перевода мьютекса в незаблокированное состояние. В случае наличия нескольких заблокированных по мьютексу потоков во время разблокировки планировщик выбирает поток для возобновления выполнения (в зависимости от реализации это может быть, как случайный, так и детерминированный по некоторым критериям поток).

Основные системные вызовы по работе с мьютексами

Прототип системного вызова

```
#include <pthread.h>
int pthread_mutex_init(pthread_mutex_t *mutex, const pthread_mutexattr_t *attr)
```

Создание и инициализация мьютекса.

Пример:

```
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
```

или

```
pthread_mutex_t mutex;
pthread_mutex_init(&mutex, NULL);
```

Прототип системного вызова

```
#include <pthread.h>
pthread_mutex_destroy(pthread_mutex_t *mutex)
```

Уничтожение мьютекса.

Пример:

```
pthread_mutex_destroy(&mutex);
```

Далее системные вызовы имеют идентичный пример использования.

Прототип системного вызова

```
#include <pthread.h>
pthread_mutex_lock(pthread_mutex_t *mutex)
```

Перевод мьютекса в заблокированное состояние (захват мьютекса).

Прототип системного вызова

```
#include <pthread.h>
pthread_mutex_trylock(pthread_mutex_t *mutex)
```

Попытка перевода мьютекса в заблокированное состояние, и возврат ошибки в случае, если должна произойти блокировка потока из-за того, что у мьютекса уже есть владелец.

Прототип системного вызова

```
#include <pthread.h>
pthread_mutex_timedlock(pthread_mutex_t *mutex)
```

Попытка перевода мьютекса в заблокированное состояние, и возврат ошибки в случае, если попытка не удалась до наступления указанного момента времени.

Прототип системного вызова

```
#include <pthread.h>
pthread_mutex_unlock(pthread_mutex_t *mutex)
```

Перевод мьютекса в незаблокированное состояние (освобождение мьютекса).

Пример решения задачи производителя-потребителя с использованием семафоров

Producer.c

```
#include <semaphore.h>
#include <sys/types.h>
```

```
#include <unistd.h>
#include <fcntl.h>
#include <stdio.h>
#include <sys/msg.h>
#include <sys/sem.h>
#include <unistd.h>
#define buf 5

int main()
{
     printf("Producer\n");
     sem_t *full;
     sem_t *empty;
     sem_t *mutex;
     const char *sem_full="full";
     const char *sem_empty="empty";
     const char *sem_mutex="mutex";
     full= sem_open(sem_full,O_CREAT,0777,0);
     empty= sem_open(sem_empty,O_CREAT,0777,buf);
     mutex=sem_open(sem_mutex,O_CREAT,0777,1);
     while(1){
           sem_wait(empty);
           sem_wait(mutex);
           printf("produce_item\n");
           sleep(1);
           sem_post(mutex);
           sem_post(full);
     }
     sem_close(full);
     sem_close(empty);
     sem_close(mutex);
     sem_unlink(sem_full);
     sem_unlink(sem_empty);
     sem_unlink(sem_mutex);
return 0;
}
```

Consumer.c

```
#include <semaphore.h>
#include <sys/types.h>
#include <unistd.h>
#include <fcntl.h>
#include <stdio.h>
#include <sys/msg.h>
#include <sys/sem.h>
#include <unistd.h>
#define buf 5

int main()
{
     printf("Consumer\n");
     sem_t *full;
     sem_t *empty;
     sem_t *mutex;
     const char *sem_full="full";
     const char *sem_empty="empty";
     const char *sem_mutex="mutex";
     full= sem_open(sem_full,O_CREAT,0777,0);
     empty= sem_open(sem_empty,O_CREAT,0777,buf);
     mutex=sem_open(sem_mutex,O_CREAT,0777,1);
     while(1){
           sem_wait(full);
           sem_wait(mutex);
```

```
            printf("consume_item\n");
            sleep(1);
            sem_post(mutex);
            sem_post(empty);
        }
        sem_close(full);
        sem_close(empty);
        sem_close(mutex);
        sem_unlink(sem_full);
        sem_unlink(sem_empty);
        sem_unlink(sem_mutex);
return 0;
}
```

В примере вместо настоящего мьютекса был использован бинарный семафор, в классическом его варианте. Для более верного решения использовать следует pthread_mutex системные вызовы и переменные.

Задание для выполнения

Ознакомиться с руководством, теоретическими сведениями и лекционным материалом по использованию и функционированию средств синхронизации - **семафоров Дейкстры**, и их реализацией в Linux - System V IPC семафоры и POSIX-семафоры.

Написать две (или более) программы, которые, работая параллельно зацикленно, обмениваются информацией согласно варианту. Передачу и получение информации каждым из процессов сопровождать выводом на экран информации типа "процесс такой-то передал/получил такую-то информацию". Синхронизацию работы процессов реализовать с помощью семафоров. Учтите, что при организации совместного доступа к разделяемому ресурсу (например, файлу) вам понадобится применять мьютексы.

Для наглядности запускайте свои процессы в разных окнах терминала. Запустите программы в нескольких экземплярах (одну первую и две/три вторых, две первых и две вторых...).

Варианты индивидуальных заданий

Вариант 1. Первый процесс в цикле ожидает ввода символа с потока stdin, после чего пишет в файл случайное число, каждый раз открывая и закрывая за собой файл. Второй эти числа из файла забирает и выводит на экран.

Вариант 2. Первый процесс семафорами передаёт второму число, а второй умножает его на свой pid и выводит значение на экран. *Внимание: никаких средств взаимодействия (файлов, pipe, ...), кроме семафоров и мьютексов, не использовать.*

Вариант 3. Первый процесс в цикле ожидает ввода символа с потока stdin, после чего передает второму процессу соответствующий символ. Второй получает и выводит его на экран случайное количество раз. *Внимание: никаких средств взаимодействия (файлов, pipe, ...), кроме семафоров и мьютексов, не использовать!*

Вариант 4. Первый процесс в цикле ожидает ввода символа с потока stdin, после чего пишет в файл соответствующий символ, если он является согласной буквой, каждый раз открывая и закрывая за собой файл. Второй процесс забирает из файла символы и выводит на экран.

Вариант 5. Первый процесс в цикле ожидает ввода символа с потока stdin, после чего пишет в файл случайное число, каждый раз открывая и закрывая за собой файл. Второй процесс забирает из файла числа и выводит на экран соответствующее числу количество любых символов.

Вариант 6. Первый процесс в цикле ожидает ввода символа с потока stdin, после чего пишет в файл соответствующий символ случайное количество раз, каждый раз открывая и закрывая за собой файл. Второй процесс забирает из файла символы и выводит на экран их количество.

Вариант 7. Первый процесс пишет в файл строки вида «pid - текущее время», каждый раз открывая и закрывая файл, а второй процесс эти строки читает и выводит, дополняя своим pid.

Вариант 8. Первый процесс в цикле ожидает ввода символа с потока stdin, после чего пишет в файл символ, если он не является ни цифрой, ни буквой, каждый раз открывая и закрывая за собой файл. Второй процесс забирает из файла символы и выводит на экран.

Вариант 9. Первый процесс семафорами передаёт второму текущее время, а второй выводит его в форматированном для человека виде. *Внимание: никаких средств взаимодействия (файлов, pipe, ...), кроме семафоров и мьютексов, не использовать!*

Вариант 10. Первый процесс в цикле ожидает ввода символа с потока stdin, после чего пишет в файл соответствующий символ, каждый раз открывая и закрывая за собой файл. Второй процесс считывает символ из файла и передает его ASCII через семафор первому процессу. *Внимание: никаких средств взаимодействия (файлов, pipe, ...), кроме семафоров и мьютексов, не использовать!*

ЛАБОРАТОРНАЯ РАБОТА №8. «СОЗДАНИЕ СЦЕНАРИЕВ КОМАНДНОЙ СТРОКИ. BASH-СЦЕНАРИИ»

Программирование в BASH-shell

Shell

Командный интерпретатор в среде UNIX выполняет две основные функции:

- представляет интерактивный интерфейс с пользователем, т.е. выдает приглашение, и обрабатывает вводимые пользователем команды;
- обрабатывает и исполняет текстовые файлы, содержащие команды интерпретатора (командные файлы);

В последнем случае, операционная система позволяет рассматривать командные файлы как разновидность исполняемых файлов. Соответственно различают два режима работы интерпретатора: интерактивный и командный.

Существует несколько типов оболочек в мире UNIX. Две главные - это "Bourne shell" и "C shell". Bourne shell (или просто shell) использует командный синтаксис, похожий на первоначально для UNIX. В большинстве UNIX-систем Bourne shell имеет имя /bin/sh (где sh сокращение от ''shell''). C shell использует иной синтаксис, чем-то напоминающий синтаксис языка программирования Си. В большинстве UNIX-систем он имеет имя /bin/csh.

В Linux есть несколько вариаций этих оболочек. Две наиболее часто используемые, это Новый Bourne shell (Bourne Again Shell) или ''Bash'' (/bin/bash) и Tcsh (/bin/tcsh). Bash - это развитие прежнего shell с добавлением многих полезных возможностей, частично содержащихся в C shell.

Поскольку Bash можно рассматривать как надмножество синтаксиса прежнего shell, любая программа, написанная на sh shell должна работать и в Bash. Tcsh является расширенной версией C shell. При входе в систему пользователю загружается командный интерпретатор по умолчанию. Информация о том, какой интерпретатор использовать для конкретного пользователя находится в файле /etc/passwd.

Возможно, вам захочется выполнить сценарий, написанный для одного из shell Linux, в то время как вы работаете в другом. Предположим, вы работаете в TCSH-shell и хотите выполнить написанный в BASH сценарий, содержащий команды этого (второго) shell. Сначала нужно с помощью команды sh перейти в BASH-shell, выполнить сценарий, а затем вернуться в TCSH. Эту процедуру можно автоматизировать, поставив первыми в сценарии символы #! и указав после них путевое имя программы нужного shell в вашей системе.

Shell всегда изучает первые символы сценария и на их основании делает вывод о том, к какому типу shell этот сценарий относится - BASH, PDKSH или TCSH. Если первый символ - пробел, это сценарий BASH-shell или PDKSH-shell. Если первый символ - знак #, это сценарий TCSH-shell. Если первые символы - #!, то shell читает указанное за ними имя программы. После символов #! всегда должно следовать путевое имя программы нужного shell, по которому можно идентифицировать его тип. Если вы запускаете сценарий из shell, отличного от того, который указан в первой строке запускаемого сценария, то будет вызван shell, указанный в первой строке, и в нем выполнится ваш сценарий. В такой ситуации одного пробела или знака 41 для указания того, что это сценарий BASH или TCSH, бывает недостаточно. Такая идентификация работает только в собственных сценариях этих shell. Чтобы обозначить сценарий другого shell, необходимо поставить символы #! и путевое имя.Например, если поставить в начало первой строки сценария hello комбинацию символов #!/bin/sh, то этот сценарий можно будет выполнять непосредственно из TCSH-shell. Сначала сценарий осуществит переход в BASH, выполнит его команды, а затем вернется в TCSH (или в тот shell, из которого он выполнялся). В следующем примере сценарий hello содержит команду #!/bin/sh. Пользователь выполняет сценарий, находясь в TCSH-shell.

Командные файлы

Командный файл в Unix представляет собой обычный текстовый файл, содержащий набор команд Unix и команд Shell. Для того чтобы командный интерпретатор воспринимал этот текстовый файл, как командный необходимо установить атрибут на исполнение.

```
$ echo "ps -af" > commandfile
$ chmod +x commandfile
$ ./commandfile
```

В представленном примере команда echo "ps -af" > commandfile создаст файл с одной строкой "ps -af", команда chmod +x commandfile установит атрибут на исполнение для этого файла, команда ./commandfile осуществит запуск этого файла.

Переменные shell

Имя shell-переменной - это начинающаяся с буквы последовательность букв, цифр и подчеркиваний. Значение shell-переменной - строка символов.

Например: Var = "String" или Var = String

Команда echo $Var выведет на экран содержимое переменной Var т.е. строку "String" , на то что мы выводим содержимое переменной указывает символ "$". Так команда echo Var выведет на экран просто строку "Var" .

Еще один вариант присвоения значения переменной Var = 'набор команд Unix'. Обратные кавычки говорят о том, что сначала должна быть выполнена заключенная в них команда), а результат ее выполнения, вместо выдачи на стандартный выход, приписывается в качестве значения переменной.

```
CurrentDate = 'date'
```

Переменной CurrentDate будет присвоен результат выполнения команды date. Можно присвоить значение переменной и с помощью команды "read", которая обеспечивает прием значения переменной с (клавиатуры) дисплея в диалоговом режиме.

```
echo "Введите число"
read X1
echo "вы ввели -" $X1
```

Несмотря на то, что shell-переменные в общем случае воспринимаются как строки, т. е. "35" - это не число, а строка из двух символов "3" и "5", в ряде случаев они могут интерпретироваться иначе, например, как целые числа.

Разнообразные возможности имеет команда "expr".

```
x=7
y=2

rez=expr $x + $y
echo результат=$rez
```

выдаст на экран результат=9

Параметры командного файла

В командный файл могут быть переданы параметры. В shell используются позиционные параметры (т.е. существенна очередность их следования). В командном файле соответствующие параметрам переменные (аналогично shell-переменным) начинаются с символа "$", а далее следует одна из цифр от 0 до 9: При обращении к параметрам перед цифрой ставится символ доллара "$" (как и при обращении к переменным):

$0 соответствует имени данного командного файла;

$1 первый по порядку параметр;

$2 второй параметр и т.д.

Поскольку число переменных, в которые могут передаваться параметры, ограничено одной цифрой, т.е. 9-ю ("0", как уже отмечалось имеет особый смысл), то для передачи большего числа параметров используется специальная команда "shift".

Команда "set" устанавливает значения параметров. Это бывает очень удобно. Например, команда "date" выдает на экран текущую дату, скажем, "Mon May 01 12:15:10 2002", состоящую из пяти слов, тогда set `dateècho $1 $3 $5 выдаст на экран Mon 01 2002

Управляющие структуры

С помощью управляющих структур пользователь может осуществлять контроль над выполнением Linux-команд в программе. Управляющие структуры позволяют повторять команды и выбирать для выполнения команды, необходимые в конкретной ситуации. Управляющая структура состоит из двух компонентов: операции проверки и команд. Если проверка считается успешной, то выполняются команды. Таким образом, с помощью управляющих структур можно принимать решения о том, какие команды следует выполнять.

Существует два вида управляющих структур: циклы и условия. Цикл используется для повторения команд, тогда как условие обеспечивает выполнение команды при соблюдении некоторых критериев. В BASH-shell используются три вида циклических управляющих структур (while, for и for-in) и две условные управляющие структуры (if и case).

Управляющие структуры while и if - это структуры общего назначения. Они используются, например, для выполнения итераций и принятия решений на основании различных проверок. Управляющие структуры case и for - более специализированные. Структура case представляет собой модифицированную форму условия if и часто используется для построения меню. Структура for - это цикл ограниченного типа. Здесь обрабатывается список значений, и на каждой итерации цикла переменной присваивается новое значение.

В управляющих структурах if и while проверка построена на выполнении Linux-команды. Все команды ОС Linux после выполнения выдают код завершения. Если команда выполнена нормально, выдается код 0. Если по какой-либо причине команда не выполняется, то выдается положительное число, обозначающее тип отказа. Управляющие структуры if и while проверяют код завершения Linux-команды. Если код - 0, выполняются действия одного типа, если нет - другого.

Задание для выполнения

Варианты индивидуальных заданий

1. Реализовать командный файл, реализующий меню из трех пунктов (в цикле):
   1) ввести пользователя и вывести на экран все процессы, запущенные данным пользователем;
   2) показать всех пользователей, в настоящий момент, находящихся в системе;
   3) завершение.
2. Реализовать командный файл, реализующий меню из трех пунктов (в цикле):
   1) вывести всех пользователей, в настоящее время, работающих в системе;
   2) послать сообщение пользователю, имя пользователя, терминал и сообщение вводятся с клавиатуры;
   3) завершение.
3. Реализовать командный файл, реализующий меню из трех пунктов (в цикле):
   1) показать все процессы пользователя, запустившего данный командный файл;
   2) послать сигнал завершения процессу текущего пользователя (ввести PID процесса);

3) завершение.

4. Реализовать командный файл, реализующий меню из трех пунктов (в цикле):

1) определить количество запущенных данным пользователем процессов bash (предусмотреть ввод имени пользователя);

2) завершить все процессы bash данного пользователя.

3) завершение.

5. Реализовать меню из трех пунктов (в цикле):

1) поиск файла в каталоге. <Имя файла> и <Имя каталога> вводятся пользователем с клавиатуры;

2) копирование одного файла в другой каталог. <Имя файла> и <Имя каталога> вводятся пользователем с клавиатуры;

3) завершение.

6. Написать командный файл который в цикле по нажатию клавиши выводит информацию о системе, активных пользователях в системе, а для введенного имени пользователя выводит список активных процессов данного пользователя.

7. Реализовать командный файл который при старте выводит информацию о системе, информацию о пользователе, запустившем данный командный файл, далее в цикле выводит список активных пользователей в системе – запрашивает имя пользователя и выводят список всех процессов bash запущенных данным пользователем.

8. Реализовать командный файл, реализующий символьное меню (в цикле):

1) вывод полной информации о файлах каталога: Ввести имя каталога для отображения;

2) изменить атрибуты файла: файл и атрибуты вводятся с клавиатуры по запросу. После изменения атрибутов вывести на экран расширенный список файлов для проверки установленных атрибутов;

3) выход.

При старте командный файл выводит информацию об имени компьютера, IP- адреса и список всех пользователей зарегистрированных в данный момент на компьютере.

9.Реализовать командный файл, реализующий символьное меню (в цикле):

1) вывод полной информации о файлах каталога: Ввести имя каталога для отображения;

2) создать командный файл: файл вводится с клавиатуры по запросу, далее изменяются атрибут файла на исполнение, затем вводится с клавиатуры строка, которую будет исполнять командный файл. После изменения атрибутов вывести на экран расширенный список файлов для проверки установленных атрибутов и запустить созданный командный файл;

3) завершение.

10. Реализовать командный файл, позволяющий в цикле посылать всем активным пользователям сообщение – сообщение вводится с клавиатуры. Командный файл при старте выводит имя компьютера, имя запустившего командный файл пользователя, тип операционной системы, IP-адрес машины.

ЛАБОРАТОРНАЯ РАБОТА №9 «ПРОЕКТИРОВАНИЕ ГЕТЕРОГЕННОЙ КОМПЬЮТЕРНОЙ СЕТИ.

Диаграмма: топология сети.

Таблица сетевых адресов.

Device	Interface	IP Address	Mask	Default Gateway
R1	**Fa0/0**	192.168.1.1	255.255.255.0	N/A
	S0/1/0	192.168.2.1	255.255.255.0	N/A
R2	**Fa0/0**	192.168.3.1	255.255.255.0	N/A
	S0/1/0	192.168.2.2	255.255.255.0	N/A
PC1	**N/A**	192.168.1.10	255.255.255.0	192.168.1.1
PC2	**N/A**	192.168.3.10	255.255.255.0	192.168.3.1

Цель работы.
Создать (собрать и сконфигурировать) изображённую на диаграмме сеть. Настроить сетевые адреса устройств в соответствии с таблицей сетевых адресов. Произвести начальную конфигурацию маршрутизаторов. С помощью команды show и утилиты ping удостовериться, что устройства функционирую правильно.

Этапы выполнения работы.
1. Произведите начальную конфигурацию маршрутизатора R1. 1.1. Двойным щелчком левой кнопки мыши откройте мену конфигурации маршрутизатора.
1.2. Перейдите на вкладку CLI (см. рисунок 1).

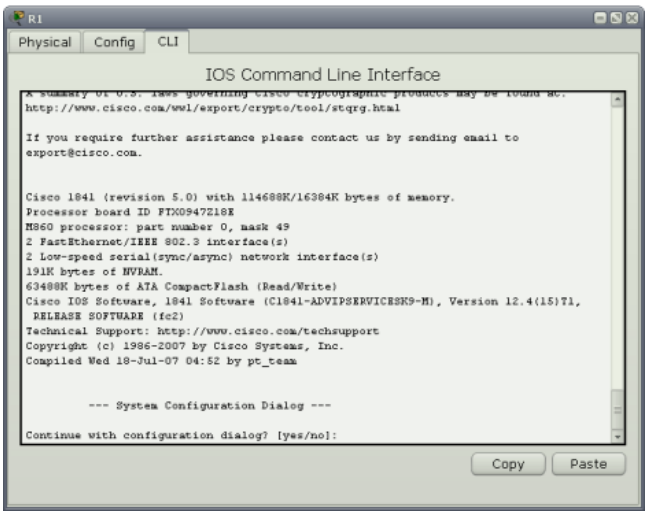


Рисунок 1 – Вкладка CLI

1.3. В появившемся окне, на вопрос **“Continue with configuration dialog? [yes/no]”** ответьте нет. Для этого необходимо напечатать **“no”** и нажать **Enter.**

Рисунок 2 – Строка запроса ответа

1.4. Зайдите в режим **“privileged EXEC”.**
Router>**enable**
Router#
1.5. Зайдите в режим глобальной конфигурации маршрутизатора.
Router#**configure terminal**
Enter configuration commands, one per line. End with CNTL/Z. Router(config)#
1.6. Сконфигурируйте имя маршрутизатора.
Router(config)#**hostname R1**
R1(config)#
1.7. Отключите **DNS lookup.**
R1(config)#**no ip domain-lookup** R1(config)#
1.8. Сконфигурируйте пароль для режима **“EXEC mode”.**
R1(config)#**enable secret** *_пароль_*
R1(config)#
1.9. Сконфигурируйте баннер.
R1(config)#**banner motd &** *_текст_* **&**
R1(config)#
1.10. Сконфигурируйте пароль, который нужно будет вводить при подключении к устройству через консоль.
R1(config)#**line console 0**
R1(config-line)#**password** *_пароль_*
R1(config-line)#**login**
R1(config-line)#**exit**
R1(config)#
1.11. Сконфигурируйте интерфейс **FastEthernet0/0** в соответствии со схемой адресации сети.
R1(config)#**interface fastethernet 0/0**
R1(config-if)#**ip address 192.168.1.1 255.255.255.0**
R1(config-if)#**no shutdown**
%LINK-5-CHANGED: Interface FastEthernet0/0, changed state to up %LINEPROTO-5-UPDOWN: Line protocol on Interface FastEthernet0/0, changed state to up
R1(config-if)#
1.12. Сконфигурируйте интерфейс **Serial0/1/0** в соответствии со схемой адресации сети. Команда **clock rate** используется для синхронизации устройств при WAN-соединениях.
R1(config-if)#**interface serial 0/1/0**
R1(config-if)#**ip address 192.168.2.1 255.255.255.0**
R1(config-if)#**clock rate 64000**
R1(config-if)#**no shutdown**
R1(config-if)#
Серийный интерфейс не активируется до тех пор, пока не будет сконфигурирован и активирован интерфейс на другой стороне. В данном случае – серийный интерфейс на маршрутизаторе R2
1.13. Вернитесь в режим **“privileged EXEC”.**
Use the **end** command to return to privileged EXEC mode.

R1(config-if)#**end**
R1#
1.14. Сохраните настройки на маршрутизаторе **R1.**
R1#**copy running-config startup-config**
Building configuration... [OK]
R1#

2. Произведите начальную конфигурацию маршрутизатора R2 2.1. Для маршрутизатора R2 повторите пункты 1.1 – 1.7
2.2. Сконфигурируйте интерфейс **Serial0/1/0** в соответствии со схемой адресации сети.
R2(config)#**interface serial 0/1/0**
R2(config-if)#**ip address 192.168.2.2 255.255.255.0**
R2(config-if)#**no shutdown**
%LINK-5-CHANGED: Interface Serial0/0/0, changed state to up %LINEPROTO-5-UPDOWN: Line protocol on Interface Serial0/0/0, changed state to up
R2(config-if)#
2.3. Сконфигурируйте интерфейс **FastEthernet0/0** в соответствии со схемой адресации сети.
R2(config-if)#**interface fastethernet 0/0**
R2(config-if)#**ip address 192.168.3.1 255.255.255.0**
R2(config-if)#**no shutdown**
%LINK-5-CHANGED: Interface FastEthernet0/0, changed state to up %LINEPROTO-5-UPDOWN: Line protocol on Interface FastEthernet0/0, changed state to up
R2(config-if)#
2.4. Вернитесь в режим **“privileged EXEC”.**
Use the **end** command to return to privileged EXEC mode.
R1(config-if)#**end**
R1#
2.5. Сохраните настройки на маршрутизаторе **R2.**
R1#**copy running-config startup-config**
Building configuration... [OK]
R1#
4
3. Сконфигурируйте сетевые настройки на конечных устройствах. 3.1. Двойным щелчком левой кнопки мыши откройте мену конфигурации PC1.
3.2. Перейдите на вкладку Desktop (см. рисунок 3).

Рисунок 3 – Вкладка Desktop

3.3. Нажмите на кнопку **IP configuration** и занесите необходимые параметры.

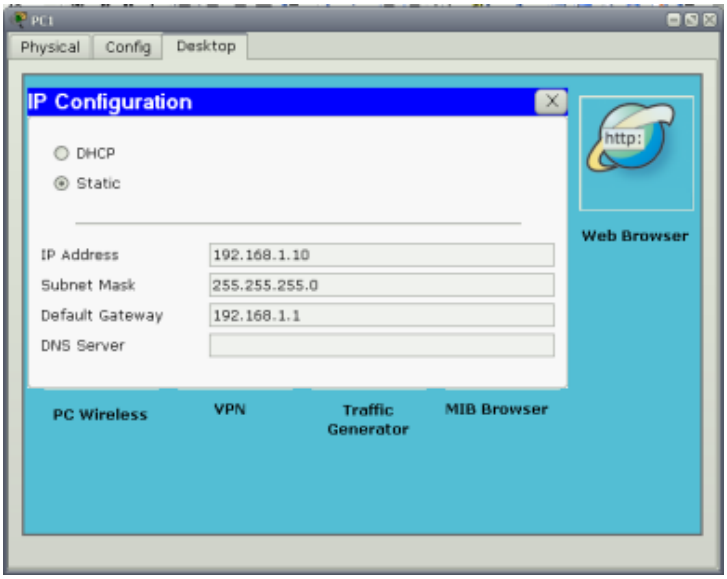


Рисунок 4 – Ввод параметров сети

3.4. Повторите пункты **3.1 – 3.3** для **PC2.**

4. Проверка и тестирование сети.

4.1. С помощью команды **show ip route** убедитесь, что в таблицах маршрутизации присутствуют сети, в которых находятся интерфейсы маршрутизатора.

Вывод команды **show ip route** должен выглядеть следующим образом (см. рисунок 5):

```
R1#show ip route
Codes: C - connected, S - static, R - RIP, M - mobile, B - BGP
       D - EIGRP, EX - EIGRP external, O - OSPF, IA - OSPF inter area
       N1 - OSPF NSSA external type 1, N2 - OSPF NSSA external type 2
       E1 - OSPF external type 1, E2 - OSPF external type 2
       i - IS-IS, su - IS-IS summary, L1 - IS-IS level-1, L2 - IS-IS level-2
       ia - IS-IS inter area, * - candidate default, U - per-user static route
       o - ODR, P - periodic downloaded static route

Gateway of last resort is not set

C    192.168.1.0/24 is directly connected, FastEthernet0/0
C    192.168.2.0/24 is directly connected, Serial0/0/0

----------------------

R2#show ip route
Codes: C - connected, S - static, R - RIP, M - mobile, B - BGP
       D - EIGRP, EX - EIGRP external, O - OSPF, IA - OSPF inter area
       N1 - OSPF NSSA external type 1, N2 - OSPF NSSA external type 2
       E1 - OSPF external type 1, E2 - OSPF external type 2
       i - IS-IS, su - IS-IS summary, L1 - IS-IS level-1, L2 - IS-IS level-2
       ia - IS-IS inter area, * - candidate default, U - per-user static route
       o - ODR, P - periodic downloaded static route

Gateway of last resort is not set

C    192.168.2.0/24 is directly connected, Serial0/0/0
C    192.168.3.0/24 is directly connected, FastEthernet0/0
```

Рисунок 5 – Вывод команды **show ip route**

4.2. С помощью команды **show ip interface brief** убедитесь, что интерфейсы маршрутизатора настроены и активизированы.
Вывод команды **show ip interface brief** должен выглядеть следующим образом (см. рисунок 6):

```
R1#show ip interface brief
Interface              IP-Address      OK? Method Status                Protocol
FastEthernet0/0        192.168.1.1     YES manual up                    up
FastEthernet0/1        unassigned      YES unset  administratively down down
Serial0/0/0              192.168.2.1     YES manual up                    up
Serial0/0/1              unassigned      YES unset  administratively down down
Vlan1                    unassigned      YES manual administratively down down

R2#show ip interface brief
Interface              IP-Address      OK? Method Status                Protocol
FastEthernet0/0        192.168.3.1     YES manual up                    up
FastEthernet0/1        unassigned      YES unset  administratively down down
Serial0/0/0              192.168.2.2     YES manual up                    up
Serial0/0/1              unassigned      YES unset  down                  down
Vlan1                    unassigned      YES manual administratively down down
```

Рисунок 6 – Вывод команды **show ip interface brief**

4.3. С помощью утилиты **ping** проверьте доступность устройств в сети.
Чтобы запустить утилиту **ping** на конечном устройстве (на PC) необходимо: На вкладке **Desktop** нажать на кнопку **Command Prompt (**эмулятор **CMD) (см. рисунок 7)**.

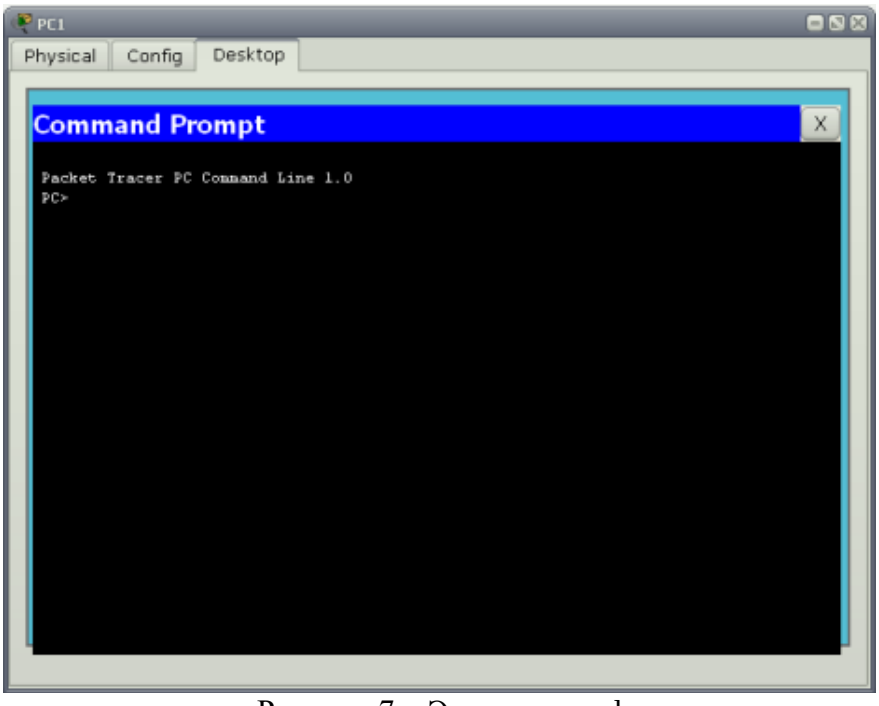


Рисунок 7 – Эмулятор cmd

Используя утилиту **ping**, ответьте на следующие вопросы:
1. С PC1 возможно пропинговать маршрутизатор R1? Если да, то какой из интерфейсов маршрутизатора?
2. С PC2 возможно пропинговать маршрутизатор R2? Если да, то какой из интерфейсов маршрутизатора?
3. С PC2 возможно пропинговать PC1?

ЛАБОРАТОРНАЯ РАБОТА №10 «СЕТЕВАЯ ИНФРАСТРУКТУРА «УМНОГО» ДОМА».

Цель работы:

приобрести практические навыки проектирования инфраструктуры «умного дома», научиться основам программирования микроконтроллерных устройств

Необходимое ПО:

Для выполнения лабораторной работы необходимо установить программу Cisco Packet Tracer версии не ниже 7.1. В процессе установки программы может потребоваться пройти регистрацию на официальном сайте компании Cisco.

Требования к оформлению отчета:

Отчет по лабораторной работе должен содержать следующие разделы (примеры оформления отчетов можно найти в папке с заданиями):

1) Изложение цели работы.
2) Задание по лабораторной работе с описанием своего варианта.
3) Спецификации ввода-вывода программы.
4) Текст программы (кратко).
5) Выводы по проделанной работе.

Задание 1

Последовательно выполните задания, используя режим реального времени CPT. В качестве результата должна быть получена работающая сетевая инфраструктура, изображенная на рис. 1.

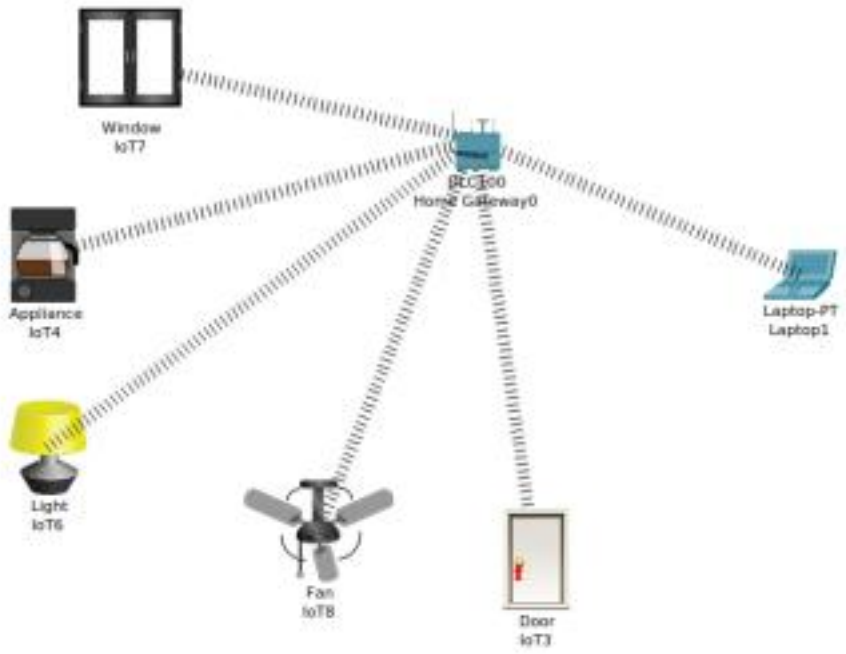


Рисунок 1 - Схема подключения устройств «умного дома»

1) Все необходимые устройства могут быть найдены во вкладках End Devices → End Devices, End Devices → Home и Network Devices → Wireless Devices. Ключевое устройство Home Gateway. Именно оно объединяет все устройства умного дома и клиентские терминалы (такие, как лэптоп) в общую беспроводную сеть. Это сервер IoT.

2) После размещения всех необходимых устройств в рабочей области откройте Home Gateway и во вкладке Config → Interface → Wireless определите тип аутентификации как WPA2-PSK и задайте любой пароль из 8 символом (например, *cisco123*).

3) После настройки сервера, переходим на любое устройство IoT и открываем расширенные настройки (Advanced). Дело в том, что эти устройства по умолчанию не поддерживают беспроводную передачу данных. Откройте вкладку I/O Config. Далее в списке Network Adapter2 выберите беспроводной адаптер PT-IOT-NM-1W.

4) После выполнения предыдущего действия во вкладке Config появится беспроводной интерфейс Wireless3. Откройте его и настройте подключение к серверу, задав правильный тип аутентификации, пароль и выбрав вариант DHCP в IP Configuration (этот вариант чаще всего задан по умолчанию, убедитесь в этом случае, что узлом получен IP-адрес из того же диапазона, что и IP-адрес сервера – как правило, из *192.168.25.0*). В данном случае сервер IoT Home Gateway является DHCP-сервером для подключаемых устройств (автоматически раздает IP-адреса).

5) Далее откройте Settings (там же, во вкладке Config) и поставьте в группе IoT Server переключатель в положение Home Gateway.

6) После выполнения всех этих действий, убедитесь, что между сервером и настраиваемым узлом появилось отображение беспроводной связи.

7) Проделайте действия 3-6 для других устройств, исключая лэптоп.

8) Откройте лэптоп и изучите его физическую конфигурацию. Вы можете заметить, что на нем также, как и на IoT-устройствах не установлен модуль беспроводной связи. Это можно исправить следующим образом: извлеките установленный Fast Ethernet-модуль (предварительно выключив лэптоп) и поместите в свободный слот модуль PT-LAPTOP-NM-1W. После этого включите устройство и произведите похожие настройки беспроводного интерфейса (укажите SSID, тип аутентификации и пароль). Между сервером и лэптопом должна появиться визуализация беспроводной связи.

9) Откройте вкладку Desktop лэптопа и далее IoT Monitor. Нажмите Ok в окне авторизации на сервере, убедившись в правильности написанного IP-адреса сервера. После этого перед вами должен появиться список всех беспроводных устройств, подключенных к нашему серверу. Поэкспериментируйте с кнопками включения/выключения устройств и изучите изменения, которые с ними происходят.

10) Добавьте фон для построенной инфраструктуры, воспользовавшись предложенными (папка background) или использовав свой (рис. 2).

Рисунок 2 - Фон для сетевой инфраструктуры

Задание 2

В первом задании, несмотря на наличие IoT-устройств, сформирована лишь сетевая инфраструк тура, но не полноценное IoT-решение. Это так, поскольку все устройства контролируются (пусть и удаленно), но человеком. Т.е. человек принимает решения о включении/выключении устройств, а не сама система. Попробуем создать решение, которое будет обладать определенной автономностью.

Для этого воспользуемся микроконтроллерными устройствами, которые будут принимать решение о активации тех или иных узлов системы. Спроектируем систему для поддержания комфортной температуры внутри помещения, изображенную на рис. 3

1) Для начала добавьте микроконтроллерную плату в рабочую область (вкладка Components → Boards). Выберите из предложенных плату SBC Board.

2) Откройте добавленную плату на вкладке Programming. Далее в списке слева выберите пункт Blink (Python) и далее скрипт main.py. Программирование для такой платы производится на языке Python. Он является достаточно простым скриптовым языком с большим количеством разработанных библиотек (подробнее о языке можно почитать в предложенной презентации). Скрипт, который откроется, нужен для решения простой задачи – он включает и выключает пин (разъем) на нашей плате, активируя подключенную к нему нагрузку. В качестве такой нагрузки может выступать светодиоды, разные датчики, LCD-экраны и т.д.

3) Попробуйте добавить светодиод (LED) с вкладки Components → Actuators к рабочей области. Затем во вкладке Connections выберите тип соединения IoT Custom Cable и соедините пин D1 вашей платы с пином D0 светодиода. Запустите программу, нажав на кнопку Run. Вы должны увидеть мигающий светодиод. Откройте программу, попытайтесь изучить и понять ее содержимое.

Команда pinMode нужна для определения режима, в котором будет работать наш пин платы (это может быть IN или OUT – для выходных и входных сигналов соответственно).

Как следует из программы, мы делаем пин D1 (или просто пин с номером 1) выходным, для того, чтобы регулировать уровень напряжения и включать и выключать его. Пины бывают цифровыми (D) и аналоговыми (A). Цифровые пины оперируют 0 и 1 (или LOW и HIGH) и лучше всего описывают взаимодействие с устройствами, которые нужно включать и выключать. Аналоговые пины нужны для передачи какой-то многоуровневой информации (например, уровня температуры и влажности).

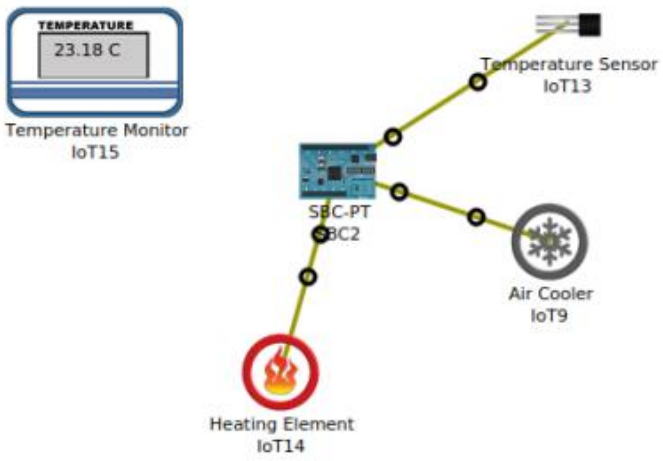


Рисунок 3 - Микроконтроллерная схема

Как вы видите, в программе мы записываем попеременно высокий и низкий сигнал в пин номер 1, что приводит к миганию светодиода (это делается с помощью функции digitalWrite с указанием номера пина и уровня сигнала). Функция delay вызывает задержку перед выполнением следующей команды на указанное количество миллисекунд.

4) Удалите LED из рабочей области. Добавьте другие компоненты, необходимые для реализации проекта (вкладка Actuators), а также цифровой термометр для отслеживания температуры (End Devices → Home→ Temperature Monitor)). Температурный сенсор находится на вкладке (Components → Sensors→ Temperature Sensor).

5) Heating Element нужен для повышения температуры, Air Cooler для понижения. О характеристиках этих устройств можно почитать, кликнув по ним. Для нас важно то, что они включаются и выключаются как цифровые устройства (т.е. вызовом команды digitalWrite). Temperature Monitor нужен для считывания данных о температуре.

Это аналоговый датчик, поэтому для считывания данных применяется функция analogRead с указанием единственного параметра – номера пина. Подсоедините все указанные датчики к плате, выбрав произвольные пины (запомните свой выбор). Для Temperature Monitor выберите пин A0 на нем.

6) Далее изучите изменение температуры в течение суток с помощью показателей температурного монитора. В CPT можно изменять текущее время суток (это делается нажатием на кнопку с «текущим» временем или Shift + E. Как вы заметите, температура изменяется. Хотелось бы, чтобы она оставалась в определенном заданном интервале (например, от 20 до 25 градусов).

7) Итак, мы подошли к самому главному. Теперь вам нужно написать программу, которая будет поддерживать текущую температуру в заданном интервале. Используйте пины, активируя устройства для обогрева и охлаждения на основании данных, считанных с температурного датчика. Имейте в виду, что датчик возвращает данные в интервале от 0 до 1023, соответствующие температуре -100 до 100 градусов. Используйте следующую формулу для получения значения температуры:

$$t_{celsius} = \frac{t_{sensor}}{1023} \times 200 - 100$$

Функция float нужна для конвертации в вещественный тип.

Замечания:

Для сдачи работы необходимо:

1) самостоятельно выполнить работу, изучив необходимую теорию; 2) представить отчет по лабораторной работе;

3) ответить на вопросы преподавателя по теме работы.

Руководства и ресурсы

1) Cisco Packet Tracer: https://www.netacad.com/portal/resources/packet-tracer 5

ЛАБОРАТОРНАЯ РАБОТА №11 «ЛЕКСИЧЕСКИЙ АНАЛИЗАТОР».

Цель работы:

изучить основные понятия теории регулярных языков и грамматик, ознакомиться с назначени ем и принципами работы конечных автоматов (КА), получить практические навыки построения КА на основе заданной регулярной грамматики.

Требования к оформлению отчета

Отчет по лабораторной работе должен содержать следующие разделы:

• Краткое изложение цели работы.

• Задание по лабораторной работе с описанием своего варианта задания.

• Описание регулярной грамматики лексем входного языка в форме Бэкуса-Наура или в форме регулярных выражений (в соответствии с вариантом задания).

• Описание КА или граф переходов КА для распознавания лексем (в соответствии с вариантом задания).

• Выводы по проделанной работе.

Варианты заданий

Любые лексемы, не предусмотренные вариантом задания, встречающиеся в исходном тексте, должны трактоваться как ошибочные.

1) Входной язык содержит логические выражения, разделенные символом ; (точка с запятой). Логические выражения состоят из идентификаторов, констант true («истина») и false («ложь»), знака присваивания (:=), знаков операций or, xor, and, not и круглых скобок.

2) Входной язык содержит операторы условия типа if . . . then . . . else и if . . . then, разделенные символом ; (точка с запятой). Операторы условия содержат идентификаторы, знаки сравне ния <, >, =, десятичные числа, знак присваивания (:=).

3) Входной язык содержит операторы цикла типа for (. . . ; . . . ; . . .) do, разделенные символом ; (точ ка с запятой). Операторы цикла содержат идентификаторы, знаки сравнения<, >, =, десятич ные числа, знак присваивания (:=).

4) Входной язык содержит операторы цикла типа while . . . do . . . , разделенные символом ; (точ ка с запятой). Операторы цикла содержат идентификаторы, знаки сравнения <,>, =, знаки операций +, -, десятичные числа, знак присваивания (:=).

5) Входной язык содержит логические выражения, разделенные символом ; (точка с запятой). Логические выражения состоят из идентификаторов, констант 0 и 1, знака присваивания (:=), знаков операций or, xor, and, not и круглых скобок.

6) Входной язык содержит операторы условия типа if . . . then . . . else и if . . . then, разделенные символом ; (точка с запятой). Операторы условия содержат идентификаторы, знаки сравне ния <, >, =, восьмеричные числа, знак присваивания (:=).

7) Входной язык содержит операторы цикла типа for (. . . ; . . . ; . . .) do, разделенные символом ; (точ ка с запятой). Операторы цикла содержат идентификаторы, знаки сравнения <, >, =, восьме ричные числа, знак присваивания (:=).

8) Входной язык содержит операторы цикла типа while . . . do . . . , разделенные символом ; (точ ка с запятой). Операторы цикла содержат идентификаторы, знаки сравнения <,>, =, знаки операций +, -, восьмеричные числа, знак присваивания (:=).

9) Входной язык содержит логические выражения, разделенные символом ; (точка с запятой). Логические выражения состоят из идентификаторов, символьных констант ‘T’ («истина») и ‘F’ («ложь»), знака присваивания (:=), знаков операций or, xor, and, not и круглых скобок.

10) Входной язык содержит операторы условия типа if . . . then . . . else и if . . . then, разделенные символом ; (точка с запятой). Операторы условия содержат идентификаторы, знаки сравне ния <, >, =, шестнадцатеричные числа, знак присваивания (:=).

11) Входной язык содержит операторы цикла типа for (. . . ; . . . ; . . .) do, разделенные символом ; (точ ка с запятой). Операторы цикла содержат идентификаторы, знаки сравнения <, >, =, шестна дцатеричные числа, знак присваивания (:=).

12) Входной язык содержит операторы цикла типа while . . . do . . . , разделенные символом ; (точ ка с запятой). Операторы цикла содержат идентификаторы, знаки сравнения <,>, =, знаки операций +, -, шестнадцатеричные числа, знак присваивания (:=).

***Рекомендации*:**

1) Идентификатором считается любая последовательность букв и цифр, начинающаяся с бук вы (часто в идентификаторах допускается также знак подчеркивания – «_», иногда и другие символы). Для определения идентификаторов рекомендуется использовать правило граммати ки: $I \rightarrow$ б | *I*б | *I*ц, где «б» обозначает любую букву, а «ц» – любую цифру.

2) Для упрощения структуры КА ключевые слова предлагается считать идентификаторами, а уже после их выделения выполнять проверку на совпадение найденного идентификатора с заданным перечнем ключевых слов.

Необходимые теоретические сведения

Функции лексического анализатора

Лексический анализатор (или сканер) – это часть компилятора, которая читает литеры программы на исходном языке и строит из них слова (лексемы). На вход лексического анализатора поступает текст исходной программы, а выходная информация передается для дальнейшей обработ ки компилятором на этапе синтаксического анализа.

Лексема (лексическая единица языка) – это структурная единица языка, которая состоит из элементарных символов языка и не содержит в своем составе других структурных единиц. Результатом работы лексического анализатора является перечень всех найденных в тексте исход ной программы лексем. Этот перечень лексем можно представить в виде таблицы, называемой *таб лицей лексем*. Каждой лексеме в таблице лексем соответствует некий уникальный условный код, зависящий от типа лексемы, и дополнительная служебная информация. Кроме того, информация о некоторых типах лексем, найденных в исходной программе, должна помещаться в таблицу иденти фикаторов (или в одну из таблиц идентификаторов, если компилятор предусматривает различные таблицы идентификаторов для различных типов лексем).

С теоретической точки зрения лексический анализатор не является обязательной частью ком пилятора. Его функции могут выполняться на этапе синтаксического анализа. Однако существует несколько причин, исходя из которых в состав практически всех компиляторов включают лексиче ский анализ. Эти причины заключаются в следующем:

• упрощается работа с текстом исходной программы на этапе синтаксического разбора и сокра щается объем обрабатываемой информации, так как лексический анализатор структурирует поступающий на вход исходный текст программы и выкидывает всю незначащую информа цию;

• для выделения в тексте и разбора лексем возможно применять простую, эффективную и тео ретически хорошо проработанную технику анализа, в то время как на этапе синтаксического анализа конструкций исходного языка используются достаточно сложные алгоритмы разбора;

• сканер отделяет сложный по конструкции синтаксический анализатор от работы непосредствен но с текстом исходный программы, структура которого может варьироваться в зависимости от версии входного языка – при такой конструкции компилятора при переходе от одной версии языка к другой достаточно только перестроить относительно простой сканер.

Функции, выполняемые лексическим анализатором, и состав лексем, которые он выделяет в тек сте исходной программы, могут меняться в зависимости от версии компилятора. В основном лекси ческие анализаторы выполняют исключение из текста исходной программы комментариев и незна чащих пробелов, а также выделение лексем следующих типов: идентификаторов, строковых, сим вольных и числовых констант, знаков операций, разделителей, а также ключевых (служебных) слов входного языка.

Лексический анализатор имеет дело с таким объектами, как различного рода константы и иден тификаторы (к последним относятся и ключевые слова). Язык констант и идентификаторов в боль шинстве случаев является *регулярным* – то есть, может быть описан с помощью регулярных (пра волинейных или леволинейных) грамматик. Распознавателями для регулярных языков являются конечные автоматы (КА). Существуют правила, с помощью которых для любой регулярной грам матики может быть построен недетерминированный КА, распознающий цепочки языка, заданного этой грамматикой.

Конечные автоматы: определение

Как уже было сказано выше, лексемы могут быть описаны с помощью регулярных языков. Рас познавателями для регулярных языков являются конечные автоматы (КА). Поэтому КА лежат в основе лексического анализа.

КА задается пятеркой:

$M = (Q, \Sigma, \delta, q_0, F)$,

где Q – конечное множество состояний автомата; Σ – конечное множество допустимых входных символов; δ – заданное отображение множества $Q * \Sigma$ во множество подмножеств $P(Q)$: $\delta : Q * \Sigma \rightarrow P(Q)$ (иногда δ называют функцией переходов автомата, которая может быть определена как: $\delta(q, a) = D$, где $q \in Q$ – текущее состояние КА, $a \in \Sigma$ – текущий входной символ, а $D \subseteq Q$ – множество возможных следующих состояний КА); $q_0 \in Q$ – начальное состояние автомата; $F \subseteq Q$ – множество заключительных состояний автомата.

Работа КА выполняется по тактам. На каждом очередном такте i автомат, находясь в некотором состоянии $q_i \in Q$ считывает очередной символ $w \in \Sigma$ из входной цепочки символов и изменяет свое состояние на $q_{i+1} \in \delta(q_i, w)$, после чего указатель в цепочке входных символов передвигается на следующий символ и начинается такт $i + 1$. Так продолжается до тех пор, пока цепочка входных символов не закончится, либо КА попадёт в состояние, из которого нет переходов. Конец цепоч ки символов часто помечают особым символом $\perp$. Считается также, что перед тактом 1 автомат находится в начальном состоянии q_0.

Говорят, что КА допускает цепочку $\alpha \in \Sigma^*$, если в результате выполнения всех тактов работы над этой цепочкой он может оказаться в состоянии $q \in F$. Язык, определяемый автоматом, является множеством всех цепочек, допускаемых данным автоматом.

Графически КА отображается нагруженным однонаправленным графом, в котором вершины представляют состояния КА, дуги отображают переходы из одного состояния в другое, а символы нагрузки (пометки) дуг соответствуют входным символам функции перехода. Если функция пере хода предусматривает переход из состояния q в q^0 по нескольким символам, то между ними строится одна дуга, которая помечается всеми символами, по которым происходит переход из q в q^0. Такой граф называют *графом переходов* КА.

Детерминированные и недетерминированные КА

КА называется *детерминированным*, если функция переходов этого автомата из любого состо яния по любому входному символу содержит не более одного следующего состояния. В противном случае КА называется *недетерминированным*. Условие детерминированности КА можно сформули ровать следующим образом: $\forall q \in Q, \forall a \in \Sigma$: либо $\delta(q, a) = \{q^0\}$, либо $\delta(q, a) = \emptyset$.

Для анализа цепочки с помощью детерминированного КА достаточно подать ее на вход автомата, выполнить все такты его работы и определить, перешел ли автомат в результате работы в одно из заданных конечных состояний.

Недетерминированный КА неудобен для анализа цепочек, так как в нем могут встречаться состо яния, допускающие неоднозначность, т.е. такие, из которых выходит две или более дуги, помеченных одним и тем же символом. Очевидно, что анализ цепочек и программирование работы для такого автомата – нетривиальная задача.

Доказано, что любой недетерминированный КА может быть преобразован в детерминированный так, чтобы их языки совпадали (говорят, что такие КА эквивалентны). В отличие от обычного КА функция переходов детерминированного КА может быть определена

как δ^0: $Q * \Sigma \rightarrow Q$ или $\delta(q, a) = q^0$, где $q \in Q$ – текущее состояние КА, $a \in \Sigma$ – текущий входной символ, а $q^0 \in Q$ – следующее состояние КА.

После построения детерминированный КА может быть минимизирован, т.е. для него может быть построен эквивалентный ему КА с минимально возможным числом состояний.

Построение КА на основе регулярной грамматики

Исходные данные для построения КА определяет регулярная грамматика, задающая язык, опи сывающий лексемы, которые должен уметь распознавать лексический анализатор. Таким образом, для построения лексического анализатора необходимо уметь строить КА на основе заданной регу лярной грамматики. Такое построение выполняется с помощью несложного алгоритма в два этапа:

1) исходная регулярная грамматика преобразуется к автоматному виду;

2) на основе автоматной грамматики строится КА.

В качестве примера рассмотрим лексический анализатор, выполняющий анализ лексем, представ ляющих собой целочисленные константы без знака в формате языка Си. В соответствии с требовани ями языка, такие константы могут быть десятичными, восьмеричными, либо шестнадцатеричными. Восьмеричной константой считается число, начинающееся с 0 и содержащее цифры от 0 до 7; шест надцатеричная константа должна начинаться с последовательности символов 0x и может содержать цифры, а также буквы от A до F (будем рассматривать только прописные буквы). Остальные числа считаются десятичными (правила их записи напоминать, наверное, не стоит). Будем считать, что всякое число завершается символом конца строки $\perp$.

Рассмотренные выше правила могут быть записаны в форме Бэкуса-Наура в грамматике цело численных констант без знака языка Си.

$G(\{S, G, X, H, Q, Z\}, \{0...9, A...F, x, \perp\}, P, S)$

P : $S \rightarrow G \perp | Z \perp | H \perp | Q \perp$

$G \rightarrow 1|2|3|4|5|6|7|8|9|G0|G1|G2|G3|G4|G5|G6|G7|G8|G9|Z8|Z9|Q8|Q9$

$H \rightarrow X0|X1|X2|X3|X4|X5|X6|X7|X8|X9|XA|XB|XC|XD|XE|XF|H0|H1|H2|H3|H4|H5|H6|H7|H8|H9|HA|HB|HC|HD|HE|HF$

$X \rightarrow Zx$

$Q \rightarrow Z0|Z1|Z2|Z3|Z4|Z5|Z6|Z7|Q0|Q1|Q2|Q3|Q4|Q5|Q6|Q7$

$Z \rightarrow 0$

Данная грамматика является регулярной грамматикой (леволинейной). Она также является ав томатной грамматикой, поэтому преобразование исходной грамматики к автоматному виду в данном случае не требуется. Следовательно, можно сразу построить КА. Получим следующий КА:

$M(\{N, Z, X, H, Q, G, S\}, 0...9, A...F, x, \perp, \delta, N, \{S\})$

δ : $\delta(G, \perp) = \{S\}$, $\delta(Z, \perp) = \{S\}$, $\delta(H, \perp) = \{S\}$, $\delta(Q, \perp) = \{S\}$, $\delta(N, 1) = \{G\}$, $\delta(N, 2) = \{G\}$, $\delta(N, 3) = \{G\}$, $\delta(N, 4) = \{G\}$, $\delta(N, 5) = \{G\}$, $\delta(N, 6) = \{G\}$, $\delta(N, 7) = \{G\}$, $\delta(N, 8) = \{G\}$, $\delta(N, 9) = \{G\}$, $\delta(G, 0) = \{G\}$, $\delta(G, 1) = \{G\}$, $\delta(G, 2) = \{G\}$, $\delta(G, 3) = \{G\}$, $\delta(G, 4) = \{G\}$, $\delta(G, 5) = \{G\}$, $\delta(G, 6) = \{G\}$, $\delta(G, 7) = \{G\}$, $\delta(G, 8) = \{G\}$, $\delta(G, 9) = \{G\}$, $\delta(Z, 8) = \{G\}$, $\delta(Z, 9) =$

$\{G\}$, $\delta(Q, 8) = \{G\}$, $\delta(Q, 9) = \{G\}$,

$\delta(X, 0) = \{H\}$, $\delta(X, 1) = \{H\}$, $\delta(X, 2) = \{H\}$, $\delta(X, 3) = \{H\}$, $\delta(X, 4) = \{H\}$, $\delta(X, 5) = \{H\}$, $\delta(X, 6) = \{H\}$, $\delta(X, 7) = \{H\}$, $\delta(X, 8) = \{H\}$, $\delta(X, 9) = \{H\}$, $\delta(X, A) = \{H\}$, $\delta(X, B) = \{H\}$, $\delta(X, C) = \{H\}$, $\delta(X, D) = \{H\}$, $\delta(X, E) = \{H\}$, $\delta(X, F) = \{H\}$, $\delta(H, 0) = \{H\}$, $\delta(H, 1) = \{H\}$, $\delta(H, 2) = \{H\}$, $\delta(H, 3) = \{H\}$, $\delta(H, 4) = \{H\}$, $\delta(H, 5) = \{H\}$, $\delta(H, 6) = \{H\}$, $\delta(H, 7) = \{H\}$, $\delta(H, 8) = \{H\}$, $\delta(H, 9) = \{H\}$, $\delta(H, A) = \{H\}$, $\delta(H, B) = \{H\}$, $\delta(H, C) = \{H\}$, $\delta(H, D) = \{H\}$, $\delta(H, E) = \{H\}$, $\delta(H, F) = \{H\}$,

$\delta(Z, x) = \{X\}$,

$\delta(Z, 0) = \{Q\}$, $\delta(Z, 1) = \{Q\}$, $\delta(Z, 2) = \{Q\}$, $\delta(Z, 3) = \{Q\}$, $\delta(Z, 4) = \{Q\}$, $\delta(Z, 5) = \{Q\}$, $\delta(Z, 6) = \{Q\}$, $\delta(Z, 7) = \{Q\}$, $\delta(Q, 0) = \{Q\}$, $\delta(Q, 1) = \{Q\}$, $\delta(Q, 2) = \{Q\}$, $\delta(Q, 3) = \{Q\}$, $\delta(Q, 4) = \{Q\}$, $\delta(Q, 5) = \{Q\}$, $\delta(Q, 6) = \{Q\}$, $\delta(Q, 7) = \{Q\}$,

$\delta(N, 0) = \{Z\}$

Построенный КА $M(\{N, Z, X, H, Q, G, S\}, \{0...9, A...F, x, \perp\}, \delta, N, \{S\})$ изображен на рис. 1 (без учёта дуг, показанных пунктирными линиями). Начальным состоянием данного КА является состо яние N, множество конечных состояний содержит одно состояние S.

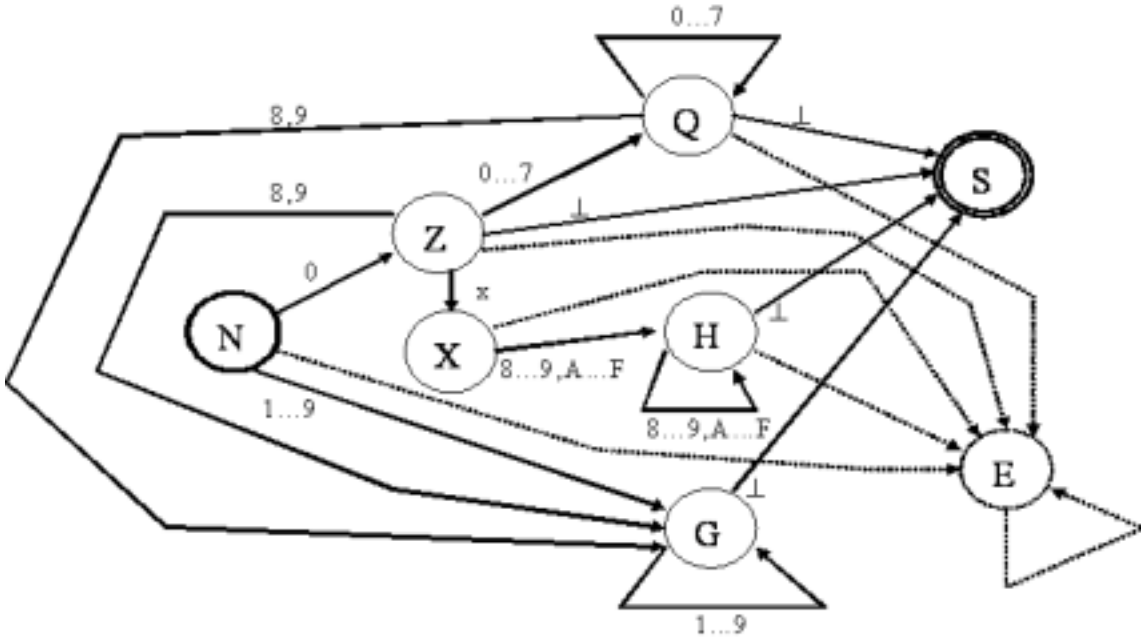


Рисунок 1 - Граф переходов детерминированного КА, распознающего целые числа языка Си

Очевидно, что построенный КА является детерминированным КА, но не является полностью определённым – в нём есть состояния, из которых нет переходов по одному или нескольким до пустимым входным символам. Например, из состояния G нет перехода по допустимому входному символу x. Это может быть неудобным при программировании работы данного КА.

Чтобы данный КА стал полностью определённым, в него дополнительно введено особое состоя ние E, обозначающее ошибку в распознавании цепочки символов (на рис. 1 это состояние и идущие в него дуги обозначены пунктирными линиями). На графе автомата дуги, идущие в это состояние, не нагружены символами. По принятому соглашению они обозначают функцию перехода по любому символу, кроме тех символов, которыми уже помечены другие дуги, выходящие из той же вершины графа. Такое соглашение принято, чтобы не загромождать обозначениями граф автомата.

ЛАБОРАТОРНАЯ РАБОТА №12 «БИБЛИОТЕКИ ДИНАМИЧЕСКОЙ КОМПОНОВКИ (DLL)».

Цель работы:

закрепить навыки решения задач на строки и манипуляции с файловыми системами, научиться проектировать и разрабатывать динамические библиотеки в ОС Windows и Linux

Требования к оформлению отчета

Отчет по лабораторной работе должен содержать следующие разделы (примеры оформления отчетов можно найти в папке с заданиями):

1) Изложение цели работы.
2) Задание по лабораторной работе с описанием своего варианта.
3) Спецификации ввода-вывода программы.
4) Текст программы (кратко).
5) Выводы по проделанной работе.

Задания – Windows

Разработать динамическую библиотеку DLL, включающую функцию (функции), реализующую следующий функционал для работы со строками (использовать только стандартные средства – биб лиотека string.h). Продемонстрировать ее подключение и использование.

1) Для двух строк, переданных в качестве параметров, получить итоговую строку-результат, пред ставляющую собой комбинацию переданных, составленную по следующему правилу: 1-я буква из 1-й строки, 2-я – из 2-й, 3-я – из 1-й, 4 – из 2-й и т.д.

2) Выполнить конкатенацию двух строк по правилу: первая строка остается без изменений, а вторая реверсируется.

3) Выполнить операцию «перекрещивания» двух строк, переданных в качестве параметров (для заданных позиций в обеих строках, первая часть новой первой строки берется из первой пе реданной строки до заданной позиции, а вторая часть – из второй переданной после заданной позиции во второй строке, для второй строки – тоже самое, только берутся альтернативные части оригинальных строк), полученные строки конкатенировать. Пример: для строк «abcd» и «hgjfs» и точек скрещивания 2 и 3 (для первой и второй строки соответственно) результат скрещивания будет «abfs» и «hgjcd», после итоговой конкатенации – «abfshgjcd».

4) Для входной строки получить строку-результат, в которой удалены все гласные буквы.

5) Для двух строк-параметров получить итоговую строку, представляющую собой конкатенацию данных строк в алфавитном порядке

6) Для входной строки получить строку-результат удалением каждого n-го символа исходной строки

7) Для двух строк-параметров метода, получить итоговую строку, в которой каждый n-й символ первой строки заменен на последовательно идущие символы из второй строки. Пример: для строк «abcd» и «hgjfs» и параметра n=2 результат операции будет такой: «ahcg»

8) Для строки-параметра получить строку-результат конкатенацией n копий исходной строки 1

Крощенко А.А., Системное программное обеспечение, ЛР8, 2020

9) Для входной строки получить строку-результат, в которой удалены все согласные буквы

10) Для входной строки получить строку-результат переводом всех символов исходной строки к верхнему регистру

Задания – Linux

Разработать динамическую библиотеку so, включающую функцию (функции), реализующую сле дующий функционал. Продемонстрировать ее подключение и использование. Для поиска необходи мых функций использовать ресурс https://www.die.net.

Все функции работают с домашней директорией пользователя!

1) Функция, создающая директорию с заданным именем и директории в ней с числовыми именами от 0 до заданного параметра n

2) Функция, дающая файлам в заданной директории новые последовательные числовые имена 3) Функция, читающая содержимое указанной директории и выводящая его на экран 4) Функция, удаляющая все пустые папки

5) Функция, дописывающая постфикс с именем директории к имени каждого файла в этой дирек тории. Например, если папка называется games, то каждый файл получает постфикс _games (если у файла имя tetris, то новое имя будет tetris_games)

6) Функция, удаляющая файл с указанным в качестве параметра именем

7) Функция, выполняющая переименование файла с указанным именем на заданное 8) Функция, проверяющая существование указанного файла

9) Функция, удаляющая указанную непустую папку

10) Функция, дописывающая префикс с именем директории к имени каждого файла в этой дирек тории. Например, если папка называется games, то каждый файл получает префикс games_ (если у файла имя tetris, то новое имя будет games_tetris)

Замечания

Для сдачи работы необходимо:

1) самостоятельно выполнить работу, изучив необходимую теорию;

2) представить отчет по лабораторной работе;

3) ответить на вопросы преподавателя по теме работы.

Руководства и ресурсы

1) Справочник по линуксовым утилитам и библиотекам Си: https://www.die.net

2) Пошаговое руководство к созданию DLL в Visual Studio (ENG): https://docs.microsoft.com/ru-ru/cpp/build/walkthrough-creating-and-using-a-dynamic-link-library-cpp?view= vs-2019

3) Создание динамических библиотек под Linux (читать шаг 4-9): http://www.firststeps.ru/linux/general1.html

4) Еще один источник по динамическим библиотекам в Linux, на этот раз в виде лабораторной работы: https://ita.sibsutis.ru/sites/csc.sibsutis.ru/files/courses/ spo/is-16/lab03.pdf

РАЗДЕЛ КОНТРОЛЯ ЗНАНИЙ

ПРИМЕРНЫЙ СПИСОК ВОПРОСОВ К ЗАЧЕТУ

1. Эволюция ОС и ЭВМ.
2. Определение ОС в качестве различных компонентов.
3. Способы построения ядра ОС.
4. Классификация ОС.
5. Требования предъявляемые к ОС.
6. Основные принципы построения ОС.
7. Процессы. Классификация процессов.
8. Диаграмма состояний процесса.
9. Планирование процессов. Виды планирования.
10. Алгоритмы планирования пакетной обработки данных.
11. Алгоритмы планирования в интерактивной системе.
12. Планирование системы реального времени.
13. Потоки. Способы реализации потоков.
14. Прерывания. Типы прерываний.
15. Механизм обработки прерываний.
16. Системные вызовы.
17. Межпроцессорное взаимодействие. Средства межпроцессорного взаимодействия.
18. Логическая организация взаимодействия.
19. Сигнальные средства связи.
20. Канальные средства связи.
21. Алгоритмы синхронизации. Взаимоисключение. Условие Бернстайна.
22. Критическая секция. Требования к алгоритмам синхронизации.
23. Запрет прерываний. Переменная замок.
24. Строгое чередование. Флаги готовности.
25. Алгоритм Патерсона.
26. Алгоритм Test-and-Set. Алгоритм Swap.
27. Механизмы синхронизации. Семафоры. Концепция семафоров.
28. Решение проблемы производителя и потребителя.
29. Мониторы, мьютексы, барьеры.
30. Тупики. Взаимоблокировка. Условия возникновения тупиков.
31. Способы борьбы с тупиками. Игнорирование тупиков.
32. Способы предотвращения тупиков.
33. Обнаружение взаимоблокировок при наличии нескольких ресурсов каждого типа.
34. Алгоритм банкира для одного вида ресурсов.
35. Нарушение условий взаимоисключения: условия ожидания, принципа отсутствия перераспределения.
36. Нарушение условия кругового ожидания.
37. Восстановление после тупиков.
38. Управления памятью. Физическая организация памяти.
39. Логическая память.
40. Связывание адресов. Функция системы управления памятью.
41. Схемы управления памятью. Схема с фиксированным разделом. Один процесс в памяти. Оверлейная структура.
42. Динамическое распределение. Свопинг. Схема с переменными разделами.
43. Страничная организация памяти.
44. Сегментная и сегментно-страничная организация памяти.

45. Виртуальная память. Страничная виртуальная память, сегментно-страничная виртуальная память.
46. Структура таблицы страниц. Ассоциативная память.
47. Инвертированная таблица страниц.
48. Исключительные ситуации при работе с памятью.
49. Алгоритмы замещения страниц. Алгоритм FIFO.
50. Оптимальный алгоритм (OPT). Алгоритм LRU,NRU.
51. Алгоритм NFU. Алгоритм «часы». Алгоритм Second-Chance.
52. Управление количеством страниц. Трешинг. Модель рабочего множества.
53. Страничные демоны. Программная поддержка сегментной модели памяти процесса.
54. Управление памятью в ОС Unix. Защита адресного пространства.
55. Основные принципы организации ввода/вывода.
56. Классификация устройств ввода/вывода. Контроллеры устройств ввода/вывода.
57. Адресация устройств ввода/вывода. Режимы управления ввода/вывода.
58. Прямой доступ к памяти DMA.
59. Базовая подсистема ввода/вывода и драйверы. Функции BIOS.
60. Буферизация и кэширование.
61. Spooling и захват устройств. Обработки прерываний и ошибок. Планирование запросов.
62. Многопроцессорные системы. Мультипроцессоры.
63. Сетевые ОС.

ПЕРЕЧЕНЬ ЛИТЕРАТУРЫ ДЛЯ ПОДГОТОВКИ

1. Тимович, Г. Л. Администрирование информационных систем / Г. Л.Тимович, Ж. И. Щербович, Юрча И. А. – Минск: Академия управления при Президенте Республики Беларусь, 2018. – 237 с.

2. Конспект лекций по дисциплине «Системное программное обеспечение ЭВМ» [Электронный ресурс]. – БГУИР, каф. электронных вычислительных систем, 2009. – Режим доступа: http://bsuir-helper.ru/sites/default/files/2010/10/08/konspekt/lkspo.pdf. – Дата доступа: 10.08.2020.

3. Таненбаум, Э. Современные операционные системы / Э. Таненбаум, Х. Бос. – 3-е изд. – СПб. : Питер. 2019. – 1120 с.

4. Гордеев, А. В. Системное программное обеспечение: учебное пособие / А. В. Гордеев, А. Ю. Молчанов. – СПб: Питер, 2001. – 736 с.

5. Хантер, Р. Проектирование и конструирование компиляторов / Р. Хантер. – М.: Финансы и статистика, 1984. – 232 с.

6. Дейтел, Х. М. Операционные системы. Основы и принципы / Х. М. Дейтел, П. Дж. Дейтел, Д. Р. Чофнес – 3-е изд. – М.: Бином. 2016. – 1024 с..

7. Руссинович, М. Внутреннее устройство Windows / М. Русинович [и др.] – 7-е изд. – СПб. : Питер. 2018. – 944 с.

8. Цай, Д. Аппаратное программное обеспечение персонального компьютера : учебное пособие / Д. Цай, И. Винниченко. – СПб. : Фолиант, 2017. – 192 с.

9. Ахо, А. В. Компиляторы: Принципы, технологии и инструменты / Ахо А.В., Сети Р. Ульман Дж. Д. – М.: Вильямс, 2008. – 1184 с.

10. Грибанов, В.П. Операционные системы : учебное пособие / В. П. Грибанов, С. В. Дробин, В. Д. Медведев. – М.: Финансы и статистика, 1990. – 242 с.

11. Конспект лекций по дисциплине «Операционные системы и системное программирование» [Электронный ресурс]. – БГУИР, каф. программного обеспечения информационных технологий, 2016. – Режим доступа: https://libeldoc.bsuir.by/handle/123456789/8491. – Дата доступа: 10.08.2020.

12. Риз, Д. Облачные вычисления / Д. Риз. – СПб : БХВ-Петербург, 2011. – 288 с.

13. Гофф, М. К. Сетевые распределенные вычисления: достижения и проблемы / М. К. Гофф. – Москва : КУДИЦ-Образ, 2005. – 320 с.

14. Столяров, А. В. Программирование на языке ассемблера NASM для ОС Unix : учебное пособие / А. В. Столяров. – М.: МАКС Пресс, 2011. – 188 с.

ВСПОМОГАТЕЛЬНЫЙ РАЗДЕЛ

Учреждение образования
«Брестский государственный технический университет»

УТВЕРЖДАЮ
Проректор по учебной работе
_____ М.В.Нерода
«28» 06 2024

Регистрационный № УД-24-1-073/уч

Операционные системы

Учебная программа для специальности:
6-05-0612-01 Программная инженерия

2024

Учебная программа составлена на основе образовательного стандарта (образовательных стандартов) ОСВО 6-05-0612-01-2022 и учебного плана специальности 6-05-0612-01 «Программная инженерия».

СОСТАВИТЕЛИ:
Ю. И. Давидюк, старший преподаватель кафедры ИИТ.

РЕЦЕНЗЕНТЫ:
А. Л. Брич, начальник сектора отдела разработки программного обеспечения ООО «Техартгрупп»
Л. П. Махнист, доцент кафедры математики и информатики БрГТУ, к.т.н.

РЕКОМЕНДОВАНА К УТВЕРЖДЕНИЮ:
Кафедрой интеллектуальных информационных технологий

Заведующий кафедрой В. А. Головко
(протокол № 9 от 17.06 2024);

Методической комиссией факультета электронно-информационных систем

Председатель методической комиссии С. С. Дереченник
(протокол № 9 от 22.06 2024);

Научно-методическим советом БрГТУ (протокол № 5 от 28.06 2024)

ПОЯСНИТЕЛЬНАЯ ЗАПИСКА

Дисциплина «Операционные системы» относится к модулю «Компьютерные технологии» компонента учреждения образования.

Актуальность изучения дисциплины обусловлена применением вычислительной техники практически на всех уровнях систем автоматизированной обработки информации в производственной и общественной деятельности. Независимо от масштаба таких систем, начиная от отдельных рабочих мест, локальных, корпоративных и глобальных вычислительных сетей, возникает ряд задач инсталляции и настройки операционной среды на условия применения прикладного программного обеспечения, обеспечения связи процессов в распределенных гетерогенных системах, функционирующих на основе различных стандартов и протоколов. Изучение системного программного обеспечения необходимо для решения комплексных задач по использованию и распределению ресурсов вычислительных систем, управлению их конфигурацией, производительностью и безопасностью. Знания, полученные при изучении предмета, позволят грамотно использовать функциональные возможности и сервисы современных операционных систем при разработке прикладного программного обеспечения, решении задач системного администрирования.

Цель дисциплины: изучение теоретических основ и приобретение практических навыков проектирования, реализации и сопровождения системных программных средств современных электронных вычислительных машин (ЭВМ), администрирования локальных и распределенных вычислительных систем.

Задачи дисциплины:

– формирование представления о системном программном обеспечении как примере использования и реализации системного подхода в чистом виде;

– приобретение знаний о возможностях, методах, моделях и средствах поддержки современных промышленных информационных технологий;

– формирование навыков практической работы со средствами обеспечения жизненного цикла создания и эволюционного развития сложных программных систем;

– овладение методами использования механизмов операционных систем для автоматизации функций прикладных систем.

Специалист, освоивший содержание образовательной программы высшего образования I ступени, должен обладать следующими специальными компетенциями (далее – СК):

СК-7. Проектировать компиляторы языков программирования.

СК-8. Строить и конфигурировать архитектуры вычислительных средств, основываясь на принципах функционирования операционных систем.

В результате изучения дисциплины «Операционные системы» студент формируются следующие компетенции:

академические:

– уметь применять базовые научно-теоретические знания для решения теоретических и практических задач;

– владеть системным и сравнительным анализом;

– владеть исследовательскими навыками;

– уметь работать самостоятельно;

– быть способным порождать новые идеи (обладать креативностью);

– владеть основными методами, способами и средствами получения, хранения, переработки информации с использованием компьютерной техники;

социально-личностные:

– быть способным к социальному взаимодействию;

– обладать способностью к межличностным коммуникациям;

– быть способным к критике и самокритике;

– уметь работать в команде;

профессиональные:

– на основе технической документации проводить работы по установке аппаратно-программного обеспечения систем обработки информации и их компонентов;

– подбирать соответствующее оборудование, аппаратуру и приборы и использовать их при проведении наладочных работ систем управления;

– организовывать и проводить испытания аппаратно-программного обеспечения систем обработки информации и их компонентов;

– проводить анализ эффективности функционирования систем обработки информации и выявлять узкие места по производительности и надежности;

– выявлять и устранять уязвимость систем обработки информации к угрозам безопасности;

– выполнять реконфигурацию баз данных и системного программного обеспечения под условия применения;

– выполнять обновление системного и прикладного программного обеспечения;

– выявлять актуальные проблемы развития и совершенствования информационных технологий;

– консультировать потребителей по вопросам выбора эффективных методов решения задач, связанных с представлением, хранением, отображением, передачей и аналитической обработкой информации.

В результате изучения учебной дисциплины обучающийся должен:

знать:

– назначение и возможности операционной системы;

– способы использования функций операционной системы и администрирования;

– принципы трансляции программ;

– командные средства системного программирования;

уметь:

– использовать средства операционной системы для решения различных прикладных задач;

– управлять операционной системой из командной строки или программы;

владеть:

– методами использования механизмов операционной системы для автоматизации функций прикладных систем;

– навыками установки, конфигурирования и администрирования современных операционных систем и компонент прикладных систем.

Базовой учебной дисциплиной по курсу «Операционные системы» является «Основы алгоритмизации и программирования». В свою очередь учебная дисциплина «Операционные системы» является базой для таких учебных дисциплин, как «Компьютерные информационные технологии», «Объектно-ориентированное программирование и проектирование», «Администрирование и программирование распределенных приложений».

План учебной дисциплины для заочной формы получения высшего образования

Код специальности (направления специальности)	Наименование специальности (направления специальности)	Курс	Семестр	Всего учебных часов	Количест-во зачетных единиц	Аудиторных часов (в соответствии с учебным планом УВО)			Форма текущей аттеста-ции
						Всего	Лек-ции	Лабора-торные занятия	
6-05-0612-01	Программная инженерия	2	3	216	6	96	48	48	- защиты лабораторных работ - сдача текущего контроля знаний по пройденной части курса лекций. - устный опрос на лекциях

СОДЕРЖАНИЕ УЧЕБНОГО МАТЕРИАЛА

ЛЕКЦИОННЫЕ ЗАНЯТИЯ, ИХ СОДЕРЖАНИЕ

Раздел 1. Введение в операционные системы.

Тема 1. Системное программное обеспечение и операционные системы (ОС).

Структура программного обеспечения вычислительной системы. Определение ОС. Требования, предъявляемые к ОС. Основные принципы построения ОС.

Тема 2. Классификация ОС.

Этапы развития ОС. Классификация ОС по критериям эффективности. Типы ядер ОС.

Раздел 2. Управление процессами в многозадачных ОС

Тема 3. Процессы и потоки.

Понятие процесса. Контекст и дескриптор процесса. Классификация процессов. Диаграмма состояния процесса. Многопоточность. Понятие потока. Способы реализации потоков.

Тема 4. Планирование процессов.

Планирование и диспетчеризация процессов. Три уровня планирования. Вытесняющее и невытесняющее планирование процессов. Цели применения алгоритмов планирования. Планирование в системах пакетной обработки данных: FCFS, SJN, SRT. Планирование в интерактивных системах: RR, приоритетное планирование. Планирование в системах реального времени: RMS, EDF.

Тема 5. Прерывания и системные вызовы.

Назначение и классы прерываний. Механизм обработки прерываний. Учет приоритета прерываний. Системный вызов. Обработка системных вызовов.

Раздел 3. Организация совместного функционирования процессов.

Тема 6. Межпроцессные коммуникации.

Средства межпроцессного взаимодействия. Сигнальные и канальные средства связи. Разделяемая память.

Тема 7. Синхронизация и взаимоисключения.

Синхронизация параллельных процессов. Критические ресурсы. Критический участок. Понятие взаимного исключения. Условия Бернстайна. Требования, предъявляемые к алгоритмам организации и взаимодействия процессов. Алгоритмы реализации взаимного исключения.

Тема 8. Механизмы синхронизации.

Системные механизмы синхронизации. Семафорные примитивы Дейкстры. Решение задач производителя и потребителя с помощью семафоров. Монитор Хоара как примитив синхронизации высокого уровня. Решение задачи производителя и потребителя с помощью мониторов.

Тема 9. Тупики.

Тупиковые ситуации. Методы борьбы с тупиками.

Раздел 4. Ввод-вывод и хранение информации.

Тема 10. Управление памятью.

Иерархия памяти. Локальность. Понятие физического и виртуального адреса. Связывание адресов. Статическое распределение памяти: разделы с фиксированными границами, один процесс в памяти. Динамическое распределение памяти: разделы с подвижными границами. Уплотнение памяти. Свопинг. Сегментная организация памяти. Страничная организация памяти. Таблицы страниц: одноуровневые, многоуровневые, инвертированные. Сегментно-страничная организация памяти. Защита адресного пространства задач в многозадачных ОС.

Тема 11. Виртуальная память.

Понятие виртуальной памяти. Страничный механизм трансляции. Стратегии управления виртуальной памятью. Дисциплины замещения страниц. Модель рабочего множества. Трешинг. Страничные демоны.

Тема 12. Ввод-вывод.

Физические принципы организации ввода-вывода. Классификация устройств ввода-вывода. Контроллеры устройств ввода-вывода. Прямой доступ к памяти DMA. Логические принципы ввода-вывода. Структура системы ввода-вывода. Драйверы устройств ввода-вывода. Функции базовой подсистемы ввода-вывода.

Тема 13. Файловые системы.

Файлы. Имена и атрибуты файлов. Варианты организации файлов. Директории. Структура файловой системы. Методы выделения дискового пространства. Учет свободного места. Монтирование файловых систем.

ПЕРЕЧЕНЬ ТЕМ ЛАБОРАТОРНЫХ ЗАНЯТИЙ, ИХ НАЗВАНИЕ

1. Интерфейс. Файлы. Команды.
2. Ссылки. Права доступа.
3. Bash. Потоки данных. Программирование.
4. Gcc. Процессы.
5. Ввод/вывод.
6. Средства межпроцессного взаимодействия.
7. Семафоры.
8. Создание сценариев командной строки. Bash-сценарии.
9. Проектирование гетерогенной компьютерной сети.
10. Сетевая инфраструктура «умного» дома.
11. Лексический анализатор.
12. Библиотеки динамической компоновки (DLL).

УЧЕБНО-МЕТОДИЧЕСКАЯ КАРТА УЧЕБНОЙ ДИСЦИПЛИНЫ
для дневной формы получения высшего образования

Номер раздела, темы	Название раздела, темы	Количество аудиторных часов		Кол-во часов самост. работы	Форма контроля знаний
		Лекции	Лабораторные занятия		
3-й семестр		48	48	120	зачет
1	Введение в операционные системы	4	8	16	Устный опрос
1.1	Системное программное обеспечение и ОС. Структура программного обеспечения вычислительной системы. Определение ОС. Требования, предъявляемые к ОС. Основные принципы построения ОС. Лабораторная работа №1. Лабораторная работа №2.	2		8	Защита отчета
1.2	Классификация ОС. Этапы развития ОС. Классификация ОС по критериям эффективности. Типы ядер ОС.	2	-	8	Устный опрос
2	Управление процессами в многозадачных ОС	16	12	30	-
2.1	Процессы и потоки. Понятие процесса. Контекст и дескриптор процесса. Классификация процессов. Диаграмма состояния процесса. Многопоточность. Понятие потока. Способы реализации потоков. Лабораторная работа №3.	4	4	12	Защита отчета
2.2	Планирование процессов. Планирование и диспетчеризация процессов. Три уровня планирования. Вытесняющее и невытесняющее планирование процессов. Цели применения алгоритмов планирования. Планирование в системах пакетной обработки данных: FCFS, SJN, SRT. Планирование в интерактивных системах: RR, приоритетное планирование. Планирование в системах реального времени: RMS, EDF.	8	-	10	Устный опрос
2.3	Прерывания и системные вызовы. Назначение и классы прерываний. Механизм обработки прерываний. Учет приоритета прерываний. Системный вызов. Обработка системных вызовов. Лабораторная работа №4. Лабораторная работа №5	4	8	8	Защита отчета
3	Организация совместного функционирования процессов	12	12	38	-
3.1	Межпроцессные коммуникации. Средства межпроцессного взаимодействия. Сигнальные и канальные средства связи. Разделяемая память. Лабораторная работа №6.	2	4	12	Защита отчета
3.2	Синхронизация и взаимоисключения. Синхронизация параллельных процессов. Критические ресурсы. Критический участок.	4	4	8	Защита отчета

Номер раздела, темы	Название раздела, темы	Количество аудиторных часов		Кол-во часов самост. работы	Форма контроля знаний
		Лекции	Лабораторные занятия		
	Понятие взаимного исключения. Условия Бернстайна. Требования, предъявляемые к алгоритмам организации и взаимодействия процессов. Алгоритмы реализации взаимного исключения. Лабораторная работа №7.				
3.3	Механизмы синхронизации. Системные механизмы синхронизации. Семафорные примитивы Дейкстры. Решение задач производителя и потребителя с помощью семафоров. Монитор Хоара как примитив синхронизации высокого уровня. Решение задачи производителя и потребителя с помощью мониторов. Лабораторная работа №8.	4	4	14	Защита отчета
3.4	Тупики. Тупиковые ситуации. Методы борьбы с тупиками.	2	-	4	Устный опрос
4	Ввод-вывод и хранение информации	16	16	36	-
4.1	Управление памятью. Иерархия памяти. Локальность. Понятие физического и виртуального адреса. Связывание адресов. Статическое распределение памяти: разделы с фиксированными границами, один процесс в памяти. Динамическое распределение памяти: разделы с подвижными границами. Уплотнение памяти. Свопинг. Сегментная организация памяти. Страничная организация памяти. Таблицы страниц: одноуровневые, многоуровневые, инвертированные. Сегментно-страничная организация памяти. Защита адресного пространства задач в многозадачных ОС. Лабораторная работа №9. Лабораторная работа №10.	8	8	12	Защита отчета
4.2	Виртуальная память. Понятие виртуальной памяти. Страничный механизм трансляции. Стратегии управления виртуальной памятью. Дисциплины замещения страниц. Модель рабочего множества. Трешинг. Страничные демоны. Лабораторная работа №11.	2	4	8	Защита отчета
4.3	Ввод-вывод. Физические принципы организации ввода-вывода. Классификация устройств ввода-вывода. Контроллеры устройств ввода-вывода. Прямой доступ к памяти (DMA). Логические принципы ввода-вывода. Структура системы ввода-вывода. Драйверы устройств ввода-вывода. Функции базовой подсистемы ввода-вывода.	4	4	8	Защита отчета

Номер раздела, темы	Название раздела, темы	Количество аудиторных часов		Кол-во часов самост. работы	Форма контроля знаний
		Лекции	Лабораторные занятия		
	Лабораторная работа №12.				
4.4	Файловые системы. Файлы. Имена и атрибуты файлов. Варианты организации файлов. Директории. Структура файловой системы. Методы выделения дискового пространства. Учет свободного места. Монтирование файловых систем.	2	-	8	Устный опрос

ИНФОРМАЦИОННО-МЕТОДИЧЕСКАЯ ЧАСТЬ

Перечень литературы

Основная

1. Тимович, Г. Л. Администрирование информационных систем / Г. Л.Тимович, Ж. И. Щербович, Юрча И. А. – Минск: Академия управления при Президенте Республики Беларусь, 2018. – 237 с.

2. Конспект лекций по дисциплине «Системное программное обеспечение ЭВМ» [Электронный ресурс]. – БГУИР, каф. электронных вычислительных систем, 2009. – Режим доступа: http://bsuir-helper.ru/sites/default/files/2010/10/08/konspekt/lkspo.pdf. – Дата доступа: 10.08.2020.

3. Таненбаум, Э. Современные операционные системы / Э. Таненбаум, Х. Бос. – 3-е изд. – СПб. : Питер. 2019. – 1120 с.

4. Гордеев, А. В. Системное программное обеспечение: учебное пособие / А. В. Гордеев, А. Ю. Молчанов. – СПб: Питер, 2001. – 736 с.

5. Хантер, Р. Проектирование и конструирование компиляторов / Р. Хантер. – М.: Финансы и статистика, 1984. – 232 с.

6. Дейтел, Х. М. Операционные системы. Основы и принципы / Х. М. Дейтел, П. Дж. Дейтел, Д. Р. Чофнес – 3-е изд. – М.: Бином. 2016. – 1024 с..

7. Руссинович, М. Внутреннее устройство Windows / М. Русинович [и др.] – 7-е изд. – СПб. : Питер. 2018. – 944 с.

8. Цай, Д. Аппаратное программное обеспечение персонального компьютера : учебное пособие / Д. Цай, И. Винниченко. – СПб. : Фолиант, 2017. – 192 с.

Дополнительная

1. Ахо, А. В. Компиляторы: Принципы, технологии и инструменты / Ахо А.В., Сети Р. Ульман Дж. Д. – М.: Вильямс, 2008. – 1184 с.

2. Грибанов, В.П. Операционные системы : учебное пособие / В. П. Грибанов, С. В. Дробин, В. Д. Медведев. – М.: Финансы и статистика, 1990. – 242 с.

3. Конспект лекций по дисциплине «Операционные системы и системное программирование» [Электронный ресурс]. – БГУИР, каф. программного обеспечения информационных технологий, 2016. – Режим доступа: https://libeldoc.bsuir.by/handle/123456789/8491. – Дата доступа: 10.08.2020.

4. Риз, Д. Облачные вычисления / Д. Риз. – СПб : БХВ-Петербург, 2011. – 288 с.

5. Гофф, М. К. Сетевые распределенные вычисления: достижения и проблемы / М. К. Гофф. – Москва : КУДИЦ-Образ, 2005. – 320 с.

6. Столяров, А. В. Программирование на языке ассемблера NASM для ОС Unix : учебное пособие / А. В. Столяров. – М.: МАКС Пресс, 2011. – 188 с.

ПЕРЕЧЕНЬ КОМПЬЮТЕРНЫХ ПРОГРАММ

Перечень компьютерных программ, наглядных и других пособий, методических указаний и материалов, технических средств обучения, оборудования для выполнения лабораторных работ:

1. Класс современных персональных ЭВМ.
2. Система программирования Microsoft Visual C++ 2005 и выше.
3. GCC компилятор.
4. Инсталляционные пакеты современных версий операционных систем Microsoft Windows. Linux.

ПЕРЕЧЕНЬ СРЕДСТВ ДИАГНОСТИКИ РЕЗУЛЬТАТОВ УЧЕБНОЙ ДЕЯТЕЛЬНОСТИ

Для диагностики результатов учебной деятельности используются:
- письменные отчеты по лабораторным работам;
- электронные тесты;
- письменные отчеты по аудиторным практическим упражнениям;
- отчеты по лабораторным работам с их устной защитой;
- контрольные работы;
- зачеты;
- текущая аттестация.

Текущая аттестация студентов по дисциплине проводится в виде:

- защиты лабораторных работ для допуска к аттестации. Для допуска к первой аттестации необходимо сдать 2 лабораторные работы, для допуска ко второй аттестации необходимо сдать 4 лабораторные работы, для допуска к промежуточному контролю необходимо сдать 7 лабораторных работ.
- сдача текущего контроля знаний по пройденной части курса лекций.
- контрольный опрос по теоретической части курса с учетом промежуточной аттестации.

При расчете итоговой отметки учитывается результат сдачи текущих аттестаций и промежуточного контроля знаний.

МЕТОДИЧЕСКИЕ РЕКОМЕНДАЦИИ ПО ОРГАНИЗАЦИИ И ВЫПОЛНЕНИЮ САМОСТОЯТЕЛЬНОЙ РАБОТЫ

В соответствии с п. 3 Положения о самостоятельной работе студентов учреждения образования «Брестский государственный технический университет», утвержденного ректором БрГТУ №56 от 01.06.2020, время, отведенное на самостоятельную работу, может использоваться обучающимися на:

- проработку тем (вопросов), вынесенных на самостоятельное изучение;
- выполнение типовых расчетов;

- решение задач;
- составление алгоритмов, схем;
- выполнение исследовательских и творческих заданий;
- подготовку сообщений, тематических докладов, рефератов, презентаций;
- выполнение практических заданий;
- конспектирование учебной литературы;
- подготовку отчетов;
- составление обзора научной (научно-технической) литературы по заданной теме;
- выполнение патентно-информационного поиска;
- аналитическую обработку текста (аннотирование, реферирование, рецензирование, составление резюме);
- подготовку докладов;
- подготовку презентаций;
- оформление рекламных, информационных и демонстрационных материалов (стенды, газеты и пр.);
- составление тестов;
- изготовление макетов, лабораторно-учебных пособий;
- составление тематической подборки литературных источников, интернет-источников;
- оформление и сопровождение интернет-страниц, сайтов, блогов.

Изучение и конспектирование литературы по разделам.

Раздел 1.

Тема: Основные принципы построения ОС. Типы ядер ОС.

Материал: Таненбаум, Э. Современные операционные системы. Глава 1. Введение. История операционных систем. Обзор аппаратного обеспечения компьютера. Зоопарк операционных систем. Структура операционной системы.

Раздел 2.

Тема: Процессы и потоки. Планирование процессов.

Материал: Таненбаум, Э. Современные операционные системы. Глава 2. Процессы и потоки. Потоки;

Тема: Прерывание и системные вызовы.

Материал: Руссинович, М. Внутреннее устройство Windows. Глава 3. Процессы и задания.

Тема: Планирование и диспетчеризация процессов и потоков.

Материал: Таненбаум, Э. Современные операционные системы. Глава 2. Процессы и потоки. Планирование.

Раздел 3.

Тема: Средства межпроцессного взаимодействия. Семафорные примитивы Дейкстры.

Материал: Таненбаум, Э. Современные операционные системы. Глава 2. Взаимодействие процессов;

Тема: Механизмы синхронизации. Тупики.

Материал: Конспект лекций по дисциплине «Системное программное обеспечение ЭВМ» [Электронный ресурс]. – БГУИР, каф. электронных вычислительных систем. Глава 3. Механизмы взаимного исключения. Типовые механизмы синхронизации.

Раздел 4.

Тема: Управление памятью. Виртуальная память

Материал: Гордеев, А. В. Системное программное обеспечение: учебное пособие. Глава 4. Управление вводом/выводом и файловые системы.

Тема: Физические принципы организации ввода-вывода. Логические принципы ввода-вывода.

Материал: Таненбаум, Э. Современные операционные системы. Глава 5. Основы аппаратного обеспечения ввода-вывода.

Тема: Файловая система.

Материал: Руссинович, М. Внутреннее устройство Windows. Глава 6. Подсистема ввода/вывода.