синтаксис ассемблеров фирмы Intel. NASM позволяет создавать программы на всех известных на сегодня платформах на базе архитектуры x86 и имеет хорошую поддержку макросов. Из отличий, как правило, выделяют то, что синтаксис ассемблера NASM является регистрочувствительным – директивы набираются прописными литерами, а инструкции – строчными.

Из-за своей специфики, а также по традиции, для программирования на языке ассемблера существует мало интерактивных сред программирования, но есть дополнительные модули к универсальным средам – Eclipse, NetBeans и пр., облегчающие программирование и отладку на этом языке. Традиционно, однако, в лабораторных работах по программированию на языке ассемблера используются автономный текстовый редактор и утилиты командной строки.

При выборе текстового редактора был проанализирован ряд программных продуктов и сделан выбор в пользу программы mcedit благодаря минимальным отличиям от консольных редакторов, привычных для студентов, и возможности автоматической подсветкой синтаксиса программ.

В качестве средства визуальной отладки выбран отладчик DDD (Data Display Debugger), разрабатывавшийся длительное время различными организациями, компаниями и университетами. Он является графической оболочкой (front-end) к различным консольным отладчикам и использует их для работы. В качестве последнего был выбран отладчик GDB (Gnu DeBugger), активно применяемый при разработке программного обеспечения в сфере встраиваемых систем и микроконтроллеров [7].

ЛИТЕРАТУРА

1. Assembly programming language statistics – Ohloh.http://www.ohloh.net/languages/19. – Доступ 28.04.2009.

2. Пирогово В.Ю. Assembler для Windows. – М.: Изд-во Молгачева, 2005. – 552 с.

3. Магда Ю.С. Использование ассемблера для оптимизации программ на C++. СПб.: БХВ-Петербург, 2004. – 492 с.

4. Магда Ю.С. Ассемблер. Разработка и оптимизация Windows-приложений. СПб.: БХВ-Петербург, 2003. – 544 с.

5. Brown E. Linux spending will defy recession, IDC claims. http://www.linuxfoundation.org/sites/main/files/publications/Linux_in_New_Economy.pdf. – Доступ 13.04.2009.

6. Таненбаум Э. Современные операционные системы, – СПб.: Питер, 2002. – 1040 с.

7. Debugging with DDD. 15/01/2004. http://www.gnu.org/manual/ddd/

УДК 004.514.62

Жук А.М.

Научный руководитель: к.т.н., доцент Костюк Д.А.

АНАЛИЗ АРХИТЕКТУР МОДУЛЕЙ ЯДРА СОВРЕМЕННЫХ ОС ДЛЯ ОБУЧЕНИЯ ПРОГРАММИРОВАНИЮ НА ЯЗЫКЕ АССЕМБЛЕР

В настоящее время программирование на ассемблере практически повсеместно преподается на примере ОС DOS. Такой выбор платформы вызван наличием большого числа учебных пособий по ассемблеру, написанных во время популярности DOS, а также ее относительной простотой по сравнению с более современными ОС.

Вместе с тем в настоящее время программное обеспечение, написанное для DOS, применяется все реже, и практически всегда – на более современных операционных системах в режиме совместимости или эмуляции. Определить число компьютеров, использующих DOS в качестве основной ОС, представляется крайне затруднительным, но т.к. данная ОС не присутствует в статистике, собираемой среди компьютеров, подклю-

ченных к сети Интернет, можно предполагать, что доля для данного сегмента вычислительных систем составляет мене 0.01%.

Таким образом, смена платформы для изучения ассемблера актуальна. Однако она требует решения ряда проблем. Одной из таких проблем является необходимость работы с защищенным режимом процессора. Ряд ассемблерных инструкций, легко доступных программисту в однозадачной ОС и на 16-битной архитектуре, может быть выполнен на 32-битной архитектуре только в привилегированном режиме, в котором работает ядро ОС. Для практического освоения таких инструкций студенту необходимо научиться создавать подключаемые модули ядра ОС, в первую очередь – драйвера устройств.

Ядра современных операционных систем позволяют расширить свою функциональность добавлением модулей драйверов устройств во время работы. Модуль обеспечивает функциональность, которая позволит взаимодействовать ядру с новым внешним устройством.

Таким образом, для выбора современной ОС в качестве платформы преподавания языка ассемблер не последнюю роль играет сложность программирования драйверов устройств. В данной работе выполнен сравнительный анализ архитектуры драйверов двух наиболее перспективных современных ОС общего назначения: Microsoft Windows и GNU/Linux.

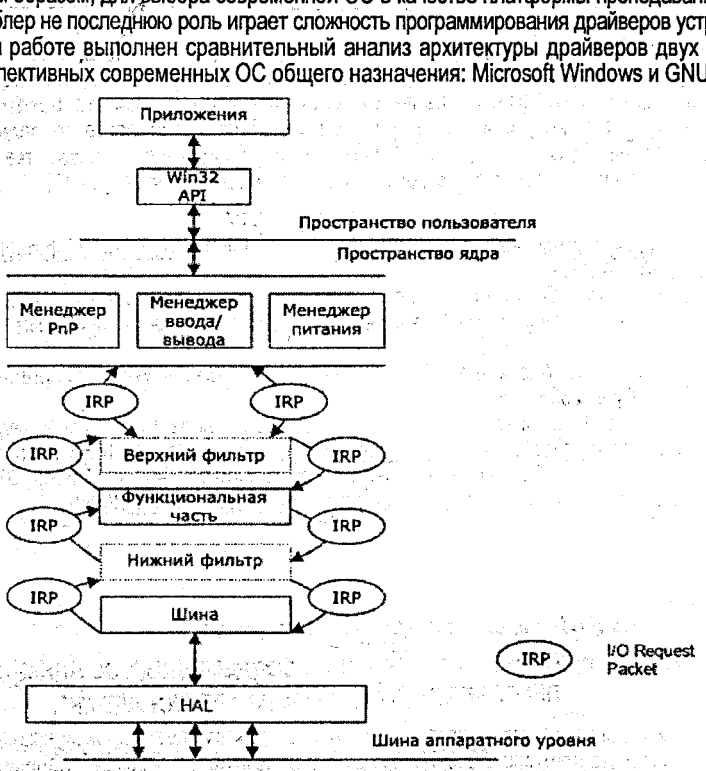

Рисунок 1 – Архитектура драйвера WDM

Существуют три класса драйверов Windows Driver Model (WDM): фильтры, функциональные части и шины. Вдобавок, драйверы WDM должны быть совместимы с технологией plug and play (PnP), поддерживать управление питанием и интерфейс Windows Management Instrumentation (WMI). Для взаимодействия модулей используются пакет запроса ввода/вывода (I/O request packet – IRP). Доступ к устройству на аппаратном уровне осуществляется через уровень абстрагирования от аппаратных средств (HAL).

Драйверы в Linux представлены модулями, которые являются кусками кода, расширяющими функциональность ядра ОС. Взаимодействие между модулями осуществляется при помощи вызовов функций. Во время загрузки модуль экспортирует все функции, которые он хочет сделать общедоступными, в символьную таблицу, поддерживаемую ядром Linux, после чего эти функции становятся видимыми другими модулями. Доступ к устройству также осуществляется через HAL, реализация которого зависит от аппаратной платформы, для которой скомпилировано ядро – например, x86 или SPARC.

Как видно на рисунках, существует определенное сходство между двумя ОС. В обеих системах драйверы являются модульными компонентами, расширяющими функциональность ядра. Взаимодействие между слоями драйверов в Windows осуществляется с использованием IRP, поддерживаемыми как аргументы стандартной системы и объявленных драйверами функций, в то время как в Linux используются вызовы функций со специальными параметрами. В Windows есть отдельные компоненты ядра, управляющие PnP, вводом/выводом и питанием.

В Linux нет четкой границы между «слоями» в модулях, это значит, что модули не делятся на шины, функциональные части и фильтры. В ядре нет четко определенных менеджеров PnP и питания, которые отправляют стандартизованные сообщения модулям в нужное время. Ядро должно иметь загружаемые модули, реализующие управление питанием или функциональность PnP, но интерфейс этих модулей с драйверами не является обязательным.

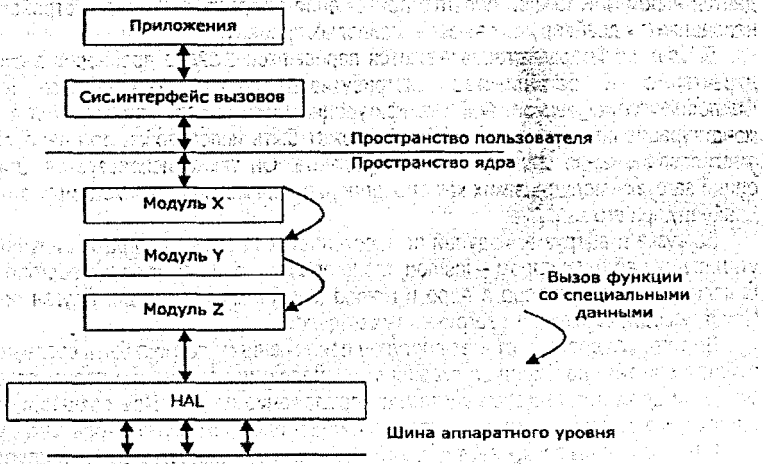

Рисунок 2 - Архитектура драйвера Linux

В обоих окружениях доступ к аппаратному уровню осуществляется через интерфейс HAL, реализованный для конкретной платформы. Также общей чертой обеих архитектур является то, что драйверы могут быть загружены в ядро во время его работы.

При инсталляции драйверов Windows использует установочную информацию, содержащуюся в текстовом файле, называемом INF-файлом. Создатель драйвера отвечает за предоставление INF-файла для драйвера. Приложение Windows с графическим интерфейсом GetInf позволяет генерировать INF-файлы для драйверов. Ему необходимы название компании и класс устройства. В Windows есть большое количество предопределенных классов для устанавливаемых драйверов.

В INF-файле также должен быть указан ID-номер для PnP-совместимого устройства, после чего система будет использовать этот ID для идентификации устройства при его

подключении. ID аппаратного обеспечения хранится в нем самом и читается системой при подключении. После успешной установки INF-файла для нового устройства, его драйвер, имеющий специальный ID, будет загружаться каждый раз при подключении устройства к системе и выгружаться при его отключении.

Драйверы устройств в Linux также выполняют различные процедуры ввода/вывода и операции контроля устройства. Здесь нет объекта драйвера, видимого для него, вместо этого драйверы внутренне управляются ядром.

Каждый драйвер в Linux включает операции регистрации драйвера (загрузки в память) и его дерегистрации (выгрузки из памяти). Создатели драйверов используют макросы ядра module_init и module_exit для указания особенных операций, которые будут назначены вместо процедур регистрации и дерегистрации.

В Linux устройства именуются номерами в пределах от 0 до 4095, называемыми старшими номерами. Это подразумевает ограничение в 4095 используемых устройств, т.е. устройств, к которым может получить доступ приложение, но каждый драйвер для такого «старшего» устройства может управлять более чем миллионом дополнительных суб-устройств. Эти управляемые драйвером устройства нумеруются, используя числа, называемые младшими номерами.

Организационные процедуры в Windows были встроены в процедуру подключения в драйвер. В Linux эти организационные процедуры известны как файловые операции и представлены структурой, называемой file_operations. Структура, называемая file, создается ядром при каждой попытке приложения получить описатель устройства и перенаправляется драйверу при вызове файловых процедур.

В Linux драйверы устанавливаются пересылкой файлов драйверов в специальную директорию. В большинстве дистрибутивов модули размещаются в каталоге /lib/modules/kernel_version. Файл конфигурации modules.conf размещается в системной конфигурационной директории /etc. Он может быть использован для переопределения месторасположения специфического драйвера. Он также используется для описания опций загрузки модуля, таких как определение параметров, необходимых для передачи драйверу при его загрузке.

Загрузка и выгрузка модулей осуществляется набором программ, идущих в пакете утилит для работы с ядром – insmod, modprobe и rmmod. Insmode и modprobe загружают бинарный образ драйвера в ядро и rmmod выгружает его оттуда. Другая программа – lsmod выводит список всех загруженных модулей.

До того, как драйвер станет доступен приложениям, должна быть создана точка устройства для того драйвера со старшими и младшими номерами в папке устройств /dev. Для этой цели используется системное приложение mknod. При создании такой точки необходимо указать, каким является устройство: блочным или символьным.

В целом анализ показывает бóльшую простоту создания драйверов и других модулей ядра для GNU/Linux благодаря изначально принятым архитектурным решениям.

В случае Windows написание драйверов устройств традиционно считается сложной темой и часто не входит в учебные пособия по программированию, однако рассмотрено в специализированной литературе, в т.ч. переведенной на русский язык. В качестве языка написания драйверов в них предлагается использовать Си.

Также существуют электронные пособия, как краткие, так и более полные, иллюстрирующие процесс написания модулей ядра Linux на С и принципиально применимые для ассемблерных программ. Однако на сегодняшний день они являются исключительно англоязычными.

ЛИТЕРАТУРА

1. Пирогов В.Ю. Assembler для Windows. М.: Изд-во Молгачева, 2005. – 552 с.

2. J. Pranevich. The Wonderful World of Linux 2.6. http://kniggit.net/wwol26.html 01.09.2003.

3. Магда Ю.С. Ассемблер. Разработка и оптимизация Windows-приложений. – СПб.: БХВ-Петербург, 2003. – 544 с.
4. R. Hyde. Writing Linux Device Drivers in Assembly Language http://webster.cs.ucr.edu/Page_Linux/1_LDD.pdf 10.11.2002. 116 p.
5. M. Tsegaye, R. Foss. A Comparison of the Linux and Windows Device Driver Architectures.http://www.cs.ru.ac.za/research/g98t4414/static/papers/oscomposr.pdf May 7, 2009. 26 p.

УДК 62-529
Иванюк Д.С.
Научный руководитель: к.т.н., доцнент Шуть В.Н.

ПРИМЕНЕНИЕ НЕЙРОКОНТРОЛЛЕРОВ В АСУТП

В современных условиях управление производством становится все сложнее, требования к эффективности более высокими. Один из путей улучшения может заключаться за счет совершенствования применяемых на уровне АСУТП подходов к управлению – применению последних разработок в данной области, одной из которых является нейроуправление.

1. Общие сведения о нейроуправлении

Нейроуправление – относительно молодое направление научных исследований, которое стало самостоятельным в 1988 г. Однако исследования в этой области начались гораздо раньше. Одно из определений науки «кибернетика» рассматривает ее как общую теорию управления и взаимодействия не только машин, но и биологических существ. Нейроуправление пытается реализовать данное положение через построения систем управления (систем принятия решений), которые могут обучаться во время функционирования и, таким образом, улучшать свою эффективность работы. При этом такие системы используют параллельные механизмы обработки информации, подобно мозгу живых организмов [1].

Долгое время была популярна идея построения совершенной системы управления – универсального контроллера, который извне выглядел бы как «черный ящик». Он мог бы использоваться для управления любыми системами, имея связи с датчиками, исполнительными механизмами, другими контроллерами и специальную связь с «модулем эффективности» – системой, которая определяет эффективность управления исходя из заданных критериев. Пользователь такой системы управления задавал бы только желаемый результат, далее обученный контроллер управлял бы самостоятельно, возможно придерживаясь сложной стратегии достижения в будущем желаемого результата. Также он бы все время корректировал свое управление исходя из реакции объекта управления для достижения максимальной эффективности. Общая схема такой системы приведена на рис. 1.

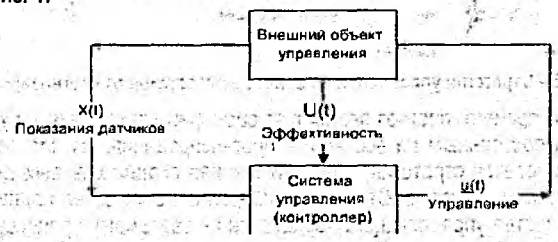

Рис 1 – Система с подкрепляющим обучением

ПИ- и ПИД-контроллеры были одними из первых систем управления [2]. Далее более подробно рассматриваются такие системы.