МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ ФЕДЕРАЦИИ
ФЕДЕРАЛЬНОЕ АГЕНТСТВО ПО ОБРАЗОВАНИЮ

Государственное образовательное учреждение
высшего профессионального образования
«Оренбургский государственный университет»

А.Ю. КРУЧИНИН

ОПЕРАЦИОННЫЕ СИСТЕМЫ

Рекомендовано Учёным советом к изданию государственного образовательного высшего профессионального образования «Оренбургский государственный университет» в качестве учебного пособия для студентов, обучающихся по программам высшего профессионального образования по направлению «Информатика и вычислительная техника»

Оренбург 2009

УДК 004(07)
ББК 32.81я7
К 84

Рецензент
кандидат технических наук А.Л. Коннов

Кручинин А.Ю.
К 84 **Операционные системы: учебное пособие / А.Ю. Кручинин. – Оренбург: ГОУ ОГУ, 2009. – 132 с.**

Данное пособие содержит цикл лекций по дисциплине «Операционные системы», охватывающий основные разделы курса в соответствии с требованиями утвержденной программы.

Учебное пособие предназначено для студентов специальности 230101 «Вычислительные машины, комплексы, системы и сети».

УДК 004(07)
ББК 32.81я7

© А.Ю. Кручинин, 2009
© ГОУ ОГУ, 2009

Содержание

1 Начальные сведения об операционных системах……………………………..5
1.1 Назначение и функции операционных систем…………………………5
1.2 История развития операционных систем……………………………….6
1.3 Классификация операционных систем…………………………………7
1.4 Обзор аппаратного обеспечения компьютера…………………………..8
1.5 Архитектура операционной системы…………………………………...13
1.5.1 Классическая архитектура……………………………………………..14
1.5.2 Микроядерная архитектура…………………………………………….17
Контрольные вопросы по разделу………..………………………………….20
2 Процессы и потоки……………………………………………………………21
2.1 Процессы……………………………………………………………………21
2.2 Потоки……………………………………………………………………….24
2.3 Межпроцессное взаимодействие…………………………………………..25
2.3.1 Взаимное исключение с активным ожиданием……………………….27
2.3.2 Примитивы межпроцессного взаимодействия…………………………29
2.4 Планирование………………………………………………………………..32
2.4.1 Планирование в системах пакетной обработки данных………………..34
2.4.2 Планирование в интерактивных системах………………………………35
2.4.3 Планирование в системах реального времени…………………………..37
2.5 Понятие взаимоблокировки………………………………………………...38
Контрольные вопросы по разделу..…………………………………………….40
3 Управление памятью……………………………………………………………41
3.1 Основы управления памятью……………………………………………….41
3.2 Методы распределения памяти без использования подкачки………………...44
3.2.1 Метод распределения с фиксированными разделами…………………44
3.2.2 Метод распределения с динамическими разделами……………………45
3.2.3 Метод распределения с перемещаемыми разделами…………………...47
3.3 Методы распределения памяти с подкачкой на жесткий диск…………………..47
3.3.1 Страничная организация памяти……………………………………….....49
3.3.2 Сегментная организация памяти…………………………………………56
3.3.3 Сегментно-страничная организация памяти…………………………….58
3.4 Кэширование данных…………………………………………………………59
Контрольные вопросы по разделу..…………………………………………….61
4 Аппаратная поддержка мультипрограммирования на примере процессора Pentium………………….…………………………………………………….....62
4.1 Регистры……………………………………………………………………….62
4.2 Привилегированные команды………………………………………………...65
4.3 Сегментация с использованием страниц……………………………………..65
4.4 Защита данных в процессоре Pentium………………………………………..70
4.5 Средства вызова процедур и задач…………………………………………...74
4.6 Механизм прерываний………………………………………………………...78
4.7 Кэширование в процессоре Pentium……………………………………….80
Контрольные вопросы по разделу..…………………………………………….81

5 Ввод-вывод……………………………………………………………………82
5.1 Принципы аппаратуры ввода-вывода……………………………………...82
5.2 Принципы программного обеспечения ввода-вывода…………………....86
Контрольные вопросы по разделу..……………………………………………88
6 Файловые системы……………………………………………………………90
6.1 Основы файловых систем…………………………………………………..90
6.2 Файловая система FAT……………………………………………………...95
6.3 Файловая система NTFS……………………………………………………97
6.4 Файловые системы Ext2, Ext3 и UFS……………………………………..100
Контрольные вопросы по разделу..…………………………………………...102
7 Безопасность операционных систем………………………………………..103
7.1 Основы безопасности……………………………………………………...103
7.2 Аутентификация пользователей…………………………………………..104
7.3 Атаки изнутри операционной системы…………………………………..106
7.4 Атаки операционной системы снаружи………………………………….108
Контрольные вопросы по разделу..…………………………………………...110
8 Обзор современных операционных систем………………………………..111
8.1 Операционная система Windows 2000…………………………………….111
8.1.1 Структура Windows 2000…………………………………………………..112
8.1.2 Реализация интерфейса Win32……………………………………………115
8.1.3 Эмуляция MS-DOS………………………………………………………...117
8.2 Архитектура UNIX-образных операционных систем…………………...118
8.3 Мультипроцессоры и мультипроцессорные операционные системы…………122
8.4 Операционные системы реального времени и мобильные операционные системы…………………………………………………………………………..126
8.4.1 Операционная система Windows CE 5.0…………………………………..128
Контрольные вопросы по разделу..…………………………………………...131
Список использованных источников...………………………………………132

1 Начальные сведения об операционных системах

1.1 Назначение и функции операционных систем

Операционная система компьютера представляет собой комплекс взаимосвязанных программ, который действует как интерфейс между приложениями и пользователями с одной стороны, и аппаратурой компьютера с другой стороны [11]. Операционная система выполняет две группы функций:

- предоставляет пользователю или программисту вместо реальной аппаратуры компьютера расширенной виртуальной машины;
- повышает эффективность использования компьютера путем рационального управления его ресурсами в соответствии с некоторым критерием.

Пользователь, как правило, не интересуется деталями устройства аппаратного обеспечения компьютера, он видится ему как набор приложений, которые можно написать на одном из языков программирования. Операционная система предоставляет программисту ряд возможностей, которые могут использовать программы с помощью специальных команд, называемых системными вызовами. Поэтому программное приложение включает в себя множества системных вызовов, необходимых, например, для работы с файлами. Операционная система скрывает от программиста детали аппаратного обеспечения и предоставляет удобный интерфейс для исполнения системы операционной среды.

В тоже время операционная система выступает в качестве менеджера ресурсов. В соответствии с этим подходом работа операционной системы заключается в обеспечении организованного и контролируемого распределения процессоров, памяти и устройств ввода-вывода между различными программами. Работа операционной системы имеет следующие особенности:

- функции операционной системы работают так же, как и остальное программное обеспечение – реализуются в виде отдельных программ или набора программ, исполняющихся процессов;
- операционная система должна передавать управление другими процессами и ожидать, когда процессор снова выделит ей время для выполнения своих обязанностей.

Управление ресурсами включает решение следующих общих, не зависящих от типа ресурса задач:

- планирование ресурса – то есть определение, какому процессу, когда и в каком количестве (если ресурс может выделяться частями) следует выделить данный ресурс;
- удовлетворение запросов на ресурсы;
- отслеживание состояния и учет использования ресурса – то есть поддержание оперативной информации о том, занят или свободен ресурс и какая доля ресурса уже распределена;
- разрешение конфликтов между процессами [11].

Управление ресурсами включает в себя их мультиплексирование (распределение) двумя способами: во времени и в пространстве. Когда ресурс

распределяется во времени, различные пользователи и программы используют его по очереди. Сначала один из них получает доступ к использованию ресурса, потом другой и т. д. Например, несколько программ хотят обратиться к центральному процессору. В этой ситуации операционная система сначала разрешает доступ к процессору одной программе, затем, после того как она поработала достаточное время, другой программе, затем следующей и, в конце концов, опять первой. Определение того, как долго ресурс будет использоваться во времени, кто будет следующим и на какое время ему предоставляется ресурс – это задача операционной системы. Другой вид распределения – это пространственное мультиплексирование. Вместо поочередной работы каждый клиент получает часть ресурса. Обычно оперативная память разделяется между несколькими работающими программами, так что все они одновременно могут постоянно находиться в памяти (например, используя центральный процессор по очереди). Если предположить, что памяти достаточно для того, чтобы хранить несколько программ, эффективнее разместить в памяти сразу несколько программ, чем выделить всю память одной программе, особенно если ей нужна лишь небольшая часть имеющейся памяти. Конечно, при этом возникают проблемы справедливого распределения, защиты памяти и т. д., и для разрешения подобных вопросов существует операционная система [14].

1.2 История развития операционных систем

Обычно историю развития операционных систем связывают с историей развития компьютеров. Первая идея компьютера была предложена английским математиком Чарльзом Бэббиджем (Charles Babbage) в середине девятнадцатого века. Им была разработана так называемая механическая «аналитическая машина», которая правда так и не заработала должным образом. Далее представлены поколения компьютеров и их связь с операционными системами.

Первое поколение 1945-1955

Компьютеры состояли из электронных ламп и коммутационных панелей. Наивысшее достижение – выпуск перфокарт. Сделанная из тонкого картона, перфокарта представляет информацию наличием или отсутствием отверстий в определённых позициях карты. Операционная система отсутствует.

Второе поколение 1955-1965

Основа компьютеров транзисторы и системы пакетной обработки. Характеризовались колодами перфокарт и устройствами для записывания магнитных лент. В основном программировали на языках Фортран и Ассемблер для операционной системы Fortran Monitor System (FMS) и IBSYS.

Третье поколение 1965-1980

Период характеризуется появлением интегральных микросхем, а также многозадачностью или, как её называют по другому, мультипрограммированием. Фирма IBM выпускает различные серии машин, начиная с IBM/360. Для них была написана операционная система OS/360, которая примерно в 1000 раз превышала по величине FMS второго поколения. На этом этапе появляется промышленная реализация многозадачности – способа организации вычислительного процесса, при котором в памяти компьютера находилось одновременно несколько программ,

попеременно выполняющихся на одном процессоре.

Другие известные операционные системы этого периода CTSS (совместимая система разделения времени) и MULTICS (мультиплексная информационная и вычислительная служба), которая была предназначена для обеспечения доступа сразу для сотни пользователей к одной машине. Дальнейшее развитие данной системы переросло в UNIX.

Четвёртое поколение 1980-наши дни

Этот период связан с появлением больших интегральных схем. В 1974 году компания Intel выпустила первый универсальный 8-разрядный процессор Intel 8080. В начале 80-х корпорация IBM разработала IBM PC – персональный компьютер. В тоже время появляется первая версия MS-DOS. Все разработанные до этого момента операционные системы поддерживали только текстовый режим общения с пользователем.

Первая попытка сделать дружественный графический интерфейс была реализована на Apple Macintosh. Под влиянием её успехов корпорация Microsoft выпускает графическую оболочку для MS-DOS – Windows. А с 1995 года вышла в свет Windows 95, которая стала автономной системой. В дальнейшем, на базе Windows 95 и другой системы Windows NT были разработаны существующие на настоящий момент операционные системы – Windows 2000, XP, Vista и другие.

1.3 Классификация операционных систем

Операционных систем очень много и не всем они известны. Далее рассмотрено 7 видов различных операционных систем по уровню от большого к малому.

Операционные системы мэйнфреймов

Мэйнфрейм – высокопроизводительный компьютер общего назначения со значительным объемом оперативной и внешней памяти, предназначенный для выполнения интенсивных вычислительных работ. Обычно это компьютеры размером с комнату и их нахождение – в крупных корпорациях. Обычно мэйнфреймы содержат тысячи дисков и терабайты оперативной памяти.

Операционные системы для мэйнфреймов в основном ориентированы на обработку множества одновременных заданий, большинству из которых требуются огромное количество операций ввода-вывода. Система должна отвечать на тысячи запросов в секунду. Примером является OS/390, произошедшая от операционной системы 3-го поколения OS/360.

Серверные операционные системы

Данные операционные системы работают на серверах, которые представляют из себя персональный компьютер, рабочую станцию или даже мэйнфрейм. Серверы предоставляют возможность работы с печатающими устройствами, файлами или Интернетом. К таким операционным системам относятся Unix, Linux, Windows 2003 Server и др.

Многопроцессорные операционные системы

Данные системы применяются на компьютерах с несколькими центральными процессорами. Для них требуются специальные операционные системы, но обычно

они представляют собой модификации серверных операционных систем.

Операционные системы для персональных компьютеров

Главный критерий этих систем – удобный интерфейс для одного пользователя. Наиболее известные системы: серии Windows 98, 2000, XP, Vista; Macintosh, Linux.

Операционные системы реального времени

Главный параметр этих систем – время. В системах управления промышленным процессом необходимо четко синхронизировать время работы конвейера, различных промышленных роботов. Это жесткая система реального времени. Есть и гибкие системы реального времени – в ней допустимы пропуски сроков выполнения операции, например мультимедийные системы. К операционным системам реального времени относятся VxWorks и QNX.

Встроенные операционные системы

К ним относятся операционные системы «карманных компьютеров» PDA (Personal Digital Assistant – персональный цифровой помощник). Кроме того, встроенные системы работают на машинах, в телевизорах, мобильных телефонах. В этих операционных системах обычно присутствуют все характеристики операционных систем реального времени с ограничением памяти, мощности и т.п. Примеры систем – PalmOS, Windows CE.

Операционные системы для смарт-карт

Смарт-карта – устройство размером с кредитную карту, содержащее центральный процессор. На такие системы налагаются жесткие ограничения по мощности и памяти. Некоторые управляют только одной операцией – электронным платежом к примеру. Отдельные смарт-карты включают в себя поддержку виртуальной машины Java.

1.4 Обзор аппаратного обеспечения компьютера

Операционная система тесно связана с оборудованием компьютера, на котором она должна работать. Аппаратное обеспечение влияет на набор команд операционной системы и управление его ресурсами. Концептуально простой компьютер можно представить в виде модели, показанной на рисунке 1 [14]. Такая структура использовалась на первых моделях IBM PC.

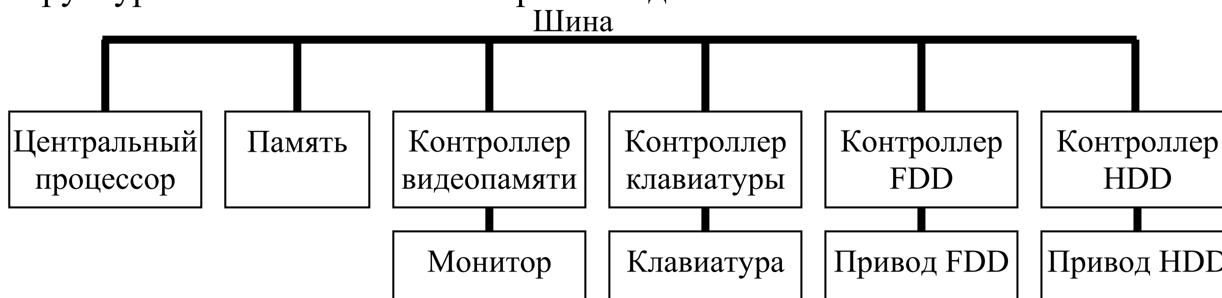

Рисунок 1 – Некоторые компоненты персонального компьютера

На рисунке центральный процессор, память, устройства ввода-вывода соединены системной шиной, по которой они обмениваются информацией.

Процессор

«Мозгом» компьютера является центральный процессор (CPU – Central Processing Unit). Он выбирает из памяти команды и выполняет их. Обычный цикл работы процессора выглядит так: читается первая команда из памяти, декодируется для определения ее типа и операндов, выполняет команду, затем считывает, декодирует последующие команды. Таким образом осуществляется выполнение программ.

Для каждого процессора существует набор команд, который он в состоянии выполнить. Поскольку доступ к памяти для получения команд или набор данных занимает намного больше времени, чем выполнение этих команд, то все процессоры содержат внутренние регистры для хранения переменных и промежуточных результатов. Поэтому набор инструкций обычно содержит команды для загрузки слова из памяти в регистр и сохранение слова из регистра в память. Кроме основных регистров, используемых для хранения переменных, большинство процессоров имеет несколько специальных регистров, используемых для хранения переменных, а также специальных регистров, видимых для программистов.

При временном мультиплексировании процессора операционная система останавливает работающую программу для запуска другой. Каждый раз при таком прерывании операционная система должна сохранять все регистры процессора, чтобы позже, когда прерванная программа продолжит свою работу, их можно было восстановить.

Для повышения быстродействия CPU их разработчики отказались от простой модели, когда за один такт может быть считана, декодирована, выполнена только одна команда. Современные процессоры обладают возможностью выполнения нескольких команд одновременно.

Большинство CPU имеет два режима работы: режим ядра и пользовательский режим. Если процессор запущен в режиме ядра, он может выполнять все команды из набора инструкций и использовать все возможности аппаратуры. Операционная система работает в режиме ядра, предоставляя доступ ко всему оборудованию. В противоположность этому, пользователи работают в пользовательском режиме, разрешающем выполнение подмножества программ и делающем доступным лишь часть аппаратных средств.

Память

Второй основной составляющей любого компьютера является память. В идеале память должна быть максимально быстрой (быстрее, чем обработка одной инструкции, чтобы работу процессора не замедляло обращение к памяти достаточно большой и чрезвычайно дешевой). На сегодня не существует технологий, удовлетворяющих всем этим требованиям. Поэтому имеется другой подход.

Система памяти конструируется в виде иерархии слоев [13], которые иллюстрируются на рисунке 2. По мере продвижение по иерархии сверху вниз возрастают два параметра: время доступа, объём памяти.

Верхний слой состоит из внутренних регистров CPU, поэтому при доступе к ним не возникает задержек. Внутренние регистры хранят менее 1Кб информации. Программы могут управлять регистрами без вмешательства аппаратуры. Доступ к регистрам быстрее всего – несколько наносекунд.

В следующем слое находится кэш-память, в основном контролируемая

аппаратурой. Наиболее часто используемые области кэша хранятся в высокоскоростной кэш-памяти, расположенной внутри центрального процессора. Когда программа должна прочитать слово из памяти, кэш-микросхема определяет, есть ли нужная строка в кэше; если это так, то происходит результативное обращение к кэш-памяти. Кэш-память ограничена в размере, что обусловлено ее высокой стоимостью. В современных машинах есть два или три уровня кэша, причем каждый последующий медленнее и больше предыдущего. Размеры кэш-памяти от десятков килобайт до нескольких мегабайт. Время доступа – несколько больше, чем к регистрам.

Рисунок 2 – Иерархическая структура памяти

Далее следует оперативная память ОЗУ (RAM – Random Acces Memory или память с произвольным доступом) – главная рабочая область запоминающего устройства машины. Все запросы CPU, которые не могут быть выполнены кэш-памятью, поступают для обработки в ОЗУ. Объёмы от сотен мегабайт до нескольких гигабайт. Время доступа – десятки наносекунд.

Следующим идёт магнитный диск. Дисковая память на два порядка дешевле ОЗУ в пересчете на бит и на два порядка больше по величине. У диска есть одна проблема – случайный доступ к данным на нем занимает примерно на три порядка больше времени. Причиной низкой скорости жестких дисков (HDD) является то, что диск представляет собой механическую конструкцию. Он состоит из одной или нескольких металлических пластин, вращающихся с определенными скоростями, например 7200 об/мин. Объёмы дисков сейчас стремительно растут, в продаже для большинства пользователей находятся диски с сотнями гигабайт. Время доступа – не менее 10мкс.

Магнитная лента часто используется для создания резервных копий HDD или для хранения очень больших наборов данных. Сейчас, конечно редко, где можно встретить применение магнитных лент, но всё же они ещё не вышли из употребления. К уровню магнитной ленты также можно отнести CD, DVD диски и флэш-память. Время доступа измеряется секундами.

Кроме описанных видов, в компьютерах есть небольшое количество постоянной памяти с произвольным доступом. В отличие от RAM, она не теряет

свое содержимое при выключении питания. Она называется ПЗУ или ROM. ПЗУ программируется в процессе производства и после этого его содержимое нельзя изменить. Эта память достаточно быстра и дешева. Программы начальной загрузки компьютера, используемые при запуске, находятся в ПЗУ. Кроме этого, некоторые карты ввода-вывода содержат ПЗУ для управления низкоуровневыми устройствами.

Вид памяти, называемый CMOS, является энергозависимым. CMOS используется для хранения текущей даты, времени и конфигурационных параметров, например, указания, с какого жесткого диска производить загрузку. Эта память потребляет энергию от установленного аккумулятора.

Устройства ввода-вывода

Операционная система взаимодействует с устройствами ввода-вывода как с ресурсами. Устройства ввода-вывода обычно состоят из контроллера и самого устройства.

Контроллер – набор микросхем на вставляемой в разъем плате, физически управляющее устройство. Он принимает команды операционной системы (например, указания прочитать данные с устройства) и выполняет их. Фактическое управление устройством очень сложно и требует высокого уровня детализации. Поэтому в функции контроллера входит представление простого интерфейса для операционной системы.

Следующей частью является само устройство. Устройства имеют достаточно простые интерфейсы, потому что их возможности невелики и их нужно привести к единому стандарту. Единый стандарт необходим, например чтобы каждый IDE контроллер диска (Integrated Drive Electronics) мог управлять любым IDE диском. IDE интерфейс является стандартным для дисков на компьютерах с процессором Pentium, а также на других компьютерах. Так как настоящий интерфейс устройства скрыт с помощью контроллера, то операционная система видит только интерфейс контроллера, который может сильно отличаться от интерфейса самого устройства.

Поскольку все виды контроллеров отличаются, то для них требуется разное программное обеспечение. Программа, которая общается с контроллером, – драйвер устройства. Каждый производитель контроллеров должен поставлять драйверы для поддерживаемых операционных систем. Для использования драйвера его нужно установить в операционную систему так, чтобы он мог работать в режиме ядра. Есть три способа установки драйвера в ядро [14]:

- заново скомпоновать ядро вместе с новым драйвером и затем перезагрузить операционную систему (так работает множество операционных систем Unix);
- создать запись во входящем в операционную систему файле, говорящую о том, что требуется драйвер и затем перезагрузить систему; во время начальной загрузки операционная система сама находит нужные драйверы и загружает их (так работает Windows);
- операционная система может принимать новые драйверы, не прерывая работы, и оперативно устанавливать их, не нуждаясь в перезагрузке. Этот способ становится все более и более распространенным. Такие устройства как шины USB, IEEE 1394 всегда нуждаются в динамически загружаемых драйверах.

Ввод-вывод данных можно осуществлять тремя различными способами [14].

- Простейший способ: пользовательская программа выдает системный запрос, который ядро транслирует в вызов процедуры, соответствующей драйверу, затем драйвер начинает процесс ввода-вывода. В этом время он выполняет короткий программный цикл, постоянно опрашивая устройство, с которым он работает. При завершении операций ввода-вывода драйвер помещает данные туда, куда требуется, и возвращается в исходное состояние. Затем операционная система возвращает управление программе, осуществлявшей вызов. Этот метод – ожидание готовности (активное ожидание). Он имеет один недостаток: процессор должен опрашивать устройство, пока оно не завершит работу.
- Драйвер запускает устройство и просит его выдать прерывания по окончании ввода-вывода; после этого драйвер возвращает управление операционной системе, и она начинает выполнять другие задания. Когда контроллер обнаруживает окончание передачи данных, он генерирует прерывание о завершении операции. Процесс ввода-вывода, использующий прерывания, состоит из четырех шагов (Рисунок 3). На первом шаге драйвер передает команду контроллеру, записывая информацию в регистры устройств. Затем контроллер запускает устройство. Когда контроллер заканчивает чтение или запись того количества байтов, которое ему было указано передать, он посылает сигнал микросхеме контроллера прерываний, используя определенные провода шины. Это шаг второй. На третьем шаге если контроллер прерываний готов к обработке прерываний, то он подает сигнал на определенный контакт CPU, информируя его таким образом. На четвертом шаге контроллер прерываний вставляет номер устройства на шину, чтобы центральный процессор мог узнать, какое устройство завершило работу.
- Третий метод ввода-вывода информации заключается в использовании специального контролера прямого доступа к памяти DMA (Direct Memory Access). DMA управляет потоком битов между оперативной памятью и некоторыми контролерами без вмешательства CPU. Процессор обращается к микросхеме DMA, сообщает ей число байтов для передачи, а также адрес устройства и памяти, направление передачи данных. По завершении работы DMA инициирует прерывание, которое обрабатывается обычным порядком.

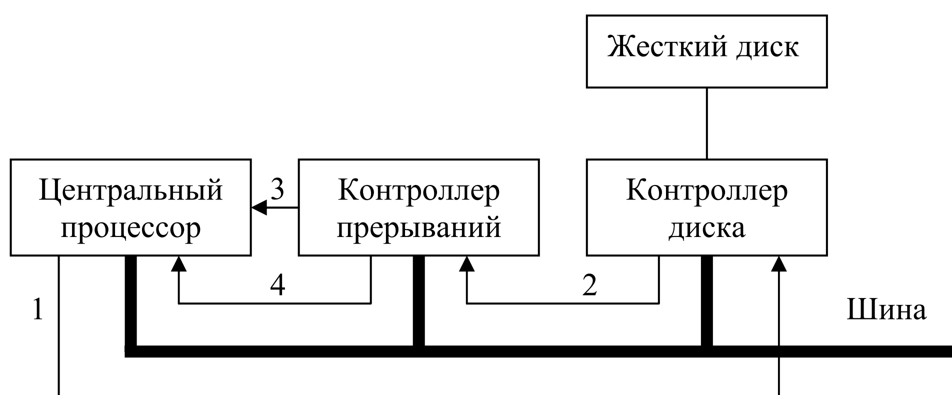

Рисунок 3 – Действия, выполняемые при запуске устройства ввода-вывода и получении прерывания

Шины

Из-за роста быстродействия процессора и памяти, в систему добавились дополнительные шины как для ускорения общения устройств ввода-вывода, так и для пересылки данных между процессором и памятью. На рисунке 4 приведена схема вычислительной системы первых Pentium.

В этой системе 8 шин (шина кэша, локальная шина, шина памяти, PCI, SCSI, USB, IDE, ISA), каждая со своей скоростью передачи данных и своими функциями. В операционной системе для управления компьютером должны находиться сведения обо всех этих шинах.

Центральный процессор по локальной шине передает данные микросхеме PCI-моста, – который в свою очередь обращается к памяти по выделенной шине. Система Pentium I имеет кэш первого уровня (L1) встроенный в процессор и намного больший кэш второго уровня (L2), подключенный к процессору отдельной шиной кэша. Шина IDE служит для присоединения периферийных устройств к системе (CD-ROM, жесткий диск).

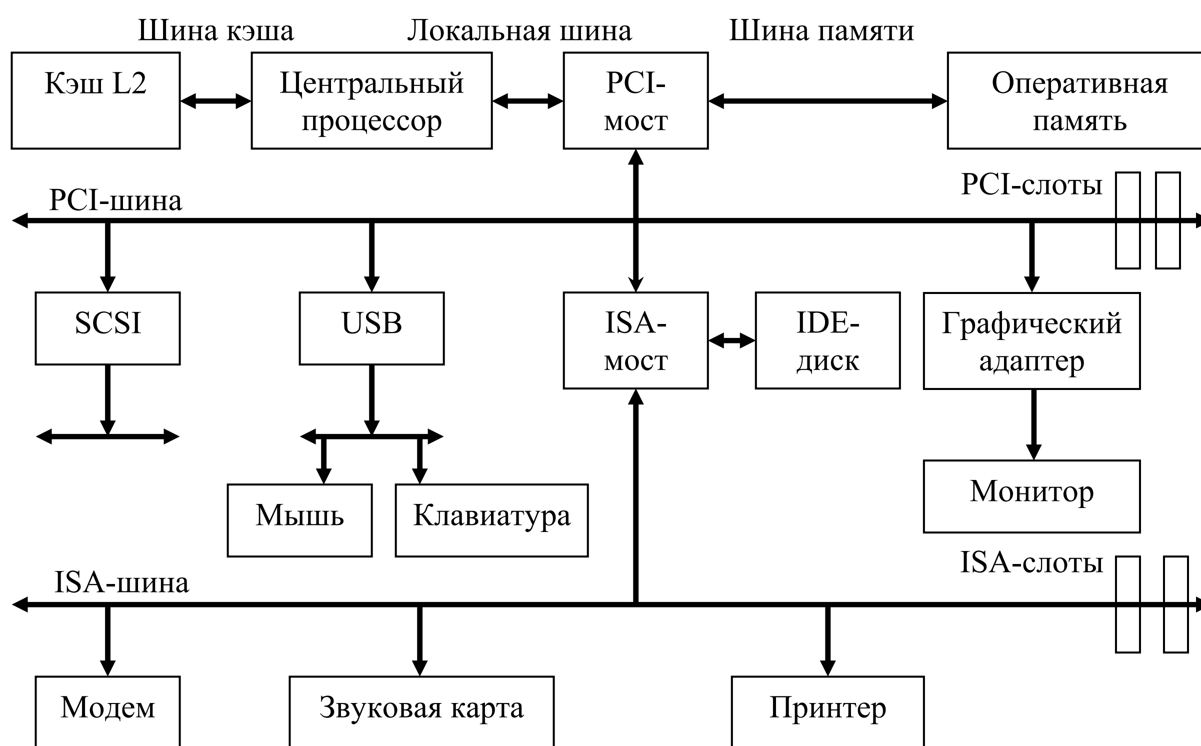

Рисунок 4 – Структура системы Pentium

Шина USB (Universal Serial Bus) предназначена для присоединения к компьютеру дополнительных устройств ввода-вывода, таких как клавиатура, мышь, принтер, флэш-память и т.д. С течением времени появляются и добавляются новые более быстрые шины.

1.5 Архитектура операционной системы

Единой архитектуры операционных систем не существует, но существуют универсальные подходы к их структурированию. Ниже дано описание двух

архитектур операционных систем, выполненное по книге Олифера В.Г., Олифера Н.А. «Сетевые операционные системы» [11].

1.5.1 Классическая архитектура

Наиболее общим подходом к структуризации операционной системы является разделение всех ее модулей на две группы:

- ядро – модули, выполняющие основные функции операционной системы;
- модули, выполняющие вспомогательные функции операционной системы.

Модули ядра выполняют такие базовые функции операционной системы, как управление процессами, памятью, устройствами ввода-вывода и т.п. Ядро составляет сердцевину операционной системы, без него она является полностью неработоспособной и не сможет выполнить ни одну из своих функций.

В состав ядра входят функции, решающие внутрисистемные задачи организации вычислительного процесса, такие как переключение контекстов, загрузка/выгрузка станиц, обработка прерываний. Эти функции недоступны для приложений. Другой класс функций ядра служит для поддержки приложений, создавая для них так называемую прикладную программную среду. Приложения могут обращаться к ядру с запросами – системными вызовами – для выполнения тех или иных действий, например для открытия и чтения файла, вывода графической информации на дисплей, получения системного времени и т. д. Функции ядра, которые могут вызываться приложениями, образуют интерфейс прикладного программирования – API.

Функции, выполняемые модулями ядра, являются наиболее часто используемыми функциями операционной системы, поэтому скорость их выполнения определяет производительность всей системы в целом. Для обеспечения высокой скорости работы операционной системы все модули ядра или большая их часть постоянно находятся в оперативной памяти, то есть являются резидентными.

Некоторые компоненты операционной системы оформлены как обычные приложения, то есть в виде исполняемых модулей стандартного для данной операционной системой формата, поэтому очень сложно провести четкую грань между операционной системой и приложениями.

Вспомогательные модули операционной системы обычно подразделяются на следующие группы:

- утилиты – программы, решающие отдельные задачи управления и сопровождения компьютерной системы, такие, например, как программы сжатия дисков, архивирования данных на магнитную ленту;
- системные обрабатывающие программы – текстовые или графические редакторы, компиляторы, компоновщики, отладчики;
- программы предоставления пользователю дополнительных услуг – специальный вариант пользовательского интерфейса, калькулятор и даже игры;

- библиотеки процедур различного назначения, упрощающие разработку приложений, например библиотека математических функций, функций ввода-вывода и т. д.

Для надежного управления ходом выполнения приложений операционная система должна иметь по отношению к приложениям определенные привилегии. Иначе некорректно работающее приложение может вмешаться в работу системы и, например, разрушить часть ее кодов. Обеспечить привилегии операционной системе невозможно без специальных средств аппаратной поддержки. Аппаратура компьютера должна поддерживать как минимум два режима работы — пользовательский режим (user mode) и привилегированный режим, который также называют режимом ядра (kernel mode). На рисунке 5 представлено такое разделение режимов.

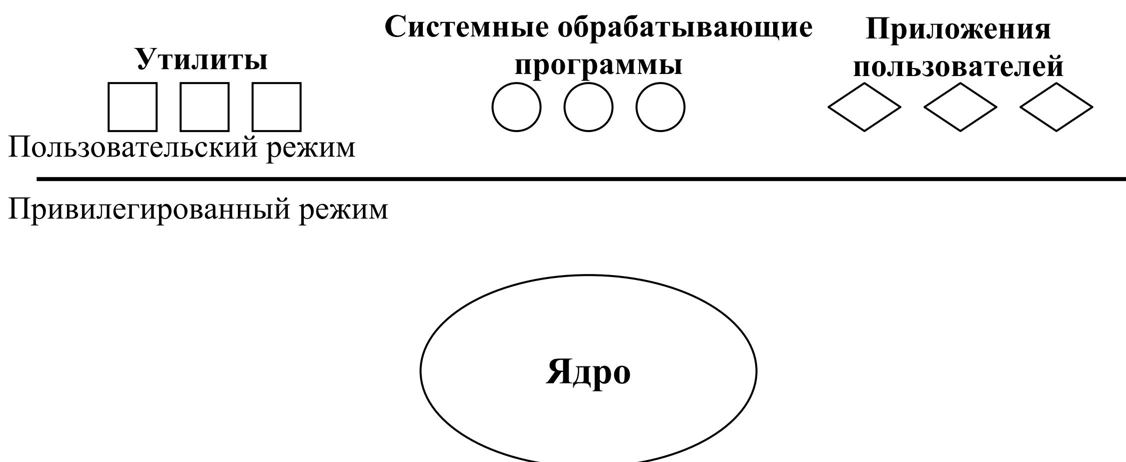

Рисунок 5 – Архитектура операционной системы с ядром в привилегированном режиме

Приложения ставятся в подчиненное положение за счет запрета выполнения в пользовательском режиме некоторых критичных команд, связанных с переключением процессора с задачи на задачу, управлением устройствами ввода-вывода, доступом к механизмам распределения и защиты памяти.

Уровней привилегий может быть несколько – 2, 3, 4 и т.д. Между количеством уровней привилегий, реализуемых аппаратно, и количеством уровней привилегий, поддерживаемых операционной системой, нет прямого соответствия. Так, на базе четырех уровней, обеспечиваемых процессорами компании Intel, операционная система OS/2 строит трехуровневую систему привилегий, а операционные системы Windows NT, UNIX и некоторые другие ограничиваются двухуровневой системой.

Повышение устойчивости операционной системы, обеспечиваемое переходом ядра в привилегированный режим, достигается за счет некоторого замедления выполнения системных вызовов. Системный вызов привилегированного ядра инициирует переключение процессора из пользовательского режима в привилегированный, а при возврате к приложению – переключение из привилегированного режима в пользовательский (Рисунок 6). Во всех типах процессоров из-за дополнительной двукратной задержки переключения переход на процедуру со сменой режима выполняется медленнее, чем вызов процедуры без

смены режима.

Вычислительную систему, работающую под управлением операционной системы на основе ядра, можно рассматривать как систему, состоящую из трех иерархически расположенных слоев: нижний слой образует аппаратура, промежуточный – ядро, а утилиты, обрабатывающие программы и приложения, составляют верхний слой системы Каждый слой обслуживает вышележащий слой, выполняя для него некоторый набор функций, которые образуют межслойный интерфейс.

Поскольку ядро представляет собой сложный многофункциональный комплекс, то многослойный подход обычно распространяется и на структуру ядра.

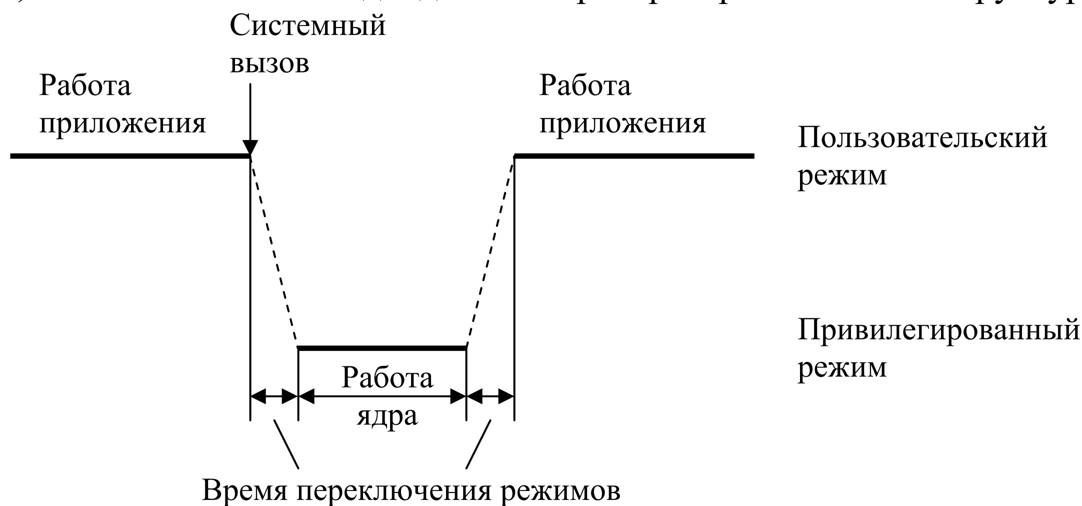

Рисунок 6 – Смена режимов при выполнении системного вызова к привилегированному ядру

Ядро может состоять из следующих слоев.

- *Средства аппаратной поддержки операционной системы.* К операционной системе относят, естественно, не все аппаратные устройства компьютера, а только средства её аппаратной поддержки, то есть те, которые прямо участвуют в организации вычислительных процессов: средства поддержки привилегированного режима, систему прерываний, средства переключения контекстов процессов, средства защиты областей памяти и т.п.
- *Машинно-зависимые компоненты операционной системы.* Этот слой образуют программные модули, в которых отражается специфика аппаратной платформы компьютера. В идеале этот слой полностью экранирует вышележащие слои ядра от особенностей аппаратуры.
- *Базовые механизмы ядра.* Этот слой выполняет наиболее примитивные операции ядра, такие как программное переключение контекстов процессов, диспетчеризацию прерываний, перемещение страниц из памяти на диск и обратно и т. п.
- *Менеджеры ресурсов.* Этот слой состоит из мощных функциональных модулей, реализующих стратегические задачи по управлению основными ресурсами вычислительной системы. Обычно на данном слое работают менеджеры (называемые также диспетчерами) процессов, ввода-вывода, файловой системы и

оперативной памяти.

- *Интерфейс системных вызовов.* Этот слой является самым верхним слоем ядра и взаимодействует непосредственно с приложениями и системными утилитами, образуя прикладной программный интерфейс операционной системы. Функции API, обслуживающие системные вызовы, предоставляют доступ к ресурсам системы в удобной и компактной форме, без указания деталей их физического расположения.

Приведенное разбиение ядра операционной системы на слои является достаточно условным. В реальной системе количество слоев и распределение функций между ними может быть и иным.

Архитектура операционной системы, основанная на привилегированном ядре и приложениях пользовательского режима, стала, по существу, классической. Ее используют многие популярные операционные системы, в том числе многочисленные версии UNIX, IBM OS/390, OS/2, и с определенными модификациями – Windows NT.

1.5.2 Микроядерная архитектура

Микроядерная архитектура является альтернативой классическому способу построения операционной системы. Суть микроядерной архитектуры состоит в следующем. В привилегированном режиме остается работать только очень небольшая часть операционной системы, называемая микроядром (Рисунок 7). Микроядро защищено от остальных частей операционной системы и приложений. В состав микроядра обычно входят машинно-зависимые модули, а также модули, выполняющие базовые (но не все) функции ядра по управлению процессами, обработке прерываний, управлению виртуальной памятью, пересылке сообщений и управлению устройствами ввода-вывода, связанные с загрузкой или чтением регистров устройств. Набор функций микроядра обычно соответствует функциям слоя базовых механизмов обычного ядра. Такие функции операционной системы трудно, если не невозможно, выполнить в пространстве пользователя.

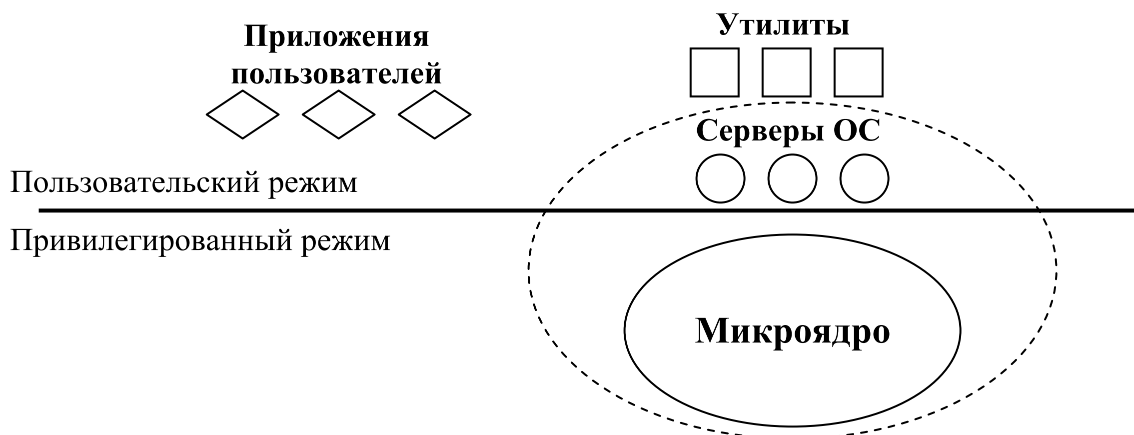

Рисунок 7 – Перенос основного объема функций ядра в пользовательское пространство

Все остальные более высокоуровневые функции ядра оформляются в виде приложений, работающих в пользовательском режиме. Однозначного решения о том, какие из системных функций нужно оставить в привилегированном режиме, а какие перенести в пользовательский, не существует. В общем случае многие менеджеры ресурсов, являющиеся неотъемлемыми частями обычного ядра – файловая система, подсистемы управления виртуальной памятью и процессами, менеджер безопасности и т. п. – становятся «периферийными» модулями, работающими в пользовательском режиме.

Менеджеры ресурсов, вынесенные в пользовательский режим, называются серверами операционной системы, то есть модулями, основным назначением которых является обслуживание запросов локальных приложений и других модулей операционной системы. Очевидно, что для реализации микроядерной архитектуры необходимым условием является наличие в операционной системе удобного и эффективного способа вызова процедур одного процесса из другого. Поддержка такого механизма и является одной из главных задач микроядра.

Схематично механизм обращения к функциям операционной системы, оформленным в виде серверов, выглядит следующим образом (Рисунок 8). Клиент, которым может быть либо прикладная программа, либо другой компонент операционной системы, запрашивает выполнение некоторой функции у соответствующего сервера, посылая ему сообщение. Непосредственная передача сообщений между приложениями невозможна, так как их адресные пространства изолированы друг от друга. Микроядро, выполняющееся в привилегированном режиме, имеет доступ к адресным пространствам каждого из этих приложений и поэтому может работать в качестве посредника. Микроядро сначала передает сообщение, содержащее имя и параметры вызываемой процедуры нужному серверу, затем сервер выполняет запрошенную операцию, после чего ядро возвращает результаты клиенту с помощью другого сообщения. Таким образом, работа микроядерной операционной системы соответствует известной модели клиент-сервер, в которой роль транспортных средств выполняет микроядро.

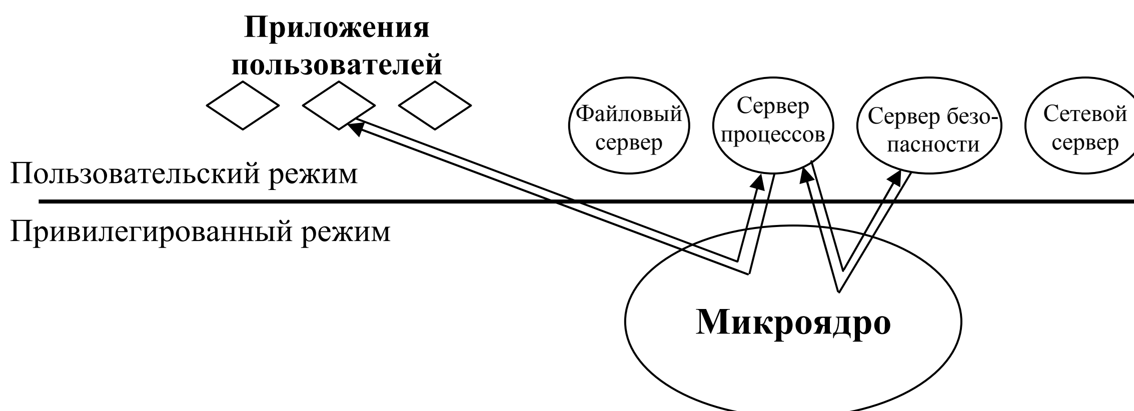

Рисунок 8 – Реализация системного вызова в микроядерной архитектуре

Достоинства микроядерной архитектуры:

1 *Переносимость.* Высокая степень переносимости обусловлена тем, что весь машинно-зависимый код изолирован в микроядре, поэтому для переноса

системы на новый процессор требуется меньше изменений и все они логически сгруппированы вместе.

2 *Расширяемость* присуща микроядерной операционной системе в очень высокой степени.

3 *Конфигурируемость.* При микроядерном подходе конфигурируемость операционной системы не вызывает никаких проблем и не требует особых мер – достаточно изменить файл с настройками начальной конфигурации системы или же остановить не нужные больше серверы в ходе работы обычными для остановки приложений средствами.

4 *Надежность*. Использование микроядерной модели повышает надежность системы. Каждый сервер выполняется в виде отдельного процесса в своей собственной области памяти и таким образом защищен от других серверов операционной системы, что не наблюдается в традиционной операционной системе, где все модули ядра могут влиять друг на друга.

5 Модель с микроядром хорошо подходит для поддержки *распределенных вычислений,* так как использует механизмы, аналогичные сетевым: взаимодействие клиентов и серверов путем обмена сообщениями.

К основному и очень существенному недостатку относится низкая производительность операционной системы микроядерного типа. При классической организации операционной системы выполнение системного вызова сопровождается двумя переключениями режимов, а при микроядерной организации – четырьмя (Рисунок 9).

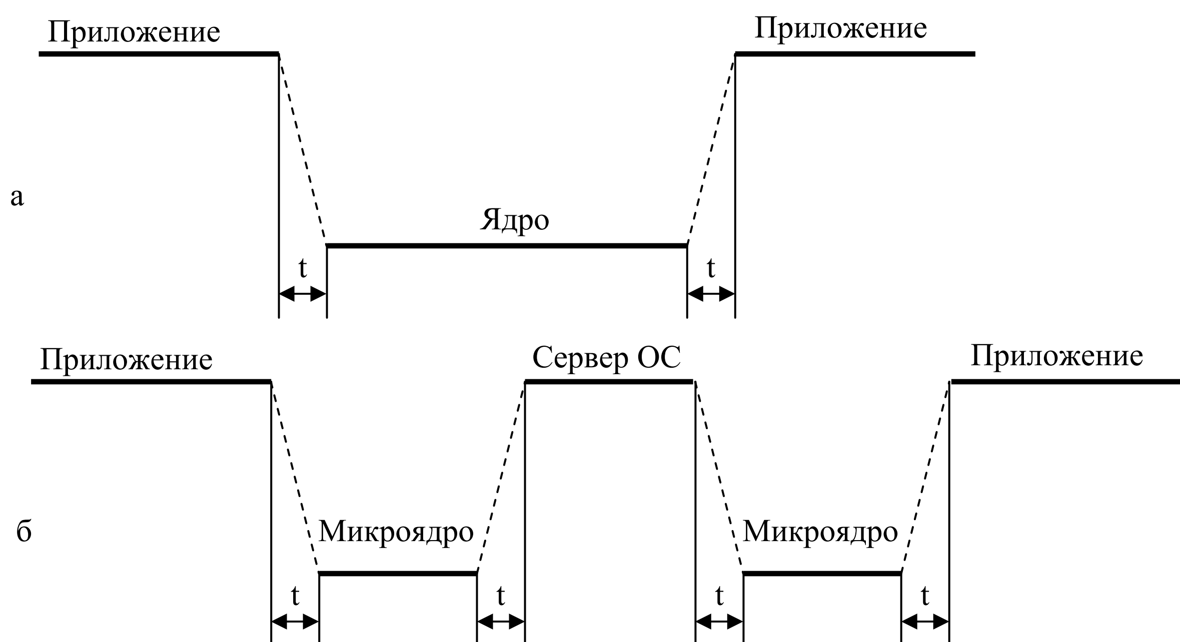

Рисунок 9 – Смена режимов при выполнении системного вызова: в классической архитектуре (а); в микроядерной (б)

Таким образом, операционная система на основе микроядра при прочих равных условиях всегда будет менее производительной, чем система с классическим ядром. Именно по этой причине микроядерный подход не получил такого широкого распространения, которое ему предрекали. Примером

микроядерной системы является VM/370, использующаяся в мейнфреймах.

Однако на настоящий момент не существует операционных систем с чисто классической или микроядерной архитектурой. В результате операционные системы образуют некоторый спектр, на одном краю которого находятся системы с минимально возможным микроядром, а на другом – системы, в которых микроядро выполняет достаточно большой объем функций.

Контрольные вопросы по разделу

1 Каковы две главные функции операционной системы?

2 Что такое многозадачность?

3 Перечислите основные различия между операционной системой для персонального компьютера и для мэйнфрейма.

4 Какие из приведенных ниже терминов являются синонимами? привилегированный режим; защищенный режим; режим супервизора; пользовательский режим; реальный режим; режим ядра.

5 В чем состоят отличия в работе процессора в привилегированном и пользовательском режимах?

6 Какими этапами отличается выполнение системного вызова в микроядерной операционной системе и системе с монолитным ядром?

7 В чем состоят современные тенденции развития операционных систем?

8 Каковы преимущества и недостатки микроядерной архитектуры?

9 Для чего служат менеджеры ресурсов?

10 Кем и на какой операционной системе был впервые опробован дружественный графический интерфейс?

2 Процессы и потоки

2.1 Процессы

В многозадачной системе процессор переключается между программами, предоставляя каждой от десятков до сотен миллисекунд. В каждый конкретный момент времени процессор работает только с одной программой, создавая иллюзию параллельной работы, т.е. псевдопараллелизм [14]. Настоящая параллельная работа присутствует в многопроцессорных и многоядерных системах, таких как Core 2 Duo. Следить за работой параллельно идущих процессов достаточно трудно, поэтому со временем разработчики операционных систем создали концептуальную модель последовательных процессов, упрощающую эту работу.

В этой модели все функционирующее на компьютере программное обеспечение организовано в виде набора последовательных процессов. С позиции модели у каждого процесса есть собственный виртуальный центральный процессор.

На рисунке 10, а представлена схема компьютера, работающего с 4 программами. На рисунке 10, б представлены 4 процесса каждый со своим логическим счетчиком команд, идущие независимо друг от друга. На самом деле существует только один физический счетчик команд, который загружается и сохраняется при переключении процессов. На рисунке 10, в видно, что за достаточно большой промежуток времени изменилось состояние всех 4 процессов.

Поскольку процессор переключается между программами, скорость, с которой процессор производит свои вычисления, будет непостоянной и, возможно, даже будет отличной при каждом новом запуске программы.

Существует четыре основных события, приводящие к созданию процессов:

- инициализация системы;
- выполнение изданного работающим процессом системного запроса на создание процесса;
- запрос пользователя на создание процесса;
- инициирование пакетного задания.

Программист для создания процесса в UNIX должен вызвать комбинацию из двух функций fork и execve, а в Windows – CreateProcess [12].

Процесс может завершиться благодаря одному из следующих действий:

- обычный выход (преднамеренно);
- выход по ошибке (преднамеренно);
- выход по неисправимой ошибке (непреднамеренно);
- уничтожение другим процессом (непреднамеренно).

Для завершения процесса программист в UNIX должен вызвать системный запрос kill, соответствующая функция в Win32 API – TerminateProcess.

Основным отличием структуры процессов в Windows и UNIX является связь между родительским и дочерним процессами. Так в UNIX существует иерархия процессов, а в Windows все процессы равноправны. Единственное, в чем проявляется что-то вроде иерархии процессов в Windows – создание процесса, в котором родительский процесс получает специальный маркер (так называемый

дескриптор), позволяющий контролировать дочерний процесс. Но маркер можно передать другому процессу, нарушая иерархию.

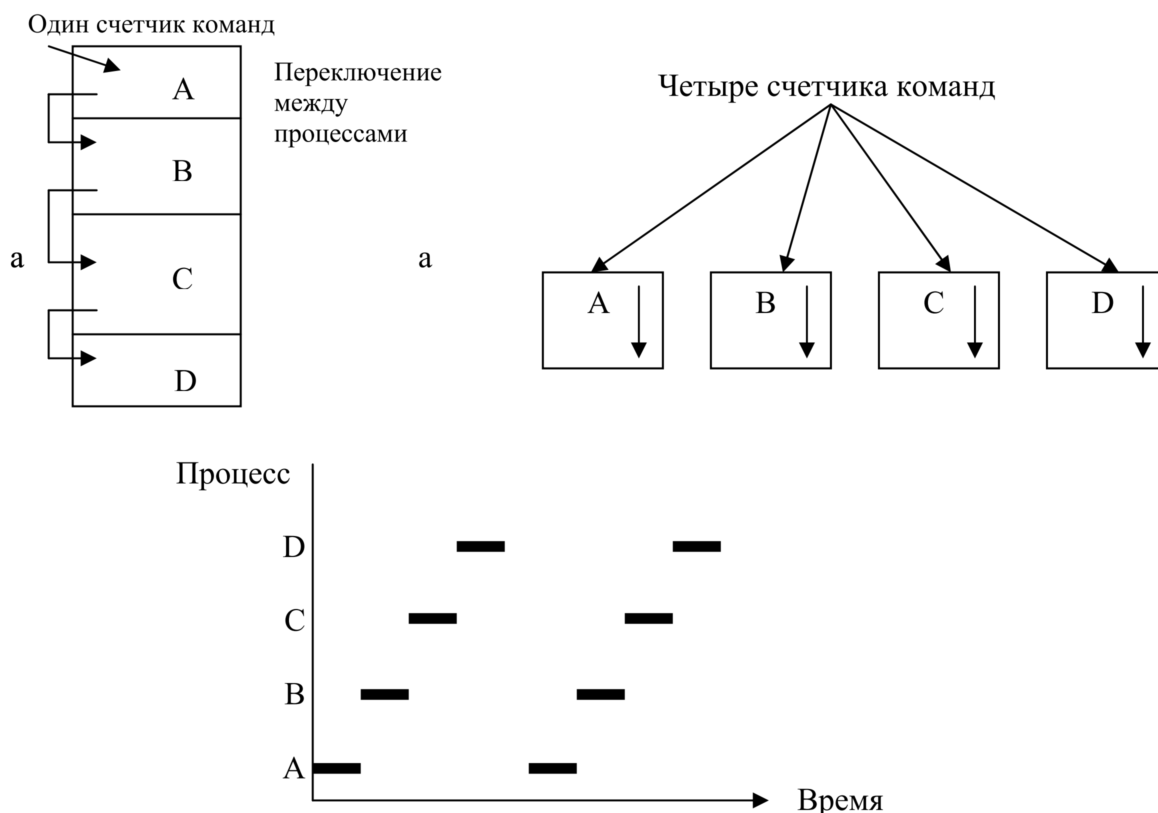

Рисунок 10 – 4 программы в многозадачном режиме (а); модель 4 независимых последовательных процессов (б); в каждый момент времени активна только одна программа (в)

Процесс может находиться в 3 возможных состояниях (Рисунок 11):

- работающий (в конкретный момент времени использующий процессор);
- готовый к работе (процесс временно приостановлен, чтобы позволить выполняться другому процессу);
- заблокированный (процесс не может быть запущен прежде, чем произойдёт некое внешнее событие).

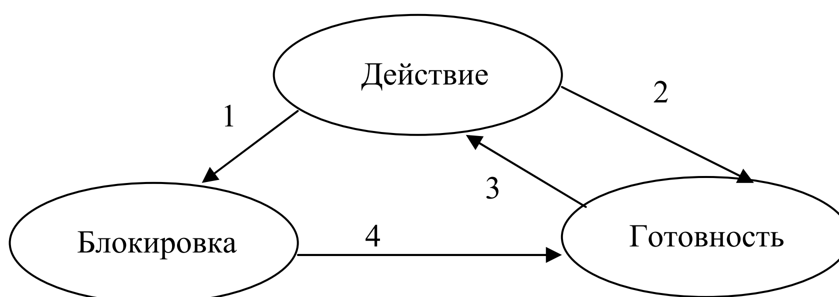

Рисунок 11 – Процесс может находиться в рабочем, готовом и заблокированном состоянии

Переходы между состояниями:

1) процесс блокируется, ожидая входных данных;

2) планировщик выбирает другой процесс;

3) планировщик выбирает этот процесс;

4) доступны входные данные.

Переход 1 происходит, когда процесс обнаруживает, что продолжение работы невозможно. Переходы 2 и 3 вызываются частью операционной системы, называемой планировщиком процессов, так что сами процессы даже не знают о существовании этих переходов. Переход 4 происходит с появлением внешнего события, ожидавшегося процессом (например, прибытие входных данных).

Для реализации модели процессов операционная система содержит таблицу (массив структур), называемую таблицей процессов, с одним элементом для каждого процесса. Элемент таблицы содержит информацию о состоянии процесса, счетчике команд, указателе стека, распределении памяти, состоянии открытых файлов, об использовании и распределении ресурсов, а также всю остальную информацию, которую необходимо сохранять при переключении в состояние готовности или блокировки для последующего запуска – как если бы процесс не останавливался. В таблице 1 представлены некоторые типичные элементы таблицы процессов.

Таблица 1 – Некоторые поля типичного элемента таблицы процессов

Управление процессом	Управление памятью	Управление файлами
Регистры Счетчик команд Слово состояния программы Указатель стека Состояние процесса Приоритет Параметры планирования Идентификатор процесса Родительский процесс Группа процесса Сигналы Время начала процесса Использованное процессорное время Процессорное время дочернего процесса	Указатель на текстовый сегмент Указатель на сегмент данных Указатель на сегмент стека	Корневой каталог Рабочий каталог Дескрипторы файла Идентификатор пользователя Идентификатор группы

Большое значение для создания иллюзии многопоточности на компьютерах с одним процессором имеет значение понятия прерывания. Прерывание (англ. interrupt) – сигнал, сообщающий процессору о совершении какого-либо асинхронного события [14]. При этом выполнение текущей последовательности команд приостанавливается, и управление передаётся обработчику прерывания, который выполняет работу по обработке события и возвращает управление в прерванный код.

Понятия программы и процесса отличаются друг от друга. Программа представляет собой статический набор команд, а процесс это набор ресурсов и данных, использующихся при выполнении программы. Процесс в Windows состоит

из следующих компонентов:

- структура данных, содержащая всю информацию о процессе;
- адресное пространство – диапазон адресов виртуальной памяти, которым может пользоваться процесс;
- исполняемая программа и данные, проецируемые на виртуальное адресное пространство процесса.

2.2 Потоки

Далее необходимо уяснить отличие между процессом и потоком. Процесс представляет собой объект, которому принадлежат ресурсы приложения. А поток (или нить) – это независимый путь выполнения внутри процесса, разделяющий вместе с процессом общее адресное пространство, код и глобальные данные. У каждого потока имеются собственные регистры, стек и механизмы ввода, в том числе очередь скрытых сообщений. Для описания использования нескольких потоков в одном процессе используется термин многопоточность.

В отличие от различных процессов, которые могут быть инициированы различными пользователями и преследовать несовместимые цели, один процесс всегда запущен одним пользователем, и потоки созданы таким образом, чтобы работать совместно, не мешая друг другу. Как показано в таблице 2, потоки разделяют не только адресное пространство, но и открытые файлы, дочерние процессы, сигналы и т. п.

Первая колонка содержит элементы, являющиеся свойствами процесса, а не потока. Например, если один поток открывает файл, этот файл тут же становится видимым для остальных потоков, и они могут считывать информацию и записывать ее в файл. Также как и процесс, поток может находиться в одном из нескольких состояний. Переходы между состояниями потоков такие же, как на рисунке 11.

У каждого потока свой собственный стек. Стек (англ. stack – стопка) – структура данных с методом доступа к элементам LIFO (англ. Last In – First Out, «последним пришел – первым вышел») [14].

В качестве примера использования нескольких потоков в одном процессе, можно привести ситуацию, когда приложению нужно записать большой файл на диск. При использовании одного потока – доступ к другим функциям программы будет недоступен до окончания операции.

Таблица 2 – Элементы процесса, общие для потоков, и индивидуальные элементы потоков

Элементы процесса	Элементы потока
Адресное пространство Глобальные переменные Открытые файлы Дочерние процессы Необработанные аварийные сигналы Сигналы и их обработчики Информация об использовании ресурсов	Счетчик команд Регистры Стек Состояние

Преимущества использования нескольких потоков перед несколькими процессами:

- возможность совместного использования параллельными объектами адресного пространства и всех содержащихся в нём данных;
- создание и уничтожение потоков происходит в примерно в 100 раз быстрее, чем для процессов;
- увеличивается производительность.

Есть два основных способа реализации пакета потоков: в пространстве пользователя и в ядре (Рисунок 12). В первом случае ядро ничего не знает о потоках и управляет обычными однопоточными процессами. Преимущество этого способа состоит в том, что его можно реализовать даже в операционных системах, не поддерживающих потоки. Раньше именно так все операционные системы и строились. Другое преимущество – это более высокая производительность по отношению ко второму способу и возможность использовать процессом собственный алгоритм планирования.

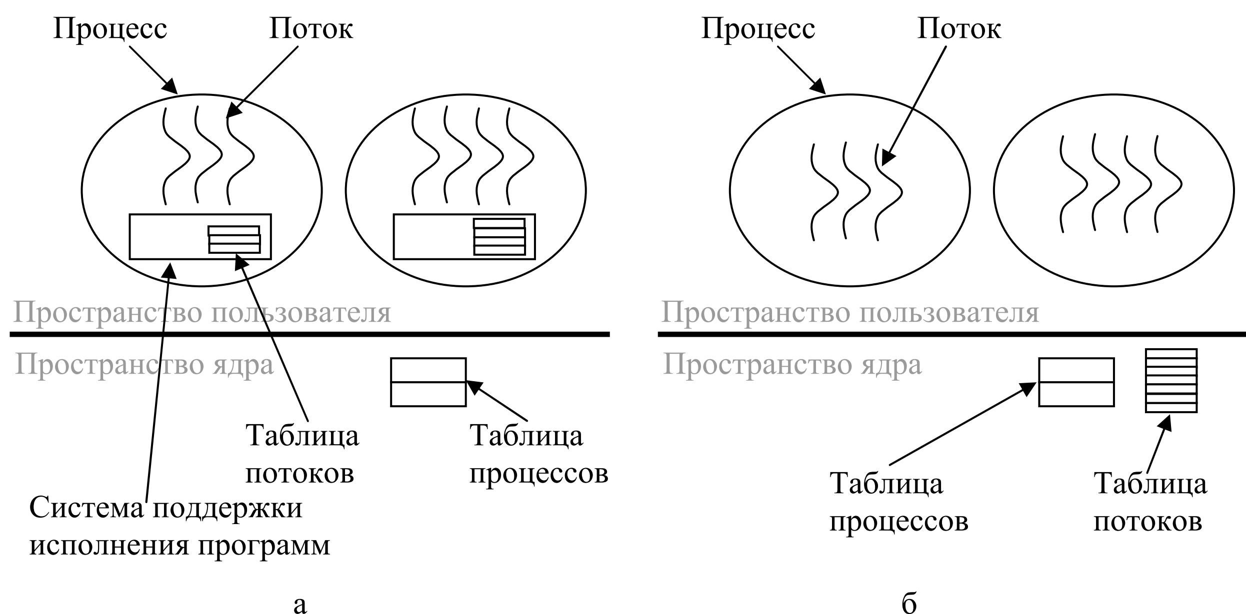

Рисунок 12 – Пакет потоков в пространстве пользователя (а); пакет потоков, управляемый ядром (б)

Однако, у первого способа есть серьёзные недостатки по отношению со вторым, например проблема добровольной отдачи процессора одним из потоков, или блокирование одного потока, что приводит к блокированию всего процесса. Поэтому на настоящий момент в большинстве известных ОС потоки реализуются в ядре или используется смешанное использование обоих способов.

2.3 Межпроцессное взаимодействие

Процессам часто бывает необходимо взаимодействовать между собой. Поэтому необходимо правильно организованное взаимодействие между процессами, по возможности не использующее прерываний. Проблема межпроцессного взаимодействия разбивается на 3 пункта [14]:

- передача информации от одного процесса другому;
- контроль над деятельностью процессов (к примеру, гарантии, что два процесса не пересекутся в критических ситуациях);
- согласование действий процессов (к примеру, если один процесс ожидает действий второго процесса, чтобы в свою очередь произвести некие действия).

Эти же пункты, не считая первого, относятся и к потокам.

Важным понятие в проблеме межпроцессного взаимодействия является состояние состязания – ситуация, в которой два или более процесса считывают и записывают данные одновременно и конечный результат зависит от того, какой из них был первым. Для предотвращения такого состояния и любой другой ситуации, связанной с совместным использованием памяти, файлов и чего-либо ещё, используется взаимное исключение – запрет одновременной записи и чтения разделенных данных более чем одним процессом.

Часть программы, в которой есть обращение к совместно используемым данным, называется критической областью или секцией. Несмотря на то, что это требование исключает состязание, его недостаточно для правильной совместной работы параллельных процессов и эффективного использования общих данных. Для этого необходимо выполнение 4 условий:

- два процесса не должны одновременно находиться в критических областях;
- в программе не должно быть предположений о скорости и количестве процессоров;
- процесс, находящийся вне критической области, не может блокировать другие процессы;
- невозможна ситуация, в которой процесс вечно ждет попадания в критическую область.

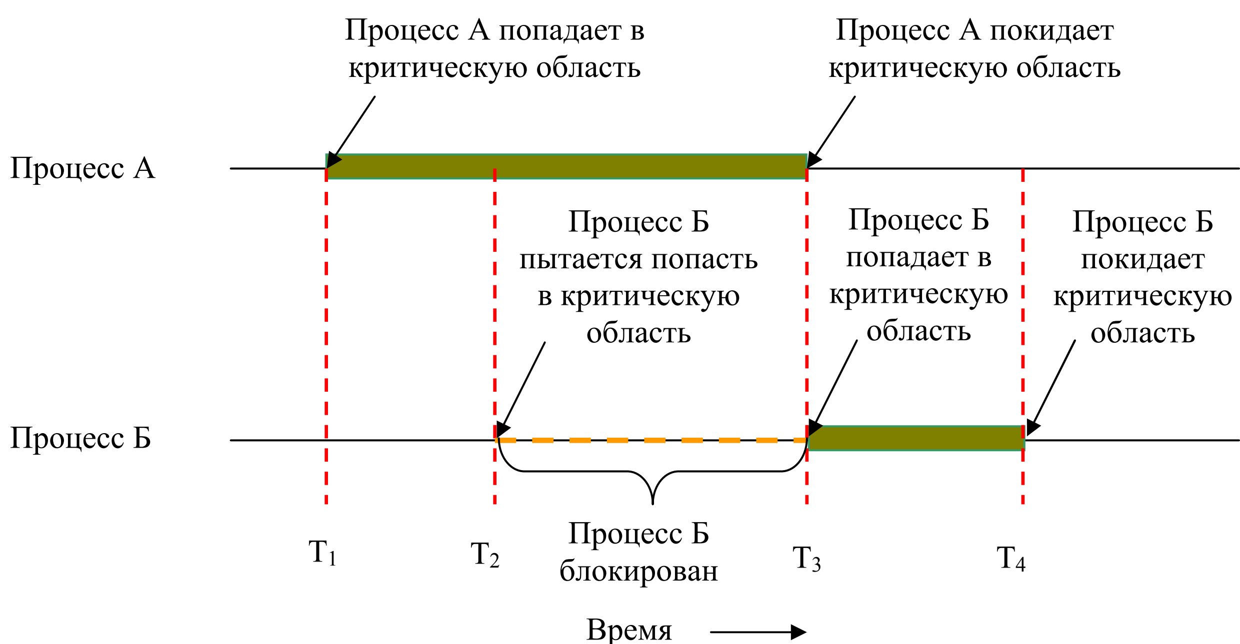

Рисунок 13 – Взаимное исключение с использованием критических областей

В абстрактном виде требуемое поведение процессов представлено на рисунке 13. Процесс ***А*** попадает в критическую область в момент времени T_1. Чуть позже, в момент времени T_2, процесс ***Б*** пытается попасть в критическую область, но ему это не удается, поскольку в критической области уже находится процесс ***А***, а два процесса не должны одновременно находиться в критических областях. Поэтому процесс ***Б*** временно приостанавливается, до наступления момента времени T_3, когда процесс ***А*** выходит из критической области. В момент времени T_4 процесс ***Б*** также покидает критическую область, и происходит возвращение в исходное состояние, когда ни одного процесса в критической области не было.

2.3.1 Взаимное исключение с активным ожиданием

Здесь рассмотрены различные способы реализации взаимного исключения с целью избежать вмешательства в критическую область одного процесса при нахождении там другого и связанных с этим проблем.

1 Запрещение прерывания

Самое простое решение состоит в запрещении всех прерываний при входе процессоров в критическую область и разрешение прерываний по выходе из области. Но это решение неразумно. Предположим все прерывания отключились, а возник какой-то сбой – в результате операционная система закончит своё существование. А если система многопроцессорная, то тогда второй процессор все равно может зайти в критическую область.

2 Переменные блокировки

Программное решение проблемы может носит следующий вид. Пусть переменная блокировки равна 0, процесс, когда хочет попасть в критическую область изменяет её на 1 и входит в критическую область. Тут также может возникнуть состояние состязания, когда два процесса одновременно считывают переменную блокировки, когда она равна 0 и оба входят в критическую область.

3 Строгое чередование

Третий метод проиллюстрирован на листинге 1.

```
//процесс 0
while (TRUE){
	while (turn!=0) ;
	critical_region();
	turn=1;
	noncritical_region();
}
//процесс 1
while (TRUE){
	while (turn!=1) ;
	critical_region();
	turn=0;
	noncritical_region();
}
```

Листинг 1 – Решение проблемы критической области методом строгого чередования

Целая переменная ***turn***, изначально равная 0, отслеживает, чья очередь входить в критическую область. Здесь для того, чтобы 0-ой процесс вошел в

область, ***turn*** должна быть равна 0, а 1-ой – ***turn*** равна 1.

Постоянная проверка значения переменной в ожидании некоторого значения называется активным ожиданием, которое используется только при уверенности в небольшом времени ожидания.

Однако здесь есть недостаток: если один процесс существенно медленнее другого, то может возникнуть ситуация, когда оба процесса находятся вне критической области, однако один процесс блокирован, ожидая пока другой войдёт в критическую область. Это нарушает 3 условие из сформулированных ранее.

4 Алгоритм Петерсона

В 1981 году датский математик Петерсон разработал простой алгоритм взаимного исключения, представленный на листинге 2 [17].

```
#define FALSE     0
#define TRUE      1
#define N         2                       //количество процессов
int turn;                                 //чья сейчас очередь
int interested[N];                        //все переменные изначально равны 0
void enter_region(int process)            //процесс 0 или 1
{
      int other;                          //номер второго процесса
      other=1-process;                    //противоположный процесс
      interested[process]=TRUE;           //индикатор интереса
      turn=process;                       //установка флага
      while (turn==process && interested[other]==TRUE);
}
void leav_region(int process)
{
      interested[process]=FALSE;    //индикатор выхода из критической области
}
```

Листинг 2 – Решение Петерсона для взаимного исключения

Перед тем, как войти в критическую область процесс вызывает процедуру ***enter_regio***n со своим номером в качестве параметра. После выхода из критической области процесс вызывает ***leav_region***.

Исходно оба процесса находятся вне критических областей. Процесс 0 вызывает ***enter_region***, задает элементы массива и устанавливает переменную ***turn*** равной 0. Поскольку процесс 1 не заинтересован в попадании в критическую область, процедура возвращается. Теперь, если процесс 1 вызовет ***enter_region***, ему придется подождать, пока ***interested[0]*** примет значение FALSE, а это произойдет только в тот момент, когда процесс 0 вызовет процедуру ***leave_region***, чтобы покинуть критическую область.

Если оба процесса вызвали ***enter_region*** практически одновременно, то оба сохранят свои номера в ***turn***. Сохранится номер того процесса, который был вторым, а предыдущий номер будет утерян. Предположим, что вторым был процесс 1, так что значение ***turn*** равно 1. Когда оба процесса дойдут до оператора while, процесс 0 войдет в критическую область, а процесс 1 останется в цикле и будет ждать, пока процесс 0 выйдет из критической области.

5 Команда TSL

Это решение требует участия аппаратного обеспечения. Многие компьютеры имеют команду:

TSL RX, LOCK

(Test and Set Lock – проверить и заблокировать), которая действует следующим образом. В регистр ***RX*** считывается содержимое слова памяти ***LOCK***, а в ячейке памяти ***LOCK*** сохраняется некоторое ненулевое значение. Операция считывания слова неделима. Процессор, выполняющий команду TSL, блокирует шину памяти, чтобы остальные процессоры, если они есть, не могли обратиться к памяти.

На листинге 3 представлены функции для входа и выхода из критической области, выполненные в синтаксисе Ассемблера.

```
enter_region:
      TSL REGISTER,LOCK       ; значение LOCK копируется в регистр, значение
                                переменной устанавливается равной 1
      GMP REGISTER,#0         ; старое значение LOCK сравнивается с нулем
      JNE enter_region        ; если оно ненулевое, значит блокировка уже
                                была установлена, поэтому цикл завершается
      RET
leave_region:
      MOVE LOCK,#0            ; сохранение 0 в переменной LOCK
      RET
```

Листинг 3 – Вход и выход из критической области с помощью команды TSL

Прежде чем попасть в критическую область, процесс вызывает процедуру ***enter_region***, которая выполняет активное ожидание вплоть до снятия блокировки, затем она устанавливает блокировку и возвращается. По выходе из критической области процесс вызывает процедуру ***leave_region***, помещающую 0 в переменную ***LOCK***. Как и во всех остальных решениях проблемы критической области, для корректной работы процесс должен вызывать эти процедуры своевременно, в противном случае взаимное исключение не удастся.

2.3.2 Примитивы межпроцессного взаимодействия

Решение Петерсона и с помощью команды TSL корректны, но у них один и тот же недостаток – использование активного ожидания. Т.е. процесс входит в цикл, ожидая возможности войти в критическую область.

Помимо бесцельной траты времени процессора на выполнение данного цикла, существует так называемая проблема инверсии приоритета. Суть её в следующем. Процессу с низким приоритетом никогда не будет предоставлено процессорное время, если в это время выполняется процесс с высоким приоритетом. Таким образом, если процесс с низким приоритетом находится в критической области, а процесс с высоким приоритетом, заканчивая операцию ввода-вывода, оказывается в режиме ожидания, то процессорное время будет отдано процессу с высоким приоритетом. В результате процесс с низким приоритетом никогда не выйдет из критической области, а процесс с высоким приоритетом будет бесконечно выполнять цикл.

Поэтому вместо циклов ожидания применяются примитивы межпроцессного взаимодействия, которые блокируют процессы в случае запрета на вход в критическую область. Одной из простейших является пара примитивов ***sleep*** и

wakeup. Примитив ***sleep*** – системный запрос, в результате которого вызывающий процесс блокируется, пока его не запустит другой процесс. У запроса ***wakeup*** есть один параметр – процесс, который следует запустить. Также возможно наличие одного параметра у обоих запросов – адреса ячейки памяти, используемой для согласования запросов ожидания и запуска.

Два процесса совместно используют буфер ограниченного размера. Один из них, производитель, помещает данные в буфер, а потребитель считывает их оттуда. Трудности начинаются в тот момент, когда производитель хочет поместить в буфер очередную порцию данных и обнаруживает, что буфер полон. Для производителя решением является ожидание, пока потребитель полностью или частично не очистит буфер. Аналогично, если потребитель хочет забрать данные из буфера, а буфер пуст, потребитель уходит в состояние ожидания и выходит из него, как только производитель положит что-нибудь в буфер и разбудит его.

Это решение кажется достаточно простым, но оно приводит к состояниям состязания. Нужна переменная ***count*** для отслеживания количества элементов в буфере. Если максимальное число элементов, хранящихся в буфере, равно ***N***, программа производителя должна проверить, не равно ли ***N*** значение ***count*** прежде, чем поместить в буфер следующую порцию данных. Если значение ***count*** равно ***N***, то производитель уходит в состояние ожидания; в противном случае производитель помещает данные в буфер и увеличивает значение ***count***.

Код программы потребителя прост: сначала проверить, не равно ли значение ***count*** нулю. Если равно, то уйти в состояние ожидания; иначе забрать порцию данных из буфера и уменьшить значение ***count***. Каждый из процессов также должен проверять, не следует ли активизировать другой процесс, и в случае необходимости проделывать это. Программы обоих процессов представлены в листинге 4.

```
#define N 100
int count = 0;

void producer()
{
      int item;
      while (TRUE) {
            item=produce_item();    //сформировать следующий элемент
            if (count==N) sleep();  //буфер полон – состояние ожидания
            insert_item(item);      //поместить элемент в буфер
            count++;
            if (count==1) wakeup(consumer);
      }
}

void consumer()
{
      int item;
      while (TRUE) {
            if (count==0) sleep();  //буфер пуст – состояние ожидания
            item=remove_item(item); //забрать элемент из буфера
            count--;
            if (count==N-1)  wakeup(producer);
      }
}
```

Листинг 4 – Проблема производителя и потребителя с состоянием соревнования

Для описания на языке С системных вызовов ***sleep*** и ***wakeup*** они были представлены в виде вызовов библиотечных процедур. В стандартной библиотеке С их нет, но они будут доступны в любой системе, в которой присутствуют такие системные вызовы. Процедуры ***insert_item*** и ***remove_item*** помещают элементы в буфер и извлекают их оттуда.

Возникновение состояния состязания возможно, поскольку доступ к переменной ***count*** не ограничен. Может возникнуть следующая ситуация: буфер пуст, и потребитель только что считал значение переменной ***count***, чтобы проверить, не равно ли оно нулю. В этот момент планировщик передал управление производителю, производитель поместил элемент в буфер и увеличил значение ***count***, проверив, что теперь оно стало равно 1. Зная, что перед этим оно было равно 0 и потребитель находился в состоянии ожидания, производитель активизирует его с помощью вызова ***wakeup***.

Но потребитель не был в состоянии ожидания, так что сигнал активизации пропал впустую. Когда управление перейдет к потребителю, он вернется к считанному когда-то значению ***count***, обнаружит, что оно равно 0, и уйдет в состояние ожидания. Рано или поздно производитель наполнит буфер и также уйдет в состояние ожидания. Оба процесса так и останутся в этом состоянии.

Суть проблемы в данном случае состоит в том, что сигнал активизации, пришедший к процессу, не находящемуся в состоянии ожидания, пропадает. Если бы не это, проблемы бы не было. Быстрым решением может быть добавление бита ожидания активизации. Если сигнал активизации послан процессу, не находящемуся в состоянии ожидания, этот бит устанавливается. Позже, когда процесс пытается уйти в состояние ожидания, бит ожидания активизации сбрасывается, но процесс остается активным. Этот бит исполняет роль копилки сигналов активизации.

Несмотря на то, что введение бита ожидания запуска спасло положение в этом примере, легко сконструировать ситуацию с несколькими процессами, в которой одного бита будет недостаточно. Можно добавить еще один бит, или 8, или 32, но это не решит проблему.

В 1965 году Дейкстра [16] предложил использовать семафор – переменную для подсчета сигналов запуска. Семафор – объект синхронизации, который может регулировать доступ к некоторому ресурсу. Также было предложено использовать вместо ***sleep*** и ***wakeup*** две операции ***down*** и ***up***. Их отличие в следующем: если значение семафора больше нуля, то ***down*** просто уменьшает его на 1 и возвращает управление процессу, в противном случае процесс переводится в режим ожидания. Все операции проверки значения семафора, его изменения и перевода процесса в состояние ожидания выполняются как единое и неделимое элементарное действие, т.е. в это время ни один процесс не может получить доступ к этому семафору. Операция ***up*** увеличивает значение семафора. Если с этим семафором связаны один или несколько ожидающих процессов, которые не могут завершить более раннюю операцию down, один из них выбирается системой и разблокируется. Проблема производителя и потребителя легко решается с помощью семафоров.

Иногда используется упрощенная версия семафора, называемая мьютексом. Мьютекс – переменная, которая может находиться в одном из двух состояний: блокированном или неблокированном. Поэтому для описания мьютекса требуется

всего один бит. Мьютекс может охранять неразделенный ресурс, к которому в каждый момент времени допускается только один поток, а семафор может охранять ресурс, с которым может одновременно работать не более ***N*** потоков.

Недостатком семафоров является то, что одна маленькая ошибка при их реализации программистом приводит к остановке всей операционной системы. Чтобы упростить написание программ в 1974 году было предложено использовать примитив синхронизации более высокого уровня, называемый монитором. Монитор – набор процедур, переменных и других структур данных, объединенных в особый модуль или пакет. Процессы могут вызывать процедуры монитора, но у процедур объявленных вне монитора, нет прямого доступа к внутренним структурам данных монитора. При обращении к монитору в любой момент времени активным может быть только один процесс. Монитор похож по своей структуре на класс в C++. Не все языки программирования поддерживают мониторы и не во всех операционных системах есть их встроенная реализация. Так в Windows их нет.

Все описанные примитивы не подходят для реализации обмена информации между компьютерами в распределенной системе с несколькими процессорами. Для этого используется передача сообщений. Этот метод межпроцессного взаимодействия использует два примитива: ***send*** и ***receive***, которые скорее являются системными вызовами, чем структурными компонентами языка. Первый запрос посылает сообщение заданному адресату, а второй получает сообщение от указанного источника. Передача сообщений часто используется в системах с параллельным программированием.

Последний из рассмотренных механизмов синхронизации называется барьер, который предназначен для синхронизации группы процессов – т.е. несколько процессов выполняют вычисления с разной скоростью, а затем посредством применения барьера ожидают, пока самый медленный не закончит работу, и только потом все вместе продолжают выполнение команд.

Литература по операционным системам содержит множество интересных проблем, которые широко обсуждались и анализировались с применением различных методов синхронизации. Часть из них описана в работе [14].

2.4 Планирование

Часть операционной системы, отвечающая за выбор рабочего процесса из группы активных процессов, называется планировщиком, а используемый алгоритм – алгоритмом планирования. Практически все процессы чередуют периоды вычислений с операциями ввода-вывода (Рисунок 14).

Обычно процессор работает некоторое время без остановки, затем происходит системный вызов, например, на чтение из файла или запись в файл. Некоторые процессы большую часть времени заняты вычислениями, а некоторые ожидают ввода-вывода. Первые процессы называются ограниченными возможностями процессора, вторые – ограниченными возможностями устройств ввода-вывода.

Основные ситуации, когда необходимо применять планирование:

- при создание нового процесса, необходимо решить, какой процесс запустить: родительский или дочерний;

- при завершении работы процесса необходимо из набора готовых процессов выбрать и запустить следующий, если нет ни одного готового, то запускается холостой процесс из операционной системы;
- при блокировании процесса на операции ввода-вывода, семафоре или по-другому, необходимо выбрать и запустить другой процесс;
- при появлении прерывания ввода-вывода, т.е. необходимо выбрать запускать ли процесс, который ожидал этого ввода-вывода или другой.

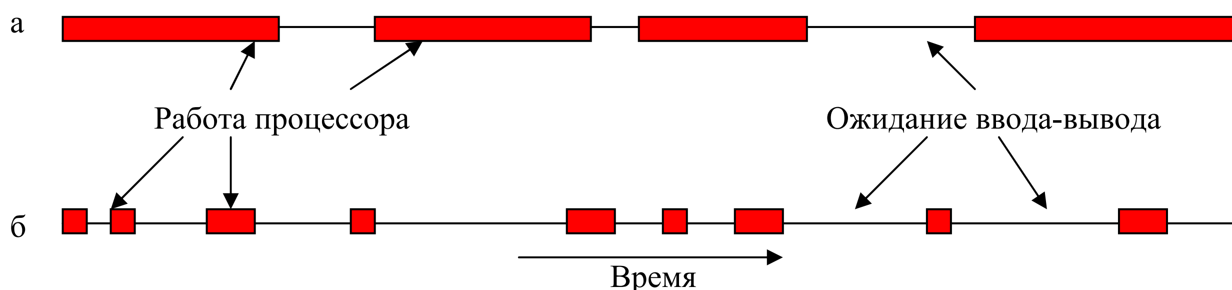

Рисунок 14 – Периоды использования процессора, чередующиеся с ожиданием ввода-вывода: процесс, ограниченный возможностями процессора (а); процесс, ограниченный возможностями устройств ввода-вывода (б)

Если аппаратный таймер выполняет периодические прерывания с частотой 50 Гц, 60 Гц или с любой другой частотой, решения планирования могут приниматься при каждом прерывании по таймеру или при каждом k-м прерывании. Алгоритмы планирования можно разделить на две категории согласно их поведению после прерываний.

1 Алгоритмы планирования без переключений (неприоритетное планирование), выбирают процесс и позволяют его работать вплоть до блокировки (в ожидании ввода-вывода или другого процесса), либо вплоть до того момента, когда процесс сам не отдаст процессор. Процесс может работать часами.

2 Алгоритмы планирования с переключениями (приоритетное планирование), выбирают процесс и позволяют ему работать максимально фиксированное время, затем приостанавливается и управление переходит к другому процессу. Приоритетное планирование требует прерываний по таймеру, чтобы передать управление планировщику.

В различных средах используются различные алгоритмы планирования. Выделяют 3 типичных среды [14]:

- системы пакетной обработки данных;
- интерактивные системы;
- системы реального времени.

В первых системах нет пользователей за терминалами и в таких системах приемлемы алгоритмы без переключений или с переключениями, но с большим временем, отводимым каждому процессу.

Во вторых системах необходимы алгоритмы с переключениями, чтобы предотвратить захват процессора одним процессом.

В третьих системах приоритетное планирование необязательно, т.к. там

существуют другие программы, которые знают когда надо блокироваться.

Основные задачи алгоритмов планирования:

а) для всех систем;

1) справедливость – предоставление каждому процессу справедливой доли процессорного времени;

2) принудительное применение политики – контроль за выполнением принятой политики (т.е., к примеру, предоставление процессам контроля безопасности процессора по первому требованию);

3) баланс – поддержка занятости всех частей системы (т.е. важно, чтобы работало больше устройств, чем один процесс только производил вычисления);

б) для систем пакетной обработки данных;

1) пропускная способность – максимальное количество задач в час;

2) оборотное время – минимизация времени, затрачиваемого на ожидание обслуживания и обработку задачи;

3) использование процессора – поддержка постоянной занятости процессора;

в) для интерактивных систем;

1) время отклика – быстрая реакция на запросы;

2) соразмерность – выполнение пожеланий пользователя;

г) для систем реального времени;

1) окончание работы к сроку – предотвращение потери данных;

2) предсказуемость – предотвращение деградации качества в мультимедийных системах.

Далее рассмотрим некоторые алгоритмы планирования процессов. Для планирования потоков используются те же алгоритмы планирования, лишь есть некоторые отличия при различной реализации управления потоками на уровне ядра и уровне пользователя, которые сводятся к различной производительности.

2.4.1 Планирование в системах пакетной обработки данных

1 «Первым пришёл – первым ушёл»

Самый простой алгоритм планирования. Категория алгоритма – без переключений. Процессам предоставляется доступ к процессору в том порядке, в котором они его запрашивают. Формируется единая очередь процессов. Когда текущий процесс блокируется, запускается следующий в очереди, а когда блокировка снимается, процесс попадает в конец очереди. Недостаток в том, что если существует очередь процессов, в котором есть процессы ограниченные устройствами ввода-вывода (т.е. большую часть времени тратящие на ожидание устройств), то это ожидание будет означать простой процессора.

2 Алгоритм «Кратчайшая задача – первая»

Категория алгоритма – без переключений. Суть алгоритма заключается в следующем: если в очереди есть несколько одинаково важных задач, планировщик выбирает первой самую короткую по времени. Эта схема работает лишь в случае одновременного наличия задач и обычно неактуальна.

3 Алгоритм «Наименьшее оставшееся время выполнения»

Версия предыдущего алгоритма. В соответствии с этим алгоритмом, планировщик каждый раз выбирает процесс с наименьшим оставшимся временем выполнения. Естественно для таких алгоритмов необходимо знать, сколько времени выполняются процессы, что обычно является сложной задачей.

4 Алгоритм трехуровневого планирования

Системы пакетной обработки позволяют реализовать трехуровневое планирование, как показано на рисунке 15. По мере поступления в систему новые задачи сначала помещаются в очередь, хранящуюся на диске. Планировщик доступа выбирает задание и передает его системе. Остальные задачи остаются в очереди. Выбор заданий обуславливается установленным приоритетом – по времени выполнения или как-то по другому, например по работе с устройствами ввода-вывода.

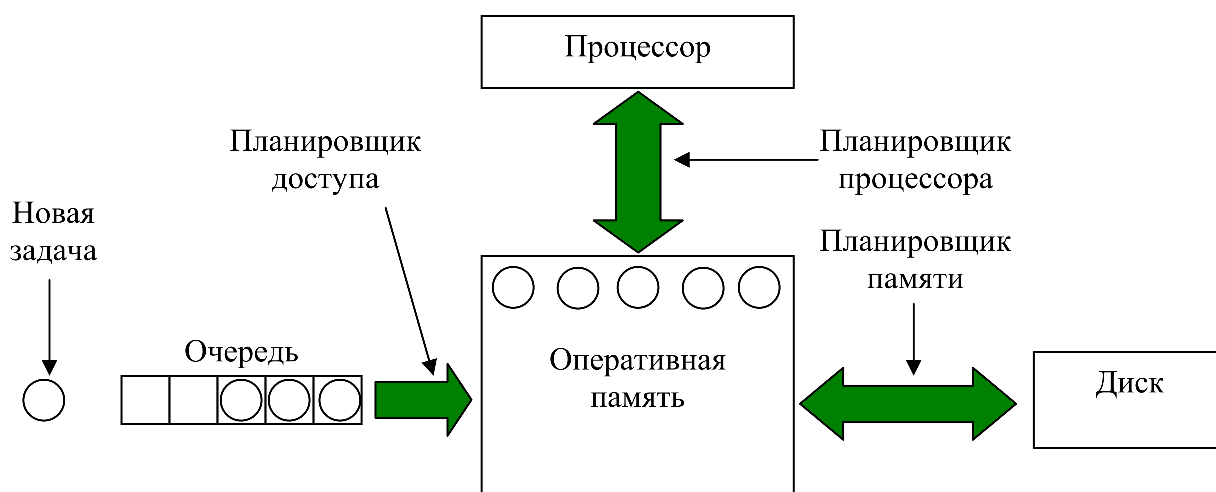

Рисунок 15 – Трехуровневое планирование

Как только задание попало в систему, для него будет создан соответствующий процесс, который вступает в борьбу за доступ к процессору. В ситуации, когда процессов слишком много, работает второй уровень планирования (планировщик памяти), который определяет, какие процессы будут находиться в памяти, а какие можно выгрузить на диск. Естественно это не должно происходить слишком часто, т.к. дисковые операции сравнительно медленные. Количество процессов, одновременно находящихся в памяти, называется степенью многозадачности.

Третий уровень планирования отвечает за доступ процессов, находящихся в состоянии готовности, к процессору. Этим планировщиком используется любой подходящий к ситуации алгоритм, как с прерыванием, так и без.

2.4.2 Планирование в интерактивных системах

В интерактивных системах невозможно трехуровневое планирование, но двухуровневое (планировщик памяти и процессора) возможно и часто используется. Ниже рассмотрены алгоритмы для планировщика процессора. Все алгоритмы планирования для интерактивных систем могут использоваться в качестве

планировщика процессора в системах пакетной обработки.

1 Алгоритм циклического планирования

Каждому процессу предоставляется некоторый интервал времени процессора, так называемый квант времени. Если к концу кванта времени процесс всё ещё работает, он прерывается, а управление передается следующему процессу, а предыдущий процесс отправляется в конец списка. Реализация циклического планирования проста. Планировщику нужно всего лишь поддерживать список процессов в состоянии готовности согласно рисунку 16, а. Когда процесс исчерпал свой лимит времени, он отправляется в конец списка (Рисунок 16, б).

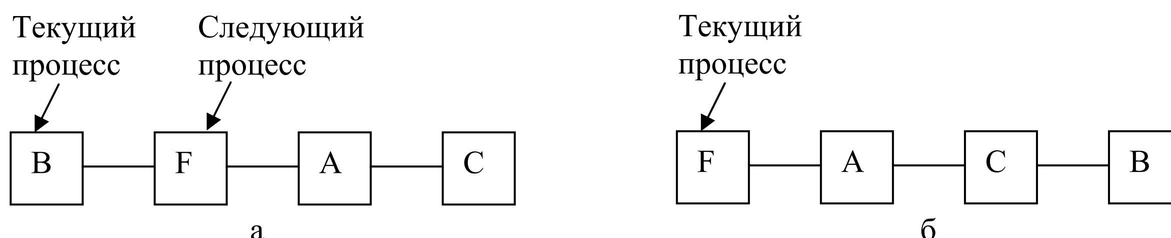

Рисунок 16 – Циклическое планирование

В данном алгоритме нужно очень осторожно выбирать квант времени, т.к. слишком малый квант приведёт к частому переключению процессов и небольшой эффективности, а слишком медленный квант может привести у медленному реагированию на короткие интерактивные запросы.

2 Алгоритм с приоритетным планированием

Циклический алгоритм исходил из того, что все процессы равнозначны, но в реальности это не так – у каждого свой приоритет. Основная идея алгоритма – управление передается процессу с самым высоким приоритетом. Чтобы предотвратить бесконечную работу процессов с высоким приоритетом, планировщик может уменьшать приоритет процесса с течением времени. Если процессов с одинаковым приоритетом несколько, то применяется циклическое планирование.

3 Алгоритм «Самый короткий процесс – следующий»

Сходен с алгоритмом «Кратчайшая задача – первая». Но если в системах пакетной обработки предполагается, что примерно известно время выполнения задачи, то для оценки длины процесса в интерактивных системах используется метод оценки по предыдущим запускам программы, называемый старением.

Допустим, что предполагаемое время исполнения команды равно $\boldsymbol{T_0}$ и предполагаемое время следующего запуска равно $\boldsymbol{T_1}$,. Можно улучшить оценку времени, взяв взвешенную сумму этих времен $\alpha\boldsymbol{T_0} + (1 - \alpha)\ \boldsymbol{T_1}$. Выбирая соответствующее значение α, можно заставить алгоритм оценки быстро забывать о предыдущих запусках или, наоборот, помнить о них в течение долгого времени. Взяв α = 1/2, получим серию оценок:

$$T_0, \frac{T_0}{2} + \frac{T_1}{2}, \frac{T_0}{4} + \frac{T_1}{4} + \frac{T_2}{2}, \frac{T_0}{8} + \frac{T_1}{8} + \frac{T_2}{4} + \frac{T_3}{2}. \qquad (1)$$

4 Гарантированное планирование

Суть данного алгоритма заключается в том, чтобы отдавать процессам

реальное количество циклов процессора. Если процессором пользуются ***n*** пользователей, то одному пользователю будет предоставлено 1/***n*** мощности процессора. Чтобы выполнить это обещание, система должна отслеживать распределение процессора между процессами с момента создания каждого процесса. Система рассчитывает количество ресурсов процессора, на которое процесс имеет право, например время с момента создания, деленное на ***n***. Теперь можно сосчитать отношение времени, предоставленного процессу, к времени, на которое он имеет право. Полученное значение 0.5 означает, что процессу выделили только половину положенного, а 2.0 означает, что процессу досталось в два раза больше, чем положено. Затем запускается процесс, у которого это отношение наименьшее, пока оно не станет больше, чем у его ближайшего соседа.

5 Лотерейное планирование

В основе алгоритма лежит раздача процессам лотерейных билетов на доступ к различным ресурсам, в том числе и к процессору. Когда планировщику необходимо принять решение, выбирается случайным образом лотерейный билет, и его обладатель получает доступ к ресурсу. Что касается доступа к процессору, «лотерея» может происходить 50 раз в секунду, и победитель получает 20 мс времени процессора. Если всего 100 билетов и 20 из них находятся у одного процесса, то ему достанется 20 % времени процессора. В отличие от приоритетного планировщика, в котором очень трудно оценить, что означает, скажем, приоритет 40, в лотерейном планировании все очевидно. Каждый процесс получит процент ресурсов, примерно равный проценту имеющихся у него билетов.

6 Справедливое планирование

Некоторые системы обращают внимание на хозяина процесса перед планированием. В такой модели каждому пользователю достается некоторая доля процессора, и планировщик выбирает процесс в соответствии с этим фактом. Если в нашем примере каждому из пользователей было обещано по 50 % процессора, то им достанется по 50 % процессора, независимо от количества процессов.

2.4.3 Планирование в системах реального времени

Планирование в системах реального времени отличается от интерактивных систем и систем пакетной обработки. Система реального времени, это, к примеру, проигрыватель компакт-дисков. Здесь стоит отметить, что существуют жесткие системы реального времени, в которых есть наличие жестких сроков выполнения задач, и гибкие системы реального времени, в которых нарушение графика работы нежелательно, но допустимо. За соблюдение графика отвечает планировщик.

Внешние события, на которые система реального времени должна реагировать, можно разделить на периодические и непериодические. В зависимости от времени, затрачиваемого на обработку каждого из событий, может оказаться, что система не в состоянии своевременно обработать все события. Существует математическое условие, согласно, которому система реального времени считается поддающейся планированию:

$$\sum_{i=1}^{m} \frac{C_i}{P_i} \le 1. \qquad (2)$$

Если в систему поступает m периодических событий, событие с номером $\boldsymbol{i}$ поступает с периодом $\boldsymbol{P_i}$, и на его обработку уходит $\boldsymbol{C_i}$ секунд работы процессора, все потоки могут быть своевременно обработаны только при выполнении этого условия.

2.5 Понятие взаимоблокировки

Когда взаимодействуют два или более процессов, они могут попадать в патовые ситуации, из которых невозможно выйти без посторонней помощи. Такая ситуация называется тупиком, тупиковой ситуацией или взаимоблокировкой.

Например, первый процесс запрашивает сканер и получает его. В это время второй процесс запрашивает DVD и получает его. Затем оба пытаются получить DVD и сканер и блокируются, ожидая пока устройства освободятся. Есть много примеров и не связанных с вводом-выводом. Система может зайти в тупик, когда процессам предоставляются исключительные права доступа к устройствам, файлам и т.д., т.е. каким-либо ресурсом компьютера. Ресурсом может быть аппаратное устройство или часть информации. Ресурсы бывают двух типов – выгружаемые (оперативная память) и невыгружаемые (DVD-дисковод). Выгружаемые можно отнять у процесса, а невыгружаемые – нельзя. Взаимоблокировки касаются именно невыгружаемых ресурсов.

Определение: группа процессов находится в тупиковой ситуации, если каждый процесс из группы ожидает события, которое может вызывать только другой процесс из той же группы.

Для возникновения ситуации взаимоблокировки должны выполняться четыре условия [15].

1 Условие взаимного исключения. Каждый ресурс в данный момент или отдан ровно одному процессу, или доступен.

2 Условие удержания и ожидания. Процессы, в данный момент удерживающие полученные ранее ресурсы, могут запрашивать новые ресурсы.

3 Условие отсутствия принудительной выгрузки ресурса. У процесса нельзя принудительным образом забрать ранее полученные ресурсы. Процесс, владеющий ими, должен сам освободить их.

4 Условия циклического ожидания. Должна существовать круговая последовательность из двух и более процессов, каждый из которых ждёт доступа к ресурсу, удерживаемому следующим членом последовательности.

Для исключения взаимоблокировки необходимо исключить какое-нибудь из этих условий.

Для моделирования взаимоблокировок удобны в использовании графы. Графы имеют два вида узлов: процессы, показанные кружочками, и ресурсы, нарисованные квадратиками. Ребро, направленное от узла ресурса (квадрат) к узлу процесса (круг), означает, что ресурс ранее был запрошен процессом, получен и в данный момент используется этим процессом. На рисунке 17 приведены примеры графов.

Рисунок 17 – Графы распределения ресурсов: ресурс R отдан процессу A (а); процесс B ждёт ресурса S (б); взаимоблокировка (в)

На рисунке 18 изображена ситуация возникновения взаимоблокировки. Есть 3 процесса и 3 ресурса. На шагах 1-3 каждый процесс получает свой ресурс. На шагах 4-6 каждый процесс впадает в режим ожидания освобождения ресурса. Возникает взаимоблокировка. Данной ситуации можно избежать, если на время заблокировать один из процессов.

Рисунок 18 – Ситуация возникновения взаимоблокировки

Существуют 4 стратегии действий при взаимоблокировке:

1 Пренебрежение проблемой в целом. Если вы проигнорируете проблему, то возможно в дальнейшем она проигнорирует вас.

2 Обнаружение и восстановление. Позволить взаимоблокировке произойти, обнаружить её и предпринять какие-либо действия.

3 Динамическое избегание тупиковых ситуаций с помощью аккуратного распределения ресурсов.

4 Предотвращение с помощью структурного опровержения одного из четырёх условий, необходимых для взаимоблокировки.

Большая часть операционных систем, включая UNIX и Windows, игнорируют проблему взаимоблокировки, т.к. исходят из предположения, что большинство пользователей предпочтут иметь дело со случающимися время от времени взаимоблокировками, чем руководствоваться правилом, ограничивающим использование ресурсов. Поэтому мы рассматривать алгоритмы взаимоблокировки не будем.

Контрольные вопросы по разделу

1 Поясните употребление терминов «программам, «процесс», «задача», «поток», «нить».

2 В системе с потоками на уровне пользователя каждому потоку соответствует собственный стек или каждому процессу? А в системе с потоками на уровне ядра? Поясните.

3 Что такое состояние состязания?

4 При разработке компьютера он сначала моделируется программой, выполняющей одновременно только одну команду. Даже компьютер с несколькими процессорами моделируется также последовательно. Может ли в такой ситуации возникнуть состояние соревнования, когда одновременных событий нет?

5 Будет ли решение Петерсона проблемы взаимного исключения работать в случае планирования процессов с переключениями? В случае планирования без переключений?

6 Обычно планировщики с циклическим алгоритмом поддерживают список процессов, готовых к работе, причем каждый процесс находится в списке в единственном экземпляре. Что произойдет, если процесс окажется в списке дважды? Существует ли причина, по которой подобное изменение будет полезным?

7 Представьте себе операционную систему, разработанную для компьютера, в котором отсутствует система прерываний. Какой алгоритм планирования процессов может быть реализован в такой системе?

8 Приведите пример алгоритма планирования, в результате работы которого процесс, располагая всеми необходимыми ресурсами, может бесконечно долго находиться в системе, не имея возможности завершиться.

9 Пусть в системе существует два процесса и три одинаковых ресурса. Каждому процессу требуется два ресурса. Возможна ли взаимоблокировка? Объясните ваш ответ.

3 Управление памятью

3.1 Основы управления памятью

Часть операционной системы, отвечающая за управление памятью, называется менеджером памяти. Функциями операционной системы по управлению памятью в мультипрограммной системе являются:

- отслеживание свободной и занятой памяти;
- выделение памяти процессам и освобождение памяти по завершении процессов;
- вытеснение кодов и данных процессов из оперативной памяти на диск (полное или частичное), когда размеры основной памяти не достаточны для размещения в ней всех процессов, и возвращение их в оперативную память, когда в ней освобождается место;
- настройка адресов программы на конкретную область физической памяти [11].

Для идентификации переменных и команд на разных этапах жизненного цикла программы используются символьные имена (метки), виртуальные адреса и физические адреса (Рисунок 19).

- Символьные имена присваивает пользователь при написании программы на алгоритмическом языке или ассемблере.
- Виртуальные адреса, называемые иногда математическими, или логическими адресами, вырабатывает транслятор, переводящий программу на машинный язык. Поскольку во время трансляции в общем случае не известно, в какое место оперативной памяти будет загружена программа, то транслятор присваивает переменным и командам виртуальные (условные) адреса, обычно считая по умолчанию, что начальным адресом программы будет нулевой адрес.
- Физические адреса соответствуют номерам ячеек оперативной памяти, где в действительности расположены или будут расположены переменные и команды.

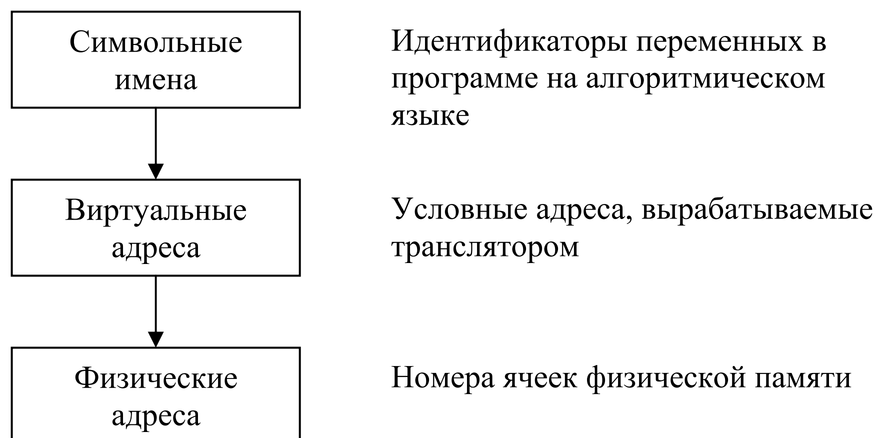

Рисунок 19 – Типы адресов

Совокупность виртуальных адресов процесса называется виртуальным адресным пространством. Диапазон возможных адресов виртуального пространства у всех процессов является одним и тем же. Совпадение виртуальных адресов переменных и команд различных процессов не приводит к конфликтам, так как в том случае, когда эти переменные одновременно присутствуют в памяти, операционная система отображает их на разные физические адреса.

Существуют два принципиально отличающихся подхода к преобразованию виртуальных адресов в физические.

В первом случае замена виртуальных адресов на физические выполняется один раз для каждого процесса во время начальной загрузки программы в память.

Второй способ заключается в том, что программа загружается в память в неизмененном виде в виртуальных адресах, то есть операнды инструкций и адреса переходов имеют те значения, которые выработал транслятор. В наиболее простом случае, когда виртуальная и физическая память процесса представляют собой единые непрерывные области адресов, операционная система выполняет преобразование виртуальных адресов в физические по следующей схеме. При загрузке операционная система фиксирует смещение действительного расположения программного кода относительно виртуального адресного пространства. Во время выполнения программы при каждом обращении к оперативной памяти выполняется преобразование виртуального адреса в физический. Схема такого преобразования показана на рисунке 20.

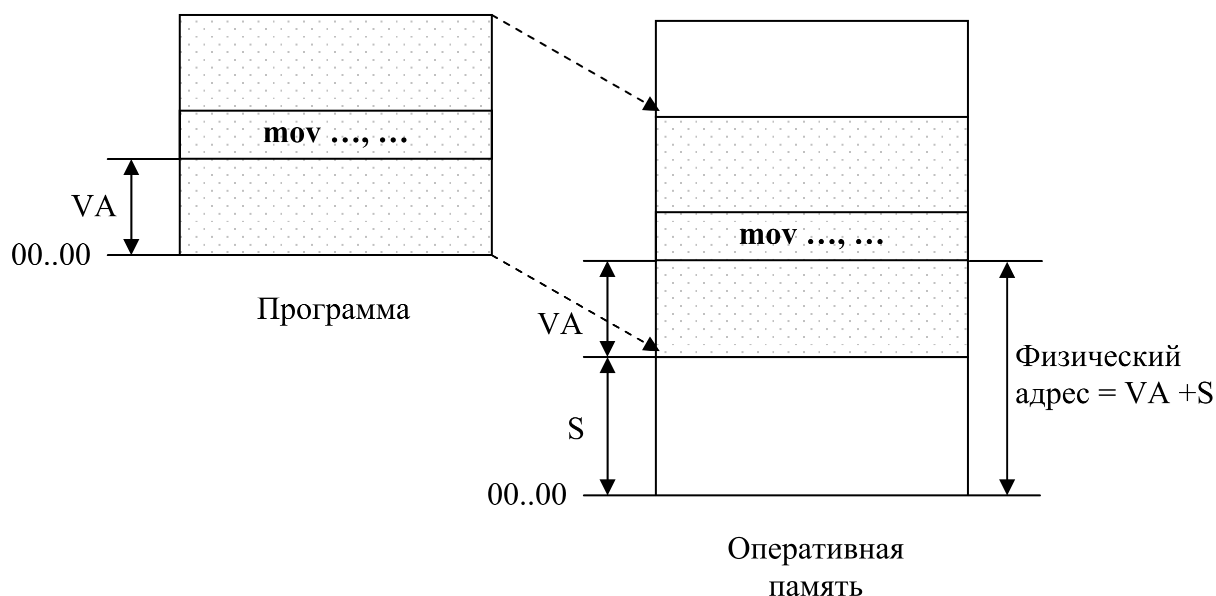

Рисунок 20 – Схема динамического преобразования адресов

Системы управления памятью разделяются на два класса по методам распределения памяти:

- перемещающие процессы между памятью и диском;
- не делающие этого, что представлено на рисунке 21.

Перед тем, как рассматривать методы распределения памяти для многозадачных систем, которые представлены на рисунке 21, рассмотрим однозадачную систему без подкачки на диск, т.е. систему, в которой в каждый

момент времени работает только одна программа. Простейшие три способа организации памяти для такой системы, представлены на рисунке 22.

Рисунок 21 – Методы распределения памяти

На рисунках представлено условное разделение памяти на верхнюю ПЗУ и нижнюю ОЗУ. Первая модель использовалась на старых компьютерах, Вторая модель используется сейчас на некоторых встроенных системах. Третья модель устанавливалась на ранних персональных компьютерах, оснащенных MS-DOS, где в роли ПЗУ выступает BIOS.

При использовании многозадачности повышается эффективность загрузки центрального процессора. К примеру, если средний процесс выполняет вычисления только 20 % от того времени, которое он находится в памяти, то при присутствии в памяти одновременно пяти процессов центральный процессор должен быть занят все время.

Рисунок 22 – Простейшие модели организации памяти при наличии операционной системы и одного пользовательского процесса

Если в памяти находится одновременно $\boldsymbol{n}$ процессов, вероятность того, что все $\boldsymbol{n}$ процессов ждут ввод-вывод (в этом случае центральный процессор будет бездействовать), равна $\boldsymbol{p^n}$. Тогда степень загрузки центрального процессора будет

выражаться формулой [14]:

$$C = 1 - p^n. \quad (3)$$

На рисунке 23 показана зависимость степени использования центрального процессора от числа n, называемого степенью многозадачности.

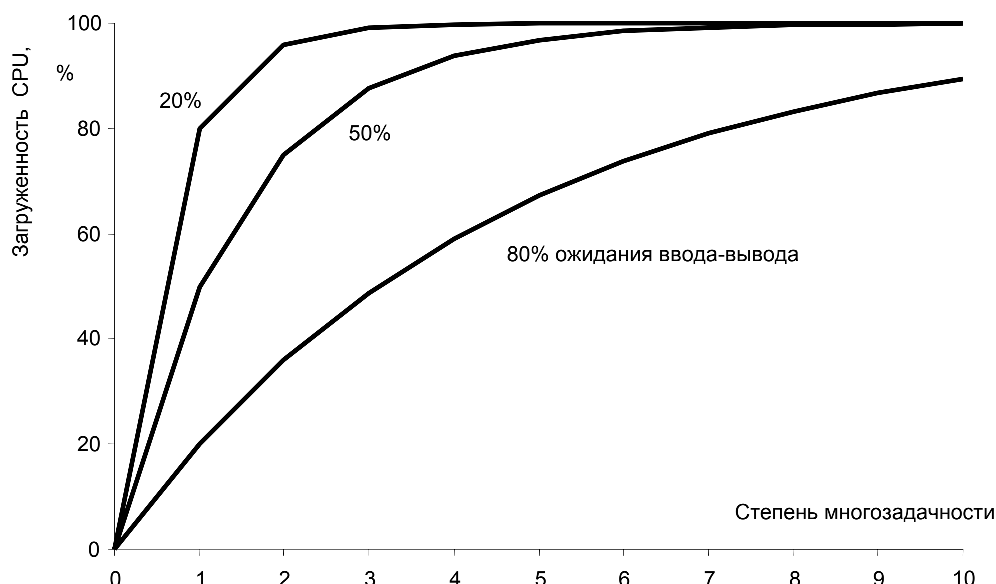

Рисунок 23 – Зависимость загрузки процессора от числа задач и процента ожидания ввода-вывода от общей времени работы процесса

Из рисунка понятно, что если процессы проводят 80 % своего времени в ожидании завершения операции ввода-вывода, то для того, чтобы получить потерю времени процессора ниже 10 %, в памяти должны одновременно находиться, по меньшей мере, 10 процессов.

3.2 Методы распределения памяти без использования подкачки

3.2.1 Метод распределения с фиксированными разделами

Первой многозадачной системой была именно система с фиксированными разделами. Память разбивается на несколько областей фиксированной величины, называемых разделами. Такое разбиение может быть выполнено вручную оператором во время старта системы или во время ее установки. После этого границы разделов не изменяются.

Очередной новый процесс, поступивший на выполнение, помещается либо в общую очередь (Рисунок 24, а), либо в очередь к некоторому разделу (Рисунок 24, б).

Подсистема управления памятью в этом случае выполняет следующие задачи.

- Сравнивает объем памяти, требуемый для вновь поступившего процесса, с размерами свободных разделов и выбирает подходящий раздел.
- Осуществляет загрузку программы в один из разделов и настройку адресов. Уже на этапе трансляции разработчик программы может задать раздел, в котором ее следует выполнять. Это позволяет сразу, без использования

перемещающего загрузчика, получить машинный код, настроенный на конкретную область памяти.

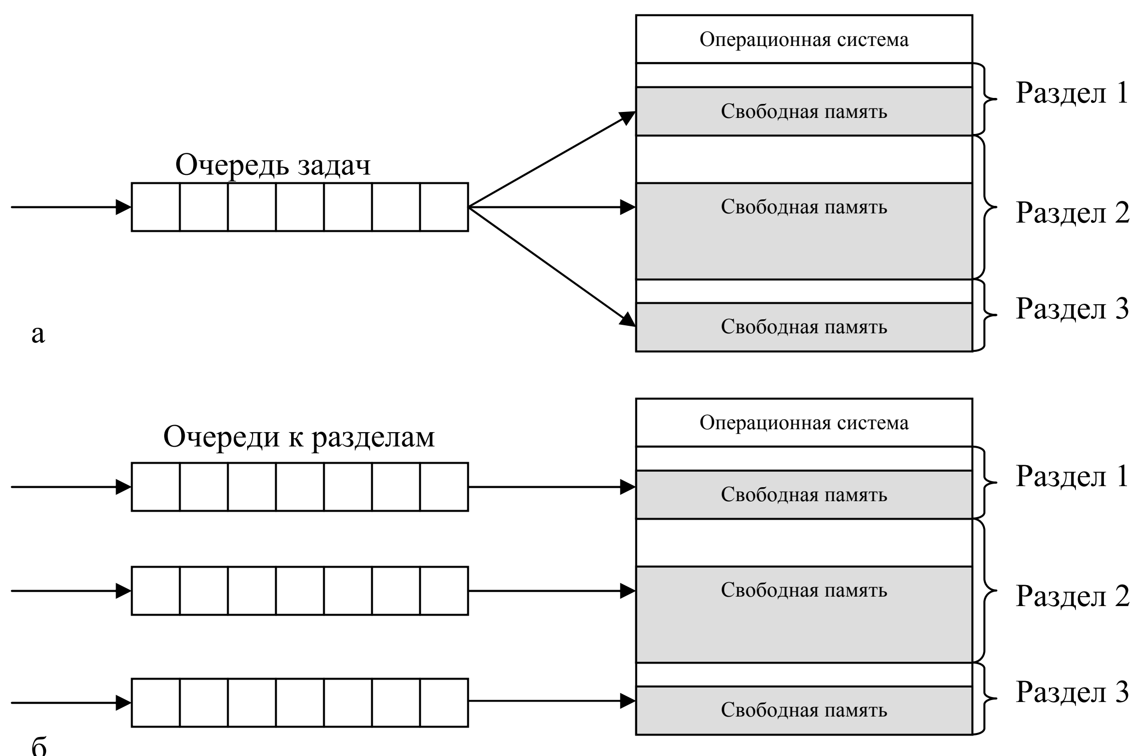

Рисунок 24 – Распределение памяти фиксированными разделами: с общей очередью (а), с отдельными очередями (б)

При очевидном преимуществе – простоте реализации, данный метод имеет существенный недостаток – жесткость заданных размеров памяти для каждого процесса. Подобная схема использовалась в OS/360 и на настоящий момент не используется.

3.2.2 Метод распределения с динамическими разделами

В этом случае память машины не делится заранее на разделы. Сначала вся память, отводимая для приложений, свободна. Каждому вновь поступающему на выполнение приложению на этапе создания процесса выделяется вся необходимая ему память (если достаточный объем памяти отсутствует, то приложение не принимается на выполнение и процесс для него не создается). После завершения процесса память освобождается, и на это место может быть загружен другой процесс.

На рисунке 25 показано состояние памяти в различные моменты времени при использовании динамического распределения. Так, в момент t_0 памяти находится только операционная система, а к моменту t_1 память разделена между пятью процессами, причем процесс ***П4***, завершаясь, покидает память. На освободившееся от процесса ***П4*** место загружается процесс ***П6***, поступивший в момент t_3.

Функции операционной системы, предназначенные для реализации данного метода управления памятью, перечислены ниже.

- Ведение таблиц свободных и занятых областей, в которых указываются начальные адреса и размеры участков памяти.
- При создании нового процесса — анализ требований к памяти, просмотр таблицы свободных областей и выбор раздела, размер которого достаточен для размещения кодов и данных нового процесса. Выбор раздела может осуществляться по разным правилам, например: «первый попавшийся раздел достаточного размера», «раздел, имеющий наименьший достаточный размер» или «раздел, имеющий наибольший достаточный размер».
- Загрузка программы в выделенный ей раздел и корректировка таблиц свободных и занятых областей Данный способ предполагает, что программный код не перемещается во время выполнения, а значит, настройка адресов может быть проведена единовременно во время загрузки.
- После завершения процесса корректировка таблиц свободных и занятых областей.

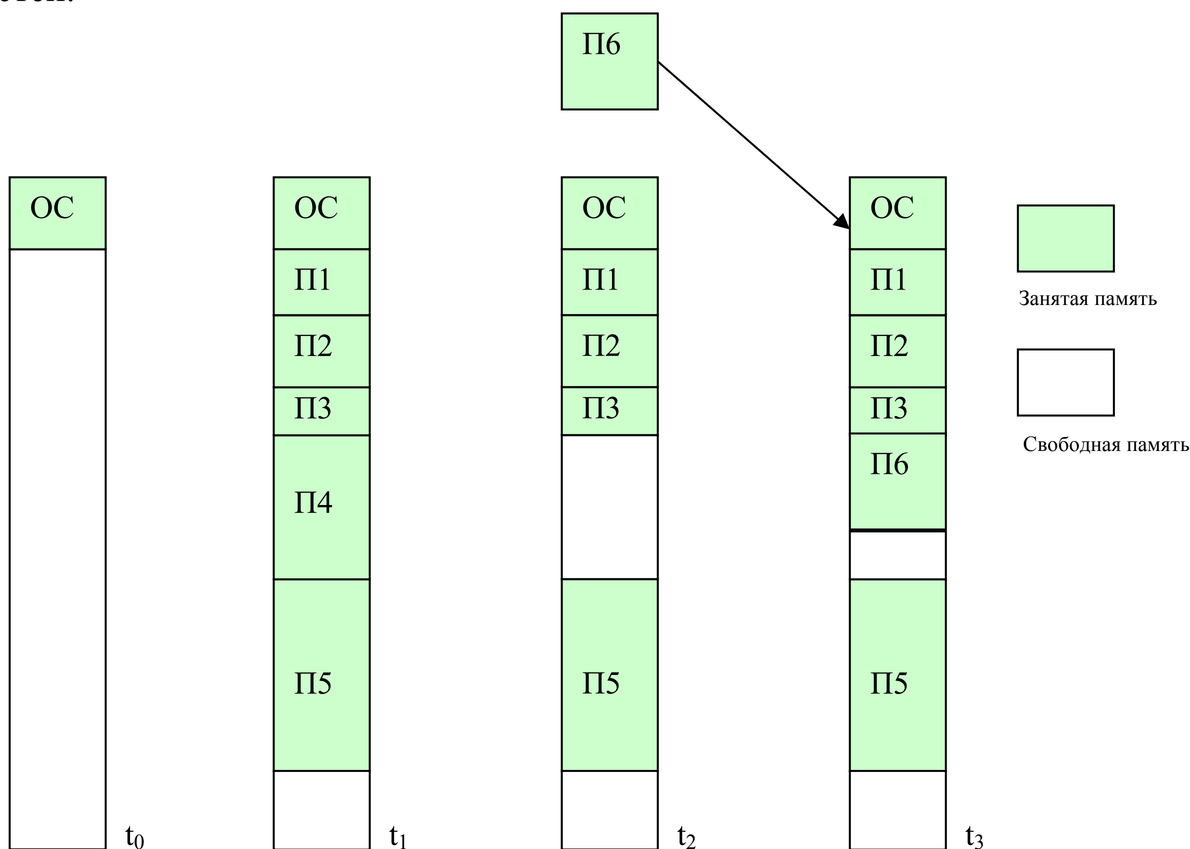

Рисунок 25 – Распределение памяти динамическими разделами

По сравнению с методом распределения памяти фиксированными разделами данный метод обладает гораздо большей гибкостью, но ему присущ очень серьезный недостаток – фрагментация памяти. Фрагментация – это наличие большого числа несмежных участков свободной памяти очень маленького размера (фрагментов).

3.2.3 Метод распределения с перемещаемыми разделами

Одним из методов борьбы с фрагментацией является перемещение всех занятых участков в сторону старших или младших адресов, так, чтобы вся свободная память образовала единую свободную область (Рисунок 26). В дополнение к функциям, которые выполняет операционная система при распределении памяти динамическими разделами, в данном случае она должна еще время от времени копировать содержимое разделов из одного места памяти в другое, корректируя таблицы свободных и занятых областей. Эта процедура называется сжатием.

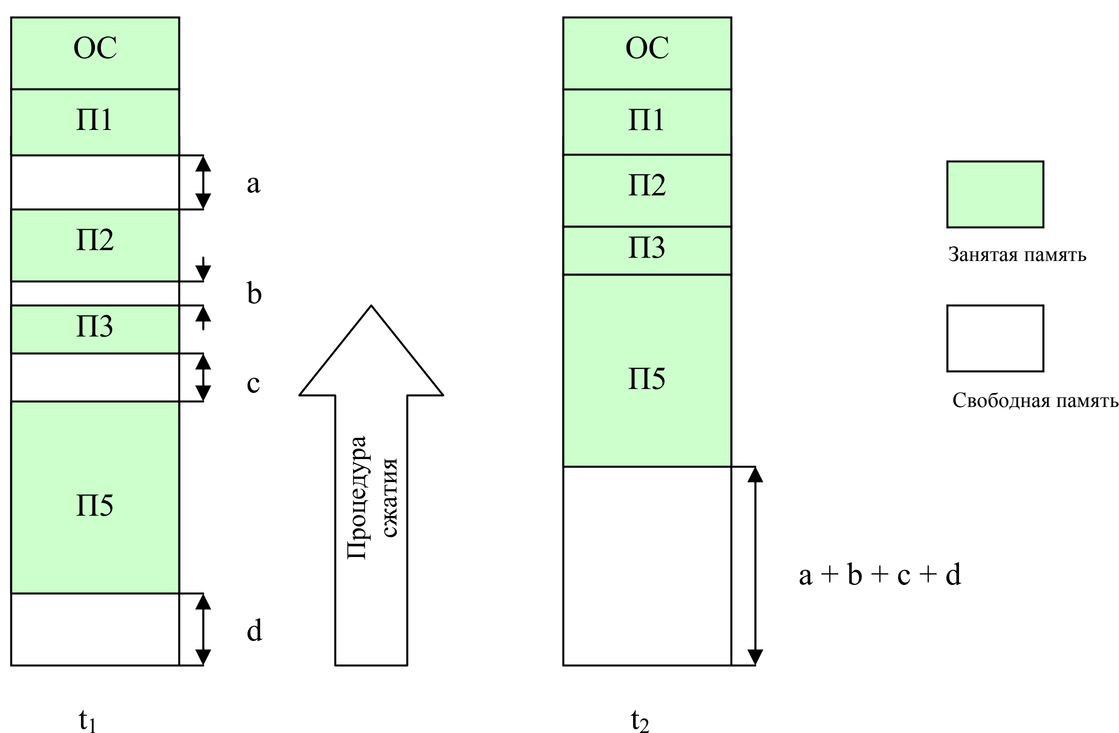

Рисунок 26 – Распределение памяти перемещаемыми разделами

Хотя процедура сжатия и приводит к более эффективному использованию памяти, она может потребовать значительного времени, что часто перевешивает преимущества данного метода.

Такой подход был использован в ранних версиях OS/2, в которых память распределялась сегментами, а возникавшая при этом фрагментация устранялась путем периодического перемещения сегментов.

3.3 Методы распределения памяти с подкачкой на жесткий диск

Оперативной памяти иногда оказывается недостаточно для того, чтобы вместить все текущие процессы, и тогда избыток процессов приходится хранить на диске, а для обработки динамически переносить их в память.

Такая подмена (виртуализация) оперативной памяти дисковой памятью позволяет повысить уровень мультипрограммирования – объем оперативной памяти компьютера не столь жестко ограничивает количество одновременно выполняемых

процессов, поскольку суммарный объем памяти, занимаемой образами этих процессов, может существенно превосходить имеющийся объем оперативной памяти. Виртуальным называется ресурс, который пользователю или пользовательской программе представляется обладающим свойствами, которыми он в действительности не обладает.

Виртуализация оперативной памяти осуществляется совокупностью программных модулей операционной системы и аппаратных схем процессора и включает решение следующих задач:

- размещение данных в запоминающих устройствах разного типа, например часть кодов программы – в оперативной памяти, а часть – на диске;
- выбор образов процессов или их частей для перемещения из оперативной памяти на диск и обратно;
- перемещение по мере необходимости данных между памятью и диском;
- преобразование виртуальных адресов в физические.

Виртуализация памяти может быть осуществлена на основе двух различных подходов:

- свопинг (swapping) или обычная подкачка – образы процессов выгружаются на диск и возвращаются в оперативную память целиком;
- виртуальная память (virtual memory) – между оперативной памятью и диском перемещаются части (сегменты, страницы и т. п.) образов процессов.

Недостатком свопинга является то, что при его осуществлении происходит перемещение избыточной информации, а также операционные системы, поддерживающие свопинг не способны загрузить для выполнения процесс, виртуальное адресное пространство которого превышает имеющуюся в наличии свободную память. Именно из-за указанных недостатков свопинг как основной механизм управления памятью почти не используется в современных операционных системах.

В настоящее время все множество реализаций виртуальной памяти может быть представлено тремя классами.

- Страничная виртуальная память организует перемещение данных между памятью и диском страницами – частями виртуального адресного пространства, фиксированного и сравнительно небольшого размера.
- Сегментная виртуальная память предусматривает перемещение данных сегментами – частями виртуального адресного пространства произвольного размера, полученными с учетом смыслового значения данных.
- Сегментно-страничная виртуальная память использует двухуровневое деление: виртуальное адресное пространство делится на сегменты, а затем сегменты делятся на страницы. Единицей перемещения данных здесь является страница. Этот способ управления памятью объединяет в себе элементы обоих предыдущих подходов.

Для временного хранения сегментов и страниц на диске отводится либо специальная область, либо специальный файл, которые во многих операционных системах по традиции продолжают называть областью, или файлом свопинга (подкачки), хотя перемещение информации между оперативной памятью и диском

осуществляется уже не в форме полного замещения одного процесса другим, а частями.

3.3.1 Страничная организация памяти

На рисунке 27 показана схема страничного распределения памяти. Виртуальное адресное пространство каждого процесса делится на части одинакового, фиксированного для данной системы размера, называемые виртуальными страницами (virtual pages). В общем случае размер виртуального адресного пространства процесса не кратен размеру страницы, поэтому последняя страница каждого процесса дополняется фиктивной областью.

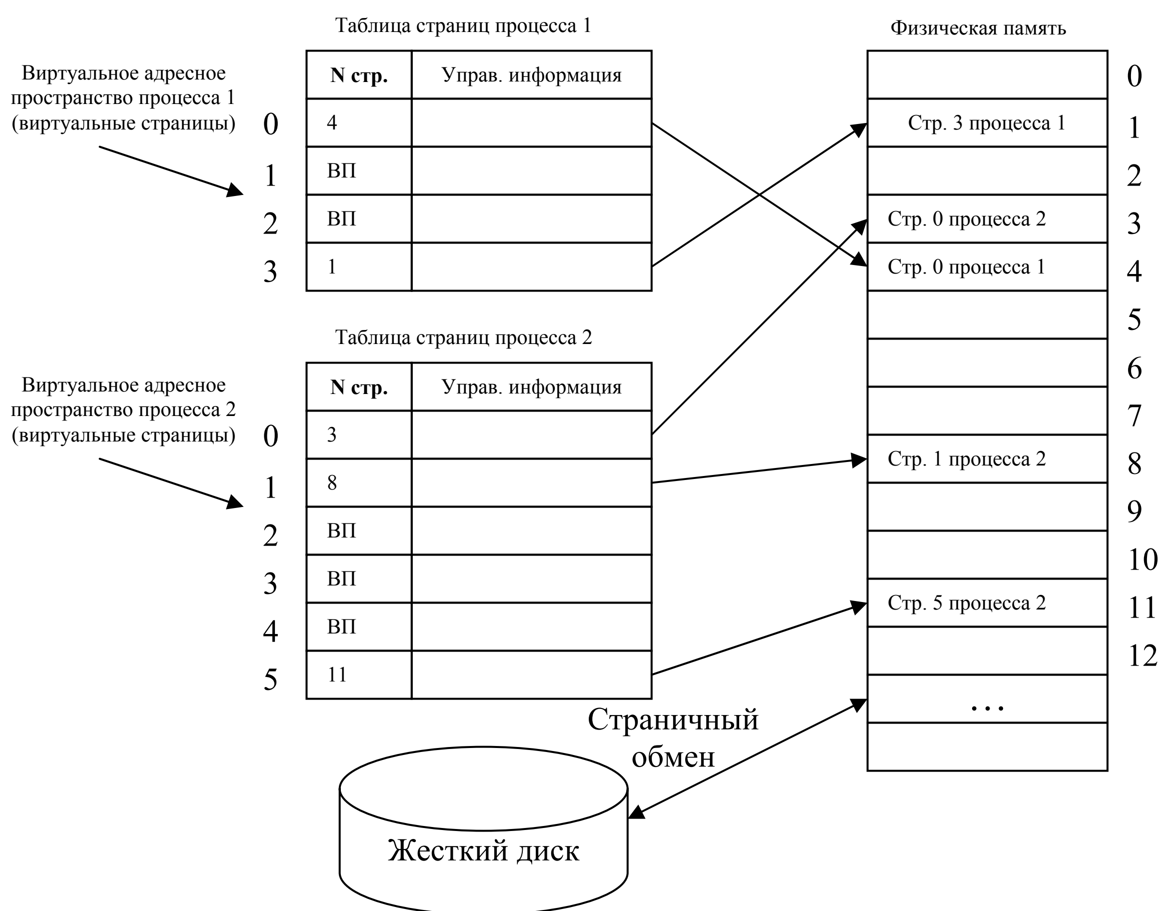

Рисунок 27 – Страничное распределение памяти

Вся оперативная память машины также делится на части такого же размера, называемые физическими страницами (или блоками, или кадрами).

Размер страницы выбирается равным степени двойки: 512, 1024, 4096 байт и т. д. Это позволяет упростить механизм преобразования адресов.

Операционная система при создании процесса загружает в оперативную память несколько его виртуальных страниц (начальные страницы кодового сегмента и сегмента данных). Копия всего виртуального адресного пространства процесса находится на диске. Смежные виртуальные страницы не обязательно располагаются в смежных физических страницах. Для каждого процесса операционная система

создает таблицу страниц – информационную структуру, содержащую записи обо всех виртуальных страницах процесса.

Запись таблицы, называемая дескриптором страницы, включает следующую информацию:

- номер физической страницы, в которую загружена данная виртуальная страница;
- признак присутствия, устанавливаемый в единицу, если виртуальная страница находится в оперативной памяти;
- признак модификации страницы, который устанавливается в единицу всякий раз, когда производится запись по адресу, относящемуся к данной странице;
- признак обращения к странице, называемый также битом доступа, который устанавливается в единицу при каждом обращении по адресу, относящемуся к данной странице.

Информация из таблиц страниц используется для решения вопроса о необходимости перемещения той или иной страницы между памятью и диском, а также для преобразования виртуального адреса в физический. Виртуальные адреса не передаются напрямую на шину памяти, а передаются на диспетчер памяти (MMU – Memory Management Unit), которые отображает виртуальные адреса на физические (Рисунок 28). Диспетчер памяти в настоящее время обычно встраивается в микросхему процессора.

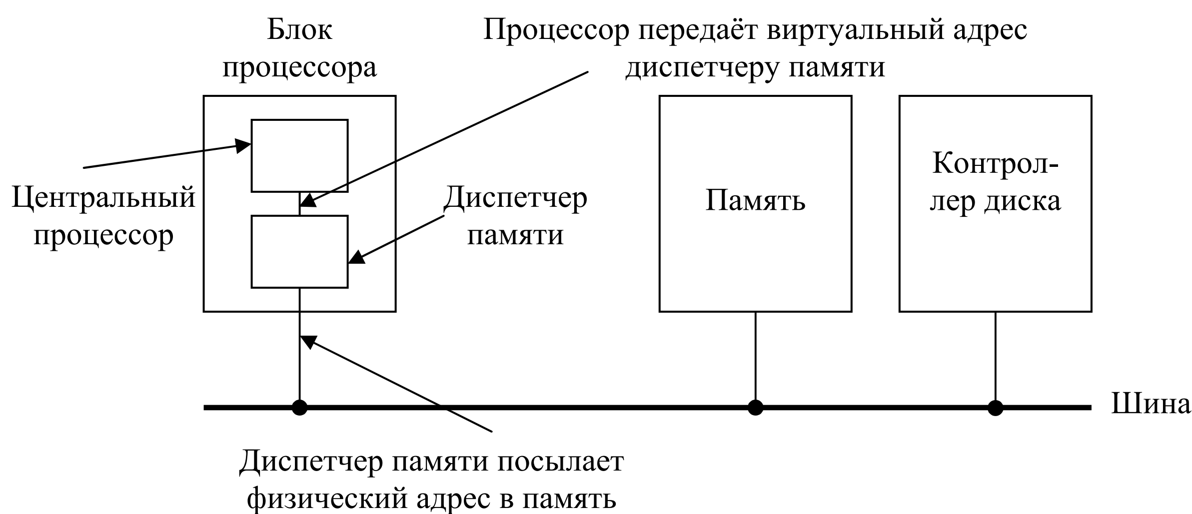

Рисунок 28 – Расположение и функции диспетчера памяти

Например, если использовать команду mov [3] для получения доступа к адресу 0

```
MOV REG, 0
```

виртуальный адрес 0 передаётся в MMU. Предположим, что размер страницы 4096 байт, тогда, руководствуясь таблицей страниц процесса 1 (Рисунок 27), MMU преобразует команду следующим образом:

```
MOV REG, 16384
```

При каждом обращении к памяти выполняется поиск номера виртуальной страницы, содержащей требуемый адрес, затем по этому номеру определяется нужный элемент таблицы страниц, и из него извлекается описывающая страницу

информация. Далее анализируется признак присутствия, и, если данная виртуальная страница находится в оперативной памяти, то выполняется преобразование виртуального адреса в физический, то есть виртуальный адрес заменяется указанным в записи таблицы физическим адресом. Если же нужная виртуальная страница в данный момент выгружена на диск, то происходит так называемое страничное прерывание. Выполняющийся процесс переводится в состояние ожидания, и активизируется другой процесс из очереди процессов, находящихся в состоянии готовности. Параллельно программа обработки страничного прерывания находит на диске требуемую виртуальную страницу (для этого операционная система должна помнить положение вытесненной страницы в страничном файле диска) и пытается загрузить ее в оперативную память. Если в памяти имеется свободная физическая страница, то загрузка выполняется немедленно, если же свободных страниц нет, то на основании принятой в данной системе стратегии замещения страниц решается вопрос о том, какую страницу следует выгрузить из оперативной памяти.

Важным фактором, влияющим на производительность системы, является частота страничных прерываний, на которую, в свою очередь, влияют размер страницы и принятые в данной системе правила выбора страниц для выгрузки и загрузки. При неправильно выбранной стратегии замещения страниц могут возникать ситуации, когда система тратит большую часть времени впустую, на подкачку страниц из оперативной памяти на диск и обратно.

При выборе страницы на выгрузку могут быть использованы различные критерии, смысл которых сводится к одному: на диск выталкивается страница, к которой в будущем, начиная с данного момента, дольше всего не будет обращений.

При страничной организации памяти есть 2 проблемы: 1) таблица страниц может быть слишком большой; 2) отображение страниц должно быть быстрым.

Для решения первой проблемы используют многоуровневые таблицы памяти [14], при использовании которых в памяти находятся только части таблицы страниц.

Для решения второй компьютер снабжается небольшим аппаратным устройством, служащим для отображения виртуальных адресов в физические без прохода по таблице страниц. Это устройство называется буфером быстрого преобразования адреса (TLB – Translation Lookaside Buffer) или ассоциативной памятью.

Большинство программ склонно делать огромное количество обращений к небольшому количеству страниц, а не наоборот. Таким образом, в таблице страниц только малая доля записей читается интенсивно, остальная часть едва ли вообще используется. Поэтому эта малая доля записей копируется в TLB, который работает гораздо быстрее стандартного обращения к таблице страниц.

Когда происходит страничное прерывание, операционная система должна выбрать страницу для удаления из памяти, чтобы освободить место для страницы, которую нужно перенести в память. Если удаляемая страница была изменена за время своего присутствия в памяти, ее необходимо переписать на диск, чтобы обновить копию, хранящуюся там. Однако если страница не была модифицирована (например, она содержит текст программы), копия на диске уже является самой новой и ее не надо переписывать. Тогда страница, которую нужно прочитать, просто

считывается поверх выгружаемой страницы.

Хотя в принципе можно при каждом страничном прерывании выбирать случайную страницу для удаления из памяти, производительность системы заметно повышается, когда предпочтение отдается редко используемой странице. Ниже описаны некоторые наиболее важные алгоритмы замещения страниц.

1Оптимальный алгоритм

В тот момент, когда происходит страничное прерывание, в памяти находится некоторый набор страниц. К одной из этих страниц будет обращаться следующая команда процессора (к странице, содержащей требуемую команду). На другие страницы, возможно, не будет ссылок в течение следующих 10, 100 или даже 1000 команд. Каждая страница может быть помечена количеством команд, которые будут выполняться перед первым обращением к этой странице. Оптимальный страничный алгоритм просто сообщает, что должна быть выгружена страница с наибольшей меткой.

С этим алгоритмом связана только одна проблема: он невыполним. В момент страничного прерывания операционная система не имеет возможности узнать, когда произойдет следующее обращение к каждой странице.

2 Алгоритм NRU – не использовавшаяся в последнее время страница

Чтобы дать возможность операционной системе собирать полезные статистические данные о том, какие страницы используются, а какие – нет, большинство компьютеров с виртуальной памятью поддерживают два статусных бита, связанных с каждой страницей. Бит ***R*** (Referenced – обращения) устанавливается всякий раз, когда происходит обращение к странице (чтение или запись). Бит ***M*** (Modified – изменение) устанавливается, когда страница записывается (то есть изменяется). Биты содержатся в каждом элементе таблицы страниц. Если аппаратное обеспечение не поддерживает эти биты, их можно смоделировать.

Биты ***R*** и ***M*** могут использоваться для построения простого алгоритма замещения страниц, описанного ниже. Когда процесс запускается, оба страничных бита для всех его страниц операционной системой установлены на 0. Периодически (например, при каждом прерывании по таймеру) бит ***R*** очищается, чтобы отличить страницы, к которым давно не происходило обращения от тех, на которые были ссылки. Когда возникает страничное прерывание, операционная система проверяет все страницы и делит их на четыре категории на основании текущих значений битов ***R*** и ***M***:

- класс 0: не было обращений и изменений;
- класс 1: не было обращений, страница изменена;
- класс 2: было обращение, страница не изменена;
- класс 3: произошло и обращение, и изменение.

Хотя класс 1 на первый взгляд кажется невозможным, такое случается, когда у страницы из класса 3 бит ***R*** сбрасывается во время прерывания по таймеру. Прерывания по таймеру не стирают бит ***M***, потому что эта информация необходима для того, чтобы знать, нужно ли переписывать страницу на диске или нет. Поэтому если бит ***R*** устанавливается на ноль, а ***M*** остается нетронутым, страница попадает в класс 1.

Алгоритм NRU (Not Recently Used) удаляет страницу с помощью случайного поиска в непустом классе с наименьшим номером. Привлекательность алгоритма NRU заключается в том, что он легок для понимания, умеренно сложен в реализации и дает производительность, которая может вполне оказаться достаточной.

3 Алгоритм FIFO – первым прибыл – первым обслужен

Операционная система поддерживает список всех страниц, находящихся в данный момент в памяти, в котором первая страница является старейшей, а страницы в хвосте списка попали в него совсем недавно. Когда происходит страничное прерывание, выгружается из памяти страница в голове списка, а новая страница добавляется в его конец. Данный алгоритм не используется, так как он может удалить наиболее часто вызываемую страницу.

4 Алгоритм «вторая попытка»

Модификация предыдущего алгоритма. Когда происходит страничное прерывание, то у самой «старейшей» страницы проверяется бит ***R***. Если он равен 0, т.е. страница не только дольше всех в памяти, но ещё и не используется, то страница заменяется новой. Если же бит равен 1, то странице даётся вторая попытка – бит изменяется в 0, а сама страница перемещается в конец очереди, т.е. становится самой «молодой».

5 Алгоритм «часы»

Предыдущий алгоритм является слишком неэффективным, потому что постоянно передвигает страницы по списку. Поэтому лучше хранить все страничные блоки в кольцевом списке в форме часов, как показано на рисунке 29. Стрелка указывает на старейшую страницу.

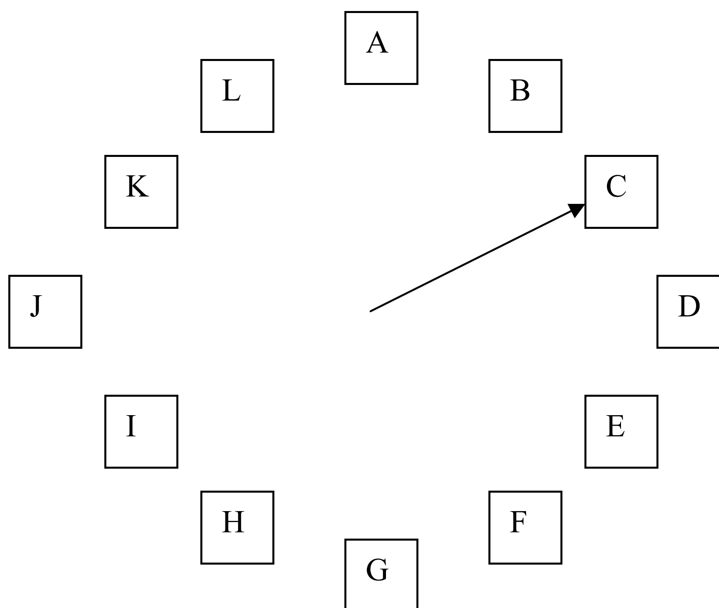

Рисунок 29 – Кольцевой список в алгоритме «часы»

Когда происходит страничное прерывание, проверяется та страница, на которую направлена стрелка. Если ее бит ***R*** равен 0, страница выгружается, на ее место в часовой круг встает новая страница, а стрелка сдвигается вперед на одну позицию. Если бит ***R*** равен 1, то он сбрасывается, стрелка перемещается к

следующей странице. Этот процесс повторяется до тех пор, пока не находится та страница, у которой бит ***R*** = 0.

6 Алгоритм LRU – страница, не использовавшаяся дольше всего

Страницы, к которым происходит многократное обращение в нескольких последних командах, вероятно, также будут часто использоваться в следующих инструкциях. И наоборот, страницы, к которым ранее не возникало обращений, не будут употребляться в течение долгого времени. Эта идея привела к следующему реализуемому алгоритму: когда происходит страничное прерывание, выгружается из памяти страница, которая не использовалась дольше всего. Такая стратегия замещения страниц называется LRU (Least Recently Used – «менее недавно»).

Для полного осуществления алгоритма LRU необходимо поддерживать связный список всех содержащихся в памяти страниц, где последняя использовавшаяся страница находится в начале списка, а та, к которой дольше всего не было обращений – в конце. Сложность заключается в том, что список должен обновляться при каждом обращении к памяти. Поиск страницы, ее удаление, а затем вставка в начало списка – это операции, поглощающие очень много времени, даже если они выполняются аппаратно (если предположить, что необходимое оборудование можно сконструировать). Способы реализации данного алгоритма описаны в работе Э. Таненбаума [14], однако из-за необходимости аппаратной поддержки разработчики операционных систем редко им пользуются.

7 Алгоритм «старение»

Одна из разновидностей схемы LRU называется алгоритмом NFU (Not Frequently Used – редко использовавшаяся страница). Для него необходим программный счетчик, связанный с каждой страницей в памяти, изначально равный нулю. Во время каждого прерывания по таймеру операционная система исследует все страницы в памяти. Бит ***R*** каждой страницы (он равен 0 или 1) прибавляется к счетчику. В сущности, счетчики пытаются отследить, как часто происходило обращение к каждой странице. При страничном прерывании для замещения выбирается страница с наименьшим значением счетчика.

Основная проблема, возникающая при работе с алгоритмом NFU, заключается в том, что он никогда ничего не забывает. Например, в многоходовом компиляторе страницы, которые часто использовались во время первого прохода, могут все еще иметь высокое значение счетчика при более поздних проходах. Небольшие изменения позволяют решить эту проблему и достаточно хорошо моделировать алгоритм LRU:

- каждый счетчик сдвигается вправо на один разряд перед прибавлением бита ***R***;
- бит ***R*** добавляется в крайний слева, а не в крайний справа бит счетчика.

В таблице 3 продемонстрировано, как работает видоизмененный алгоритм, известный под названием «старение» (aging). Между тиком 0 и тиком 1 произошло обращение к страницам 0, 2, 4 и 5, их биты ***R*** приняли значение 1, остальные сохранили значение 0. После того как шесть соответствующих счетчиков сдвинулись на разряд, и бит ***R*** занял крайнюю слева позицию, счетчики получили значения, показанные в первом столбце (Такт 0). Остальные четыре колонки таблицы изображают шесть счетчиков после следующих четырех тиков часов.

Когда происходит страничное прерывание, удаляется та страница, чей счетчик имеет наименьшую величину. Ясно, что счетчик страницы, к которой не было обращений, скажем, за четыре тика, будет начинаться с четырех нулей и, таким образом, иметь более низкое значение, чем счетчик страницы, на которую не ссылались в течение только трех тиков часов.

Таблица 3 – Пример работы алгоритма «старение»: в строках – 0-5 страницы памяти; в столбцах – биты ***R*** для страниц

C	**R**				
	Такт 0: 101011	**Такт 1: 110010**	**Такт 2: 110101**	**Такт 3: 100010**	**Такт 4: 011000**
0	10000000	11000000	11100000	11110000	01111000
1	00000000	10000000	11000000	01100000	10110000
2	10000000	01000000	00100000	00010000	10001000
3	00000000	00000000	10000000	01000000	00100000
4	10000000	11000000	01100000	10110000	01011000
5	10000000	01000000	10100000	01010000	00101000

8 Алгоритм «рабочий набор»

В простейшей схеме страничной подкачки в момент запуска процессов нужные им страницы отсутствуют в памяти. Как только центральный процессор пытается выбрать первую команду, он получает страничное прерывание, побуждающее операционную систему перенести в память страницу, содержащую первую инструкцию. Обычно следом быстро происходят страничные прерывания для глобальных переменных и стека. Через некоторое время в памяти скапливается большинство необходимых процессу страниц, и он приступает к работе с относительно небольшим количеством ошибок из-за отсутствия страниц. Этот метод называется замещением страниц по запросу (demand paging), потому что страницы загружаются в память по требованию, а не заранее.

Большинство процессов характеризуется тем, что во время выполнения любой фазы обращается к сравнительно небольшой части своих страниц.

Рабочий набор – множество страниц, которое процесс использует в данный момент.

Базовая идея алгоритма замещения страниц заключается в том, чтобы найти страницу, не включенную в рабочий набор, и выгрузить ее. Каждая запись информации о странице содержит (по крайней мере) два элемента информации: приближенное время, в которое страница использовалась в последний раз, и бит ***R*** (обращения).

Алгоритм работает следующим образом. Предполагается, что аппаратное обеспечение устанавливает биты ***R*** и ***M***, как в алгоритме NRU. Предполагается также, что периодическое прерывание по таймеру вызывает запуск программы, очищающей бит ***R*** при каждом тике часов. При каждом страничном прерывании исследуется таблица страниц и ищется страница, подходящая для удаления из памяти. Эта страница должна соответствовать следующим параметрам: бит ***R*** равен 0 и время последнего использования больше некоторой заранее заданной величины ***T***. Однако сканирование таблицы продолжается, обновляя остальные записи. Если

проверена вся таблица, а кандидат на удаление не найден, это означает, что все страницы входят в рабочий набор. В этом случае, если были найдены одна или больше страниц с битом ***R*** = 0, удаляется та из них, которая имеет наибольший возраст.

Данный алгоритм очень громоздок, так как при каждом страничном прерывании следует проверять таблицу страниц до тех пор, пока не определится местоположение подходящего кандидата.

9 Алгоритм WSClock

Этот алгоритм является модификацией предыдущего. Для его использования необходима структура данных в виде кольцевого списка (Рисунок 29). В исходном положении этот список пустой. Когда загружается первая страница, она добавляется в список. По мере прихода страниц они поступают в список, формируя кольцо. Каждая запись, кроме бита ***R*** и бита ***M***, содержит поле «время последнего использования» из базового алгоритма «рабочий набор».

Как и в случае алгоритма «часы», при каждом страничном прерывании первой проверяется та страница, на которую указывает стрелка. Если бит ***R*** равен 1, это значит, что страница использовалась в течение последнего такта часов, поэтому она не является идеальным кандидатом на удаление. Тогда бит ***R*** устанавливается на 0, стрелка передвигается на следующую страницу и для нее повторяется алгоритм.

Если в момент проверки бит ***R*** равен 0 и время последнего использования больше некоторой заранее заданной величины ***T***, то проверяется бит ***M*** – были ли изменения. Если нет, то страница удаляется. Если изменения были – страница помечается как необходимая для копирования, а стрелка «часов» сдвигается.

Если стрелка часов обходит круг и возвращается обратно, то возможно два варианта:

1) запланирована операция переноса страницы на диск;

2) ничего не запланировано.

В первом случае выбирается первая попавшаяся страница без изменений с битом ***R*** равным 0. Во втором случае предъявляются права на любую страницу.

Двумя наилучшими алгоритмами являются «старение» и WSClock. Оба обеспечивают хорошую постраничную подкачку и могут быть реализованы за разумную цену.

Недостатки страничного распределения памяти – размеры страниц и частота страничных прерываний сильно влияют на производительность, все данные находятся перемешенными друг с другом.

3.3.2 Сегментная организация памяти

При страничной организации виртуальное адресное пространство процесса делится на равные части механически, без учета смыслового значения данных. В одной странице могут оказаться и коды команд, и инициализируемые переменные, и массив исходных данных программы. Такой подход не позволяет обеспечить дифференцированный доступ к разным частям программы, а это свойство могло бы быть очень полезным во многих случаях. Например, можно было бы запретить

обращаться с операциями записи в сегмент программы, содержащий коды команд, разрешив эту операцию для сегментов данных.

Кроме того, разбиение виртуального адресного пространства на части по типу данных делает принципиально возможным совместное использование фрагментов программ разными процессами. Пусть, например, двум процессам требуется одна и та же подпрограмма, которая к тому же обладает свойством реентерабельности (Реентерабельность – свойство повторной входимости кода, которое позволяет одновременно использовать его несколькими процессами [11]). При выполнении реентерабельного кода процессы не изменяют его, поэтому в память достаточно загрузить только одну копию кода. Тогда коды этой подпрограммы могут быть оформлены в виде отдельного сегмента и включены в виртуальные адресные пространства обоих процессов. При отображении в физическую память сегменты, содержащие коды подпрограммы из обоих виртуальных пространств, проецируются на одну и ту же область физической памяти. Таким образом оба процесса получат доступ к одной и той же копии подпрограммы (Рисунок 30).

Виртуальное адресное пространство процесса делится на части – сегменты, размер которых определяется с учетом смыслового значения содержащейся в них информации. Отдельный сегмент может представлять собой подпрограмму, массив данных и т. п. Деление виртуального адресного пространства на сегменты осуществляется компилятором на основе указаний программиста или по умолчанию, в соответствии с принятыми в системе соглашениями. Максимальный размер сегмента определяется разрядностью виртуального адреса, например при 32-разрядной организации процессора он равен 4 Гбайт. При этом максимально возможное виртуальное адресное пространство процесса представляет собой набор из ***N*** виртуальных сегментов, каждый размером по 4 Гбайт.

Рисунок 30 – Распределение памяти сегментами

При загрузке процесса в оперативную память помещается только часть его сегментов, полная копия виртуального адресного пространства находится в дисковой памяти. Для каждого загружаемого сегмента операционная система подыскивает непрерывный участок свободной памяти достаточного размера. Смежные в виртуальной памяти сегменты одного процесса могут занимать в оперативной памяти несмежные участки. Если во время выполнения процесса происходит обращение по виртуальному адресу, относящемуся к сегменту, который в данный момент отсутствует в памяти, то происходит прерывание. Операционная система приостанавливает активный процесс, запускает на выполнение следующий процесс из очереди, а параллельно организует загрузку нужного сегмента с диска.

На этапе создания процесса во время загрузки его образа в оперативную память система создает таблицу сегментов процесса (аналогичную таблице страниц), в которой для каждого сегмента указывается:

- базовый физический адрес сегмента в оперативной памяти;
- размер сегмента;
- правила доступа к сегменту;
- признаки модификации, присутствия и обращения к данному сегменту, а также некоторая другая информация.

Если виртуальные адресные пространства нескольких процессов включают один и тот же сегмент, то в таблицах сегментов этих процессов делаются ссылки на один и тот же участок оперативной памяти, в который данный сегмент загружается в единственном экземпляре.

Недостатки сегментного распределения памяти:

1) более медленное по сравнению со страничным распределением преобразование виртуального адреса в физический;
2) избыточность, связанная с излишней загрузкой памяти, т.к. во многих случаях информация, находящаяся в сегменте нужна лишь частично;
3) фрагментация, которая возникает из-за непредсказуемости размеров сегментов.

В процессе работы системы в памяти образуются небольшие участки свободной памяти, в которые не может быть загружен ни один сегмент. Суммарный объем, занимаемый фрагментами, может составить существенную часть общей памяти системы, приводя к ее неэффективному использованию.

Одним из существенных отличий сегментной организации памяти от страничной является возможность задания дифференцированных прав доступа процесса к его сегментам. Например, один сегмент данных, содержащий исходную информацию для приложения, может иметь права доступа «только чтение», а сегмент данных, представляющий результаты – «чтение и запись». Это свойство дает принципиальное преимущество сегментной модели памяти над страничной.

3.3.3 Сегментно-страничная организация памяти

Данный метод представляет собой комбинацию страничного и сегментного механизмов управления памятью и направлен на реализацию достоинств обоих подходов.

Так же как и при сегментной организации памяти, виртуальное адресное пространство процесса разделено на сегменты. Это позволяет определять разные права доступа к разным частям кодов и данных программы.

Перемещение данных между памятью и диском осуществляется не сегментами, а страницами. Для этого каждый виртуальный сегмент и физическая память делятся на страницы равного размера, что позволяет более эффективно использовать память, сократив до минимума фрагментацию.

3.4 Кэширование данных

Кэш-память, или просто кэш (cache) – это способ совместного функционирования двух типов запоминающих устройств, отличающихся временем доступа и стоимостью хранения данных, который за счет динамического копирования в «быстрое» запоминающее устройство наиболее часто используемой информации из «медленного» запоминающего устройства позволяет, с одной стороны, уменьшить среднее время доступа к данным, а с другой стороны, экономить более дорогую быстродействующую память [11].

Кэш-памятью, или кэшем, часто называют не только способ организации работы двух типов запоминающих устройств, но и одно из устройств – «быстрое» запоминающее устройство.

Схема кэширования представлена на рисунке 31.

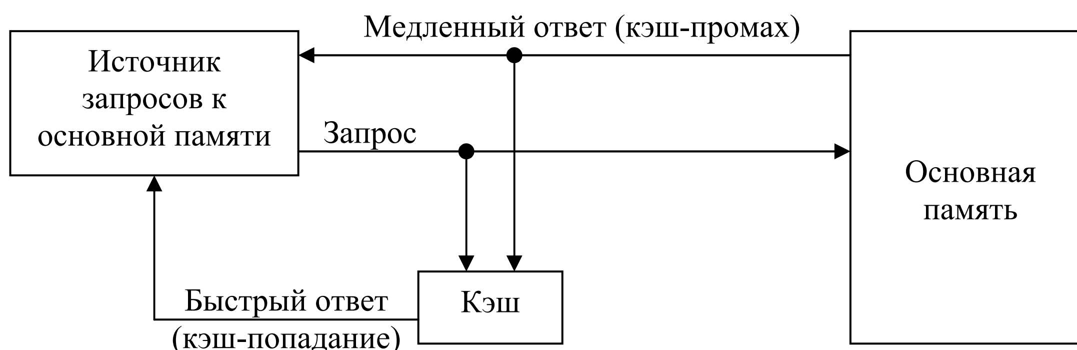

Рисунок 31 – Схема функционирования кэш-памяти

Содержимое кэш-памяти представляет собой совокупность *записей* обо всех загруженных в нее элементах данных из основной памяти. Каждая запись об элементе данных включает в себя:

- значение элемента данных;
- адрес, который этот элемент данных имеет в основной памяти;
- дополнительную информацию, которая используется для реализации алгоритма замещения данных в кэше и обычно включает признак модификации и признак действительности данных.

При каждом обращении к основной памяти по физическому адресу просматривается содержимое кэш-памяти с целью определения, не находятся ли там нужные данные. Кэш-память не является адресуемой, поэтому поиск нужных данных осуществляется по содержимому, которое взято из запроса. Далее возможен один из двух вариантов развития событий:

- если данные обнаруживаются в кэш-памяти, то есть произошло кэш-попадание (cache-hit), они считываются из нее и результат передается источнику запроса;
- если нужные данные отсутствуют в кэш-памяти, то есть произошел кэш-промах (cache-miss), они считываются из основной памяти, передаются источнику запроса и одновременно с этим копируются в кэш-память.

Использование кэш-памяти имеет смысл только при высокой вероятности кэш-попадания. Вероятность обнаружения данных в кэше зависит от разных факторов, таких, например, как объем кэша, объем кэшируемой памяти, алгоритм замещения данных в кэше, особенности выполняемой программы, время ее работы, уровень мультипрограммирования и других особенностей вычислительного процесса. В большинстве реализаций кэш-памяти процент кэш-попаданий оказывается весьма высоким – свыше 90 %. Такое высокое значение вероятности нахождения данных в кэш-памяти объясняется наличием у данных объективных свойств: пространственной и временной локальности.

- *Временная локальность.* Если произошло обращение по некоторому адресу, то следующее обращение по тому же адресу с большой вероятностью произойдет в ближайшее время.
- *Пространственная локальность.* Если произошло обращение по некоторому адресу, то с высокой степенью вероятности в ближайшее время произойдет обращение к соседним адресам.

В процессе работы содержимое кэш-памяти постоянно обновляется, а значит, время от времени данные из нее должны вытесняться. Вытеснение означает либо простое объявление свободной соответствующей области кэш-памяти (сброс бита действительности), если вытесняемые данные за время нахождения в кэше не были изменены, либо в дополнение к этому копирование данных в основную память, если они были модифицированы. Алгоритм замены данных в кэш-памяти существенно влияет на ее эффективность. В идеале такой алгоритм должен, во-первых, быть максимально быстрым, чтобы не замедлять работу кэш-памяти, а во-вторых, обеспечивать максимально возможную вероятность кэш-попаданий.

Наличие в компьютере двух копий данных (в основной памяти и в кэше) порождает проблему согласования данных. Если происходит запись в основную память по некоторому адресу, а содержимое этой ячейки находится в кэше, то в результате соответствующая запись в кэше становится недостоверной. Есть два подхода к решению этой проблемы:

1 *Сквозная запись (write through).* При каждом запросе к основной памяти, в том числе и при записи, просматривается кэш. Если данные по запрашиваемому адресу отсутствуют, то запись выполняется только в основную память. Если же данные, к которым выполняется обращение, находятся в кэше, то запись выполняется одновременно в кэш и основную память.

2 *Обратная запись (write back).* Аналогично при возникновении запроса к памяти выполняется просмотр кэша, и если запрашиваемых данных там нет, то запись выполняется только в основную память. В противном же случае запись производится только в кэш-память, при этом в описателе данных делается специальная отметка (признак модификации), которая указывает на то, что при

вытеснении этих данных из кэша необходимо переписать их в основную память, чтобы актуализировать устаревшее содержимое основной памяти.

Контрольные вопросы по разделу

1 Может ли прикладной процесс использовать системную часть виртуальной памяти?

2 Какое из этих двух утверждений верно?

А) все виртуальные адреса заменяются на физические во время загрузки программы в оперативную память;

В) виртуальные адреса заменяются на физические во время выполнения программы в момент обращения по данному виртуальному адресу.

3 Распределение памяти перемещаемыми разделами основано на применении процедуры сжатия. Имеет ли смысл использовать данную процедуру при страничном распределении? А при сегментном?

4 На что влияет размер страницы?

5 У маленького компьютера четыре страничных блока. Во время первого тика часов биты R равны 0111 (у страницы 0 бит R равен 0, у остальных - 1). Во время последующих тиков часов биты R принимают значения 1011,1010, 1101, 0010, 1010, 1100 и 0001. Считая, что используется алгоритм старения с 8-разрядным счетчиком, напишите четыре значения, которые примет счетчик после последнего тика.

6 В кэше хранятся данные, которые наиболее активно используются в последнее время. Каким образом система определяет, какие данные должны быть загружены в кэш?

7 Как обеспечивается согласование данных в кэше с помощью методов обратной и сквозной записи?

8 Перечислите недостатки страничной организации памяти.

9 Перечислите недостатки сегментной организации памяти.

4 Аппаратная поддержка мультипрограммирования на примере процессора Pentium

Аппаратные средства поддержки мультипрограммирования имеются во всех современных процессорах. Несмотря на различия в реализации, для большинства типов процессоров эти средства имеют общие черты.

Основным режимом работы процессора Pentium (включая современные модели Pentium IV) является защищенный режим (protected mode). Для совместимости с программным обеспечением, разработанным для предшествующих моделей процессоров Intel (модели 8086), в процессорах Pentium предусмотрен так называемый реальный резким (real mode). В реальном режиме процессор Pentium выполняет 16-разрядные инструкции и адресует 1 Мбайт памяти.

4.1 Регистры

В организации вычислительного процесса важную роль играют регистры процессора. В процессорах Pentium эти регистры делятся на несколько групп:

- регистры общего назначения;
- сегментные регистры;
- указатель инструкций;
- регистр флагов;
- регистры управления памятью;
- регистры управления процессором;
- регистры отладки и тестирования;
- машинно-специфичные регистры.

Регистры общего назначения

В процессоре Pentium имеется восемь 32-разрядных регистров общего назначения. Четыре из них, которые можно условно назвать A, B, C и D, используются для временного хранения операндов арифметических, логических и других команд. Программист может обращаться к этим регистрам как к единому целому, используя обозначения EAX, EBX, ECX, EDX, а также к некоторым их частям, как это показано на рисунке 32. Здесь обозначение AL (L — Low) относится к первому, самому младшему байту регистра EAX, AH (H — High) — к следующему по старшинству байту, а AX обозначает оба младших байта регистра. Приставка E в обозначении этих регистров (а также некоторых других) образована от слова extended (расширенный), что указывает на то, что в прежних моделях процессоров Intel эти регистры были 16-разрядными, а затем их разрядность была увеличена до 32 бит.

Другие четыре регистра общего назначения – ESI, EDI, EBP и ESP – предназначены для задания смещения адреса относительно начала некоторого сегмента данных. Эти регистры используются совместно с регистрами сегментов в системе адресации процессора Pentium для задания виртуального адреса, который затем с помощью таблиц страниц отображается на физический адрес. Эти регистры

могут применяться для хранения всевозможных переменных только тогда, когда они не используются по назначению.

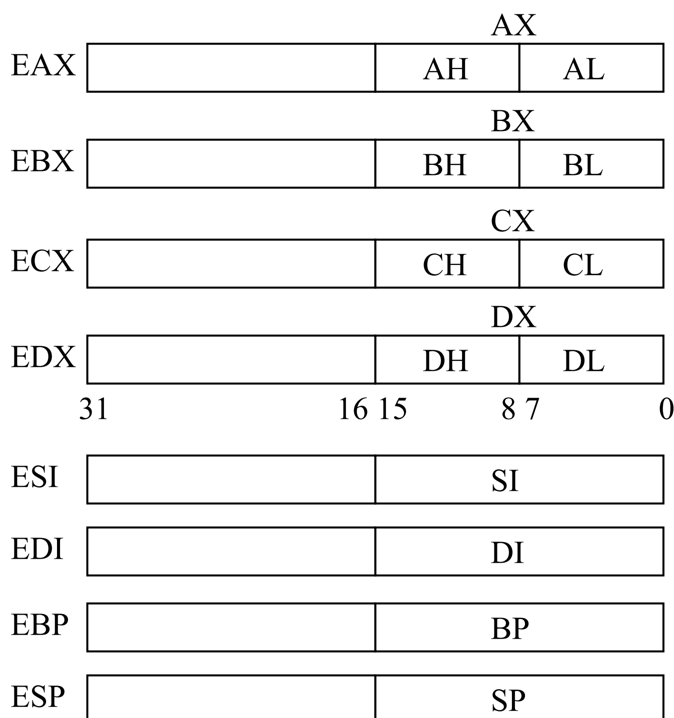

Рисунок 32 – Регистры общего назначения процессора Pentium

Сегментные регистры

Регистры сегментов CS, SS, DS, ES, FS и GS в защищенном режиме ссылаются на дескрипторы сегментов памяти – описатели, в которых содержатся базовый адрес, размер сегмента, атрибуты защиты и некоторые другие параметры. Регистры сегментов хранят 16-разрядное число, называемое селектором, которое хранит индекс дескриптора сегмента и уровень привилегий. Регистр CS (Code Segment) предназначен для хранения индекса дескриптора кодового сегмента, регистр SS (Stack Segment) – дескриптора сегмента стека, а остальные регистры используются для указания на дескрипторы сегментов данных. Все регистры сегментов, кроме CS, программно доступны, то есть в них можно загрузить новое значение селектора соответствующей командой (например, LDS). Значение регистра CS изменяется при выполнении команд межсегментных вызовов CALL и переходов JMP, а также при переключении задач. Запись в регистр CS приводит к тому, что будет выполнена не следующая команда, а команда из другого сегмента.

Указатель инструкций

Указатель инструкций EIP содержит смещение адреса следующей выполняемой команды, которое используется совместно с регистром CS для получения соответствующего виртуального адреса.

Регистр флагов

Регистр флагов EFLAGS содержит признаки, характеризующие результат выполнения операции: флаг знака (CF), флаг четности (PF), флаг полупереноса (AF), флаг нуля (ZF), флаг знака (SF), флаг ловушки или трассировки (TF), флаг прерываний (IF), флаг направления (DF), флаг переполнения (OF), флаг уровня привилегий 2 бита (IOPL), флаг вложенной задачи (NT), флаг продолжения задачи

(RF), флаг режима V86 (VM), флаг контроля за выравниванием (AC), флаг виртуального прерывания (VIF), флаг ожидания виртуального прерывания (VIP), флаг идентификации (ID).

Регистры управления памятью

Регистры управления памятью (другое название – регистры системных адресов) содержат адреса важных системных таблиц и структур, используемых при управлении процессами и памятью. Регистр GDTR (Global Descriptor Table Register) содержит физический 32-разрядный адрес глобальной таблицы дескрипторов GDT сегментов памяти, образующих общую часть виртуального адресного пространства всех процессов. Регистр IDTR (Interrupt Descriptor Table Register) хранит физический 32-разрядный адрес таблицы дескрипторов прерываний IDT, используемой для вызова процедур обработки прерываний в защищенном режиме работы процессора. Кроме этих адресов в регистрах GDTR и IDTR хранятся 16-битные лимиты, задающие ограничения на размер соответствующих таблиц. Общий размер этих регистров – 48 бит.

Два 10-байтных регистра хранят не физические адреса системных структур, а значения индексов дескрипторов этих структур в таблице GDT, что позволяет косвенно получить соответствующие физические адреса. Регистр TR (Task Register) содержит индекс дескриптора сегмента состояния задачи TSS. Регистр LDTR (Local Descriptor Table Register) содержит индекс дескриптора сегмента локальной таблицы дескрипторов LDT сегментов памяти, образующих индивидуальную часть виртуального адресного пространства процесса. 2 байта – селектор для GDT, 8 байт – сам дескриптор из GDT.

Регистры управления процессором

В процессоре Pentium имеется пять управляющих регистров – CR0, CR1, CR2, CR3 и CR4, которые хранят признаки и данные, характеризующие общее состояния процессора.

Регистр CR0 содержит все основные признаки, существенно влияющие на работу процессора, такие как реальный/защищенный режим работы, включение/выключение страничного механизма системы виртуальной памяти, а также признаки, влияющие на работу кэша и выполнение команд с плавающей точкой. Младшие два байта регистра CR0 имеют название Mashine State Word, MSW – «слово состояния машины». Это название использовалось в процессоре 80286 для обозначения управляющего регистра, имевшего аналогичное назначение.

Регистр CR1 в настоящее время не используется (зарезервирован).

Регистры CR2 и CR3 предназначены для поддержки работы системы виртуальной памяти. Регистр CR2 содержит линейный виртуальный адрес, который вызвал так называемый страничный отказ (отсутствие страницы в оперативной памяти или отказ из-за нарушения прав доступа). Регистр CR3 содержит физический адрес таблицы разделов, используемой страничным механизмом процессора.

В регистре CR4 хранятся признаки, разрешающие работу так называемых архитектурных расширений, например возможности использования страниц размером 4 Мбайт и т. п.

Регистры отладки и тестирования

Регистры отладки хранят значения точек останова, а регистры тестирования позволяют проверить корректность работы внутренних блоков процессора.

Машинно-специфичные регистры

Большая группа регистров (более ста), назначение которых отличается в разных моделях процессоров.

4.2 Привилегированные команды

Привилегированные команды – это команды, которые могут быть выполнены только при определенном уровне привилегий текущего кода CPL (Current Privilege Level). В процессорах Pentium поддерживается четыре уровня привилегий, от самого привилегированного нулевого, до наименее привилегированного третьего. С помощью привилегированных команд осуществляется защита структур операционной системы от некорректного поведения пользовательских процессов, а также взаимная защита ресурсов этих процессов. В процессоре Pentium к привилегированным командам относятся:

- команды для работы с управляющими регистрами CRn, а также для загрузки регистров системных адресов GDTR, LDTR, IDTR и TR:
- команда останова процессора HALT;
- команды запрета/разрешения маскируемых аппаратных прерываний CLI/SLI;
- команды ввода-вывода IN, INS, OUT, OUTS.

Первые две группы команд могут выполняться только при самом высшем уровне привилегий кода, то есть при CPL равном 0. Для двух последних групп команд, иногда называемых чувствительными, условия выполнения не требуют высшего уровня привилегий кода, а связаны с соотношением уровня привилегий ввода-вывода IOPL и уровня привилегий кода CPL – выполнение этих команд разрешено в том случае, если CPL меньше либо равно IOPL.

4.3 Сегментация с использованием страниц

Средства поддержки механизмов виртуальной памяти в процессоре Pentium позволяют отображать виртуальное адресное пространство на физическую память размером максимум в 4 Гбайт (этот максимум определяется использованием 32-разрядных адресов при работе с оперативной памятью). Хотя реально процессу пользователю может предоставляться меньше объёма в зависимости от операционной системы.

Процессор может поддерживать как сегментную модель распределения памяти, так и сегментно-страничную. Средства сегментации образуют верхний уровень средств управления виртуальной памятью процессора Pentium, а средства страничной организации – нижний уровень. Это означает, что сегментные средства работают всегда, а средства страничной организации могут быть как включены, так и выключены путем установки однобитного признака PE (Paging Enable) в регистре CRO процессора. В зависимости от того, включены ли средства страничной организации, изменяется смысл процедуры преобразования адресов, которая

выполняется средствами сегментации.

При работе процессора Pentium в сегментном режиме в распоряжении программиста имеется виртуальное адресное пространство, представляемое совокупностью сегментов. Каждый сегмент виртуальной памяти процесса имеет описание, называемое дескриптором сегмента. Дескриптор сегмента имеет размер 8 байт и содержит все характеристики сегмента, необходимые для проверки правильности доступа к нему и нахождения его в физическом адресном пространстве (Рисунок 33).

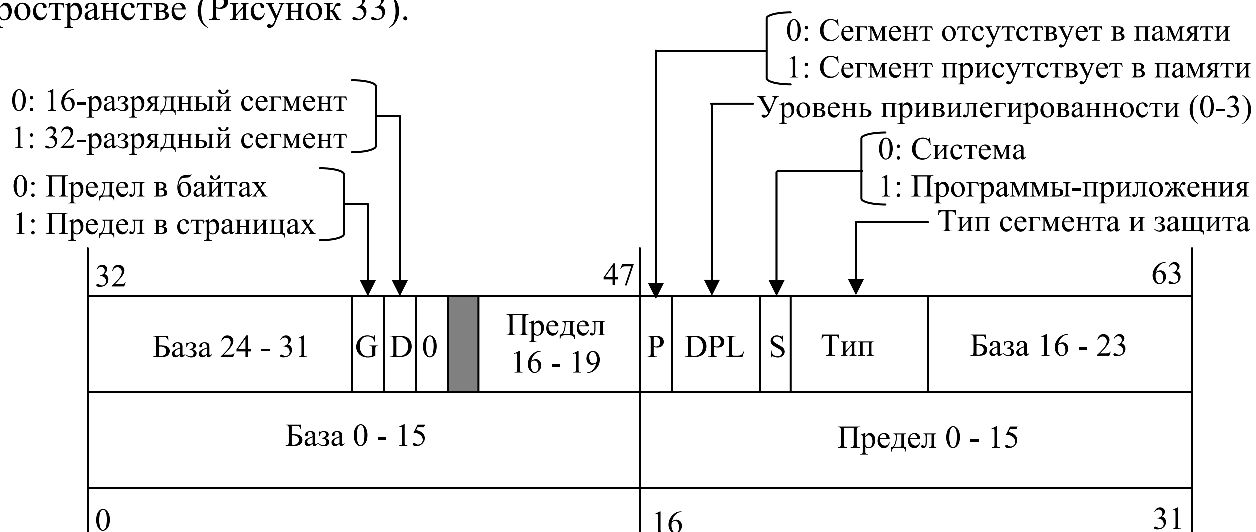

Рисунок 33 – Дескриптор программного сегмента в системе Pentium. Сегменты данных немного отличаются от программных сегментов

Структура дескриптора, которая поддерживается в процессоре Pentium, сложилась исторически. Многое в ней связано с обеспечением совместимости с предыдущими процессорами семейства x86. Именно этим объясняется то, что базовый адрес сегмента представлен в дескрипторе в виде трех частей, а предел сегмента занимает два поля. Ниже перечислены основные поля дескриптора.

База – базовый адрес сегмента (32 бита). Предел – размер сегмента (20 бит). G (Granularity) – единица измерения размера сегмента, один бит. Если G равен 1, то размер сегмента измеряется в страницах по 4 Кбайт.

Байт доступа (5-й байт дескриптора) содержит информацию, которая используется для принятия решения о возможности или невозможности обращения к данному сегменту. Бит P (Present) определяет, находится ли соответствующий сегмент в данный момент в памяти (P=1) или он выгружен на диск (P=0). Поле DPL (Descriptor Privilege Level) содержит данные об уровне привилегий, необходимом для доступа к сегменту. Остальные пять битов байта доступа зависят от типа сегмента и определяют способ, которым можно использовать данный сегмент (то есть читать, писать, выполнять).

Различаются три основных типа сегментов:

- сегмент данных;
- кодовый сегмент;
- системный сегмент (GDT, TSS и т. п.).

Дескрипторы сегментов объединяются в таблицы. Процессор Pentium для управления памятью поддерживает два типа таблиц дескрипторов сегментов:

- глобальная таблица дескрипторов (Global Descriptor Table, GDT), которая предназначена для описания сегментов операционной системы и общих сегментов для всех прикладных процессов, например сегментов межпроцессного взаимодействия;
- локальная таблица дескрипторов (Local Descriptor Table, LDT), которая содержит дескрипторы сегментов отдельного пользовательского процесса.

Таблица GDT одна, а таблиц LDT столько, сколько в системе выполняется задач (процессов). При этом в каждый момент времени операционной системой и аппаратными средствами процессора используется только одна из таблиц LDT, а именно та, которая соответствует выполняемому в данный момент пользовательскому процессу. Таблица GDT описывает общую часть виртуального адресного пространства процессов, а LDT – индивидуальную часть для каждого процесса. Таблицы GDT и LDT размещены в оперативной памяти в виде отдельных сегментов. Сегменты LDT и GDT содержат системные данные, поэтому их дескрипторы хранятся в таблице GDT. Таким образом, таблица GDT наряду с записями о других сегментах содержит запись о самой себе, а также обо всех таблицах LDT.

В каждый момент времени в специальных регистрах GDTR и LDTR хранится информация о местоположении и размерах глобальной таблицы GDT и активной таблицы LDT соответственно. Регистр GDTR содержит 32-разрядный физический адрес начала сегмента GDT в памяти, а также 16-битный размер этого сегмента. Регистр LDTR указывает на расположение сегмента LDT в оперативной памяти косвенно – он содержит индекс дескриптора в таблице GDT, в котором содержится адрес таблицы LDT и ее размер.

Процесс обращается к физической памяти по виртуальному адресу, представляющему собой пару (селектор, смещение). Селектор однозначно определяет виртуальный сегмент, к которому относится искомый адрес, то есть он может интерпретироваться как номер сегмента, а смещение, как это и следует из его названия, фиксирует положение искомого адреса относительно начала сегмента. Смещение задается в машинной инструкции, а селектор помещается в один из сегментных регистров процессора. Под смещение отводится 32 бита, что обеспечивает максимальный размер сегмента 4 Гбайт.

Чтобы получить доступ к сегменту, программа системы Pentium сначала загружает селектор для этого сегмента в один из шести сегментных регистров машины. Во время выполнения регистр CS содержит селектор для сегмента кода команд, а регистр DS хранит селектор для сегмента данных. Каждый селектор представляет собой 16-разрядный номер (Рисунок 34).

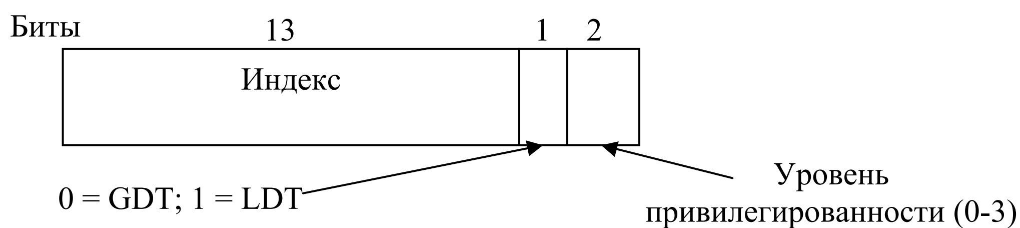

Рисунок 34 – Селектор в системе Pentium

Один из битов селектора несет информацию о том, является ли данный сегмент локальным или глобальным (то есть находится ли он в локальной или глобальной таблице дескрипторов). Следующие тринадцать битов определяют номер записи в таблице дескрипторов, поэтому эти таблицы ограничены: каждая содержит 8 Кбайт сегментных дескрипторов. Дескриптор 0 является запрещенным. Его можно безопасно загрузить в сегментный регистр, чтобы обозначить, что сегментный регистр в данный момент недоступен. При попытке его использовать происходит прерывание.

Виртуальное адресное пространство процесса складывается из всех сегментов, описанных в общей для всех процессов таблице GDT, и сегментов, описанных в его собственной таблице LDT. Разрядность поля индекса определяет максимальное число глобальных и локальных сегментов процесса – по 8 Кбайт сегментов каждого типа, всего 16 Кбайт сегментов. С учетом максимального размера сегмента – 4 Гбайт – каждый процесс при чисто сегментной организации виртуальной памяти (без включения страничного механизма) может работать в виртуальном адресном пространстве в 64 Тбайт.

Каждый дескриптор в таблицах GDT и LDT имеет размер 8 байт, поэтому максимальный размер каждой из этих таблиц – 64 Кбайт (8 байт на 8 Кбайт дескрипторов).

Теперь проследим шаги, с помощью которых пара (селектор, смещение) преобразуется в физический адрес. Как только микропрограмма узнает, какой сегментный регистр используется, она может найти в своих внутренних регистрах полный дескриптор, соответствующий этому селектору. Если сегмент не существует (селектор равен 0) или в данный момент выгружен, возникает прерывание.

Затем она проверяет, выходит ли смещение за пределы сегмента, в случае чего также возникает прерывание. Логически в дескрипторе просто должно существовать 32-разрядное поле, дающее размер сегмента, но там доступны только 20 бит, поэтому используется другая схема. Если поле Gbit равно 0, поле Предел содержит точный размер сегмента, до 1 Мбайт. Если оно равно 1, поле Предел даёт размер сегмента в страницах вместо байтов. Размер страницы в системе Pentium фиксирован на величине 4 Кбайт, поэтому 20 битов достаточно для сегментов размером до 232 байтов.

Если сегмент находится в памяти, смещение попало в нужный интервал, тогда система Pentium прибавляет 32-разрядное поле База в дескрипторе к смещению, формируя то, что называется линейным адресом, как показано на рисунке 35. Поле база разбито на три части, которые разбросаны по дескриптору для совместимости с процессором Intel 80286, в котором поле База имеет только 24 бита. В сущности, поле База позволяет каждому сегменту начинаться в произвольном месте внутри 32-разрядного линейного адресного пространства.

Если разбиение на страницы блокировано (с помощью бита в глобальном управляющем регистре), линейный адрес интерпретируется как физический адрес и посылается в память для чтения или записи. При отключенной страничной схеме памяти получается чистая схема сегментации с базовым адресом каждого сегмента, выдаваемым его дескриптором. Сегментам разрешено перекрываться случайным

образом потому, что контроль за тем, чтобы они не пересекались, мог бы занять много времени.

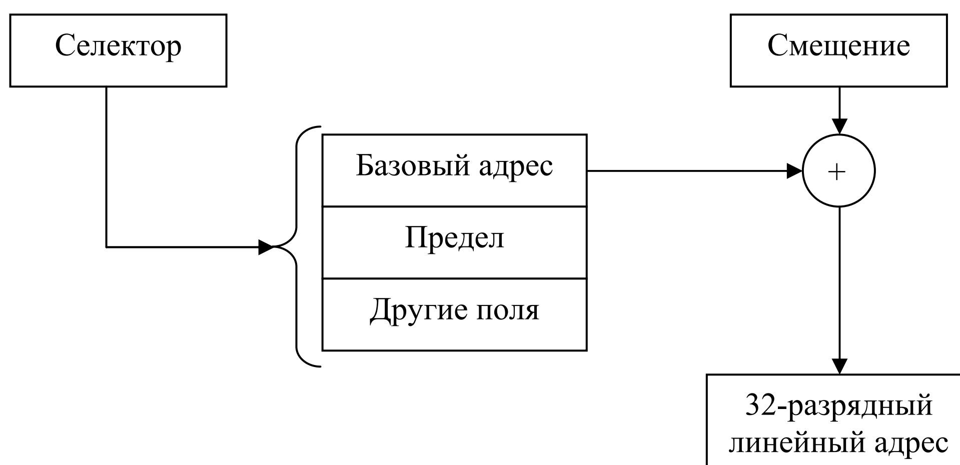

Рисунок 35 – Преобразование пары (селектор, смещение) в физический адрес

Если доступна страничная подкачка, линейный адрес интерпретируется как виртуальный адрес и отображается на физический адрес с помощью таблицы страниц практически так же, как в наших предыдущих примерах. Единственная серьезная трудность заключается в том, что при 32-разрядном виртуальном адресе и странице размером 4 Кбайт сегмент может содержать 1 млн. страниц, поэтому используется двухуровневое отображение с целью уменьшения размера таблицы страниц для маленьких сегментов.

У каждой работающей программы есть страничный каталог, состоящий из 1024 32-разрядных записей. Он расположен по адресу, хранящемуся в глобальном регистре. Каждая запись в каталоге ссылается на таблицу страниц, также содержащую 1024 32-разрядных записей. Записи в таблицах страниц в свою очередь указывают на страничные блоки. Эта схема продемонстрирована на рисунке 36.

Поле Каталог используется как индекс в страничном каталоге, определяющий расположение указателя на правильную таблицу страниц. Затем поле Страница используется в качестве индекса в таблице страниц, чтобы найти физический адрес страничного блока. Чтобы получить физический адрес требуемого байта или слова, к адресу страничного блока прибавляется последнее поле Смещение.

Каждая запись в таблице имеет размер 32 бита, 20 из которых содержат номер страничного блока. Каждая таблица страниц включает в себя записи для 1024 страничных блоков размером по 4 Кбайт, таким образом, одна таблица страниц управляет четырьмя мегабайтами памяти. Сегмент, длина которого меньше 4 Мбайт, будет иметь страничный каталог с единственной записью - указателем на его единственную таблицу страниц. Следовательно, в случае короткого сегмента на поддержку таблиц страниц расходуется только две страницы вместо миллиона, который был бы нужен в одноуровневой таблице страниц.

Чтобы избежать создания повторных обращений к памяти, система Pentium имеет небольшой буфер быстрого преобразования адреса (TLB), который напрямую отображает наиболее часто использующиеся комбинации Каталог-Страница на физический адрес страничного блока. Только когда текущая комбинация

отсутствует в буфере TLB, действительно выполняется механизм, показанный на рисунке 36, и буфер TLB обновляется. Система обладает хорошей производительностью до тех пор, пока обращения к отсутствующим страницам в буфере TLB происходят относительно редко.

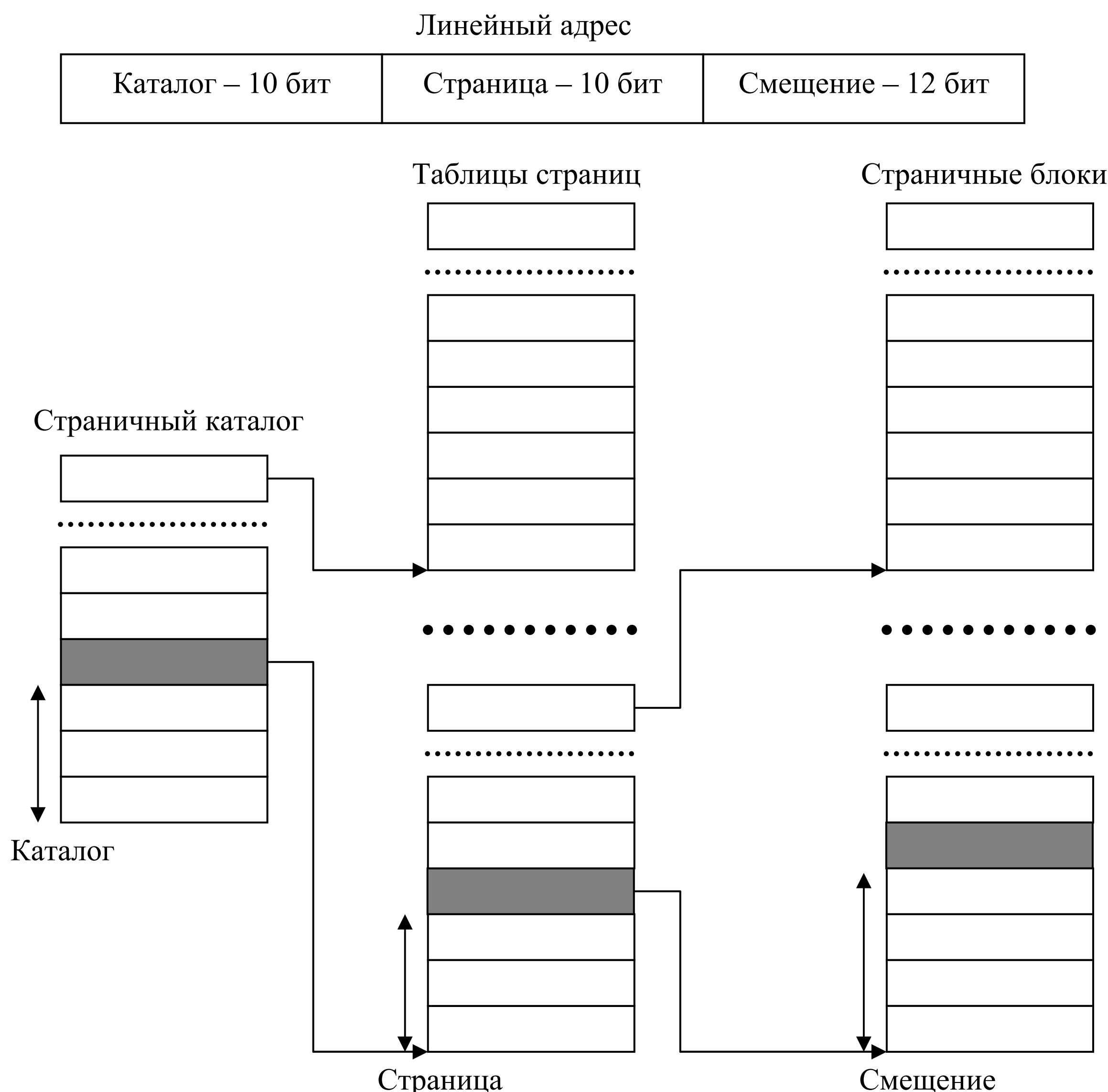

Рисунок 36 – Отображение линейного адреса на физический адрес

Из приведенного описания видно, что процессор Pentium обеспечивает поддержку работы операционной системы в двух отношениях:

- поддерживает работу виртуальной памяти за счет быстрого аппаратного способа преобразования виртуального адреса в физический;
- обеспечивает защиту данных и кодов различных приложений.

4.4 Защита данных в процессоре Pentium

Процессор Pentium при работе в сегментном режиме предоставляет операционной системе следующие средства, направленные на обеспечение защиты процессов друг от друга [11].

- Процессор Pentium поддерживает для каждого процесса отдельную таблицу дескрипторов сегментов LDT. Эта таблица формируется операционной системой на этапе создания процесса. После активизации процесса в регистр LDTR заносится адрес (селектор) его таблицы LDT. Тем самым система делает недоступным для процесса локальные сегменты других процессов, описанные в их таблицах LDT.
- Вместе с тем таблица GDT, в которой хранятся дескрипторы сегментов операционной системы, а также сегменты, используемые несколькими процессами совместно, доступна всем процессам, а следовательно, не защищена от их несанкционированного вмешательства. Для решения этой проблемы в процессоре Pentium предусмотрена поддержка системы безопасности на основе привилегий. Каждый сегмент памяти наделяется атрибутами безопасности, которые характеризуют степень защищенности данного сегмента. Процессы также наделяются атрибутами безопасности, которые в этом случае характеризуют степень привилегированности процесса. Уровень привилегий процесса определяется уровнем привилегий его кодового сегмента. Доступ к сегменту регулируется в зависимости от его защищенности и уровня привилегированности запроса. Это позволяет защитить от несанкционированного доступа сегменты GDT, а также обеспечить дифференцированный доступ к локальным сегментам процесса.
- Само по себе наличие отдельных таблиц дескрипторов для каждого процесса еще не обеспечивает надежной защиты процессов друг от друга. Поэтому необходимым элементом защиты является наличие аппаратных ограничений в наборе инструкций, разрешенных для выполнения процессу. Некоторые инструкции, имеющие критически важное значение для функционирования системы, такие, например, как останов процессора, загрузка регистра GDTR, загрузка регистра LDTR, являются привилегированными как раз по этой причине – процесс может выполнять их, только обладая наивысшим уровнем привилегий.
- Еще одним механизмом защиты, поддерживаемым в процессоре Pentium, является ограничение на способ использования сегмента. В зависимости от того, к какому типу относится сегмент – сегмент данных, кодовый сегмент или системный сегмент – некоторые действия по отношению к нему могут быть запрещены. Например, в кодовый сегмент нельзя записывать какие-либо данные, а сегменту данных нельзя передать управление, даже если загрузить его селектор в регистр CS.

В процессоре Pentium используется мандатный способ определения прав доступа к сегментам памяти, при котором имеется несколько уровней привилегий и объекты каждого уровня имеют доступ ко всем объектам равного уровня или более низких уровней, но не имеют доступа к объектам более высоких уровней. В процессоре Pentium существует четыре уровня привилегий – от нулевого, который является самым высоким, до третьего – самого низкого. Предполагается, что нулевой уровень доступен для ядра операционной системы, третий уровень – для прикладных программ, а промежуточные уровни – для утилит и подсистем операционной системы, менее привилегированных, чем ядро.

Система защиты манипулирует несколькими переменными, характеризующими уровень привилегий:

- DPL (Descriptor Privilege Level) – уровень привилегий дескриптора, задается полем DPL в дескрипторе сегмента;
- RPL (Requested Privilege Level) – запрашиваемый уровень привилегий, задается полем RPL селектора сегмента;
- CPL (Current Privilege Level) – текущий уровень привилегий выполняемого кода, задается полем RPL селектора кодового сегмента;
- EPL (Effective Privilege Level) – эффективный уровень привилегий запроса.

Под запросом здесь понимается любое обращение к памяти независимо от того, произошло ли оно при выполнении кода, оформленного в виде процесса, или вне рамок процесса, как это бывает при выполнении кодов операционной системы.

Уровни привилегий назначаются дескрипторам и селекторам. Во время загрузки операционной системы в память, а также при создании новых процессов операционная система назначает процессу сегменты кода и данных, генерирует дескрипторы этих сегментов и помещает их в таблицы GDT или LDT Конкретные значения уровней привилегий DPL и RPL задаются операционной системой и транслятором либо по умолчанию, либо на основании указаний программиста. Значения DPL и RPL определяют возможности создаваемого процесса

Уровень привилегий дескриптора DPL является в некотором смысле первичной характеристикой, которая «переносится» на соответствующие сегменты и запросы. Сегмент обладает тем уровнем привилегий, который записан в поле DPL его дескриптора. DPL определяет степень защищенности сегмента. Уровень привилегий сегмента данных учитывается системой защиты, когда она принимает решения о возможности выполнения для этого сегмента чтения или записи. Уровень привилегий кодового сегмента используется системой защиты при проверке возможности чтения или выполнения кода для данного сегмента.

Уровень привилегий кодового сегмента определяет не только степень защищенности этого сегмента, но и текущий уровень привилегий CPL всех запросов к памяти (на чтение, запись или выполнение), которые возникнут при выполнении этого кодового сегмента. Уровень привилегий кодового сегмента DPL характеризует текущий уровень привилегий CPL выполняемого кода. При запуске кода на выполнение значение DPL из дескриптора копируется в поле RPL селектора кодового сегмента, загружаемого в регистр сегмента команд CS. Значение поля RPL кодового сегмента и является текущим уровнем привилегий выполняемого кода, то есть уровнем CPL.

От того, какой уровень привилегий имеет выполняемый код, зависят не только возможности его доступа к сегментам и дескрипторам, но и разрешенный ему набор инструкций.

Во время приостановки процесса его текущий уровень привилегий сохраняется в контексте, роль которого в процессоре Pentium играет системный сегмент состояния задачи (Task State Segment, TSS). Если какой-либо процесс имеет несколько кодовых сегментов с разными уровнями привилегий, то поле RPL регистра CS позволяет узнать значение текущего уровня привилегий процесса. Пользовательский процесс не может изменить значение поля привилегий в дескрипторе, так как необходимые для этого инструкции ему недоступны. В

дальнейшем уровень привилегий процесса может измениться только в случае передачи управления другому кодовому сегменту путем использования особого дескриптора – шлюза.

Контроль доступа процесса к сегментам данных осуществляется на основе сопоставления эффективного уровня привилегий EPL запроса и уровня привилегий DPL дескриптора сегмента данных. Эффективный уровень привилегий учитывает не только значение CPL, но и значение запрашиваемого уровня привилегий для конкретного сегмента, к которому выполняется обращение. Перед тем как обратиться к сегменту данных, выполняемый код загружает селектор, указывающий на этот сегмент, в один из регистров сегментов данных: DS, ES, FS или GS. Значение поля RPL данного селектора задает уровень запрашиваемых привилегий. Эффективный уровень привилегий выполняемого кода EPL определяется как максимальное (то есть худшее) из значений текущего и запрашиваемого уровней привилегий:

$$EPL = \max\{CPL, RPL\}. \quad (4)$$

Выполняемый код может получить доступ к сегменту данных для операций чтения или записи, если его эффективный уровень привилегий не ниже (а в арифметическом смысле «не больше») уровня привилегий дескриптора этого сегмента:

$$\max\{CPL, RPL\} \le DPL. \quad (5)$$

Уровень привилегий дескриптора DPL определяет степень защищенности сегмента. Значение DPL говорит о том, каким эффективным уровнем привилегий должен обладать запрос, чтобы получить доступ к данному сегменту. Например, если дескриптор имеет DPL=2, то к нему разрешено обращаться всем процессам, имеющим уровень привилегий EPL, равный 0, 1 или 2, а для процессов с EPL, равным 3, доступ запрещен.

Контроль доступа процесса к сегменту стека позволяет предотвратить доступ низкоуровневого кода к данным, выработанным высокоуровневым кодом, и помещенным в стек, например к локальным переменным процедуры. Доступ к сегменту стека разрешается только в том случае, когда уровень EPL кода совпадает с уровнем DPL сегмента стека, то есть коду разрешается работать только со стеком своего уровня привилегий. Использование одного и того же стека для процедур разного уровня привилегий может привести к тому, что низкоуровневая процедура, получив управление после возврата из вызванной ею высокоуровневой процедуры, может прочитать из стекового сегмента записываемые туда во время работы высокоуровневой процедуры данные. Так как в ходе выполнения процесса уровень привилегий его кода может измениться, то для каждого уровня привилегий используется отдельный сегмент стека.

Контроль доступа процесса к кодовому сегменту производится путем сопоставления уровня привилегий дескриптора этого кодового сегмента DPL с текущим уровнем привилегий выполняемого кода CPL. В зависимости от того, какой способ обращения к кодовому сегменту используется, выполняется внутрисегментная или межсегментная передача управления, вызывается подчиненный или неподчиненный сегмент – по-разному формулируются правила контроля доступа.

Осуществляя контроль доступа к сегменту, аппаратура процессора учитывает не только уровень привилегий, но и «легитимность» способа использования данного сегмента.

4.5 Средства вызова процедур и задач

Операционная система, как однозадачная, так и многозадачная, должна предоставлять задачам средства вызова процедур операционной системы, библиотечных процедур. Она должна также иметь средства для запуска задач, а при многозадачной работе – средства быстрого переключения с задачи на задачу. Вызов процедуры отличается от запуска задачи тем, что в первом случае виртуальное адресное пространство задачи остается тем же (таблица LDT остается прежней), а при вызове задачи это адресное пространство полностью меняется.

Вызов процедуры без смены кодового сегмента в защищенном режиме процессора Pentium производится обычным образом с помощью команд JMP и CALL.

Для вызова процедуры, код которой находится в другом сегменте (этот сегмент может принадлежать другому программному модулю приложения, библиотеке, другой задаче или операционной системе), процессор Pentium предоставляет несколько способов вызова, причем во всех используется защита, основанная на уровнях привилегий.

Прямой вызов процедуры из неподчиненного и подчинённого сегментов. Этот способ состоит в непосредственном указании в поле команды JMP или CALL селектора, который указывает на дескриптор нового кодового сегмента. Этот сегмент содержит код вызываемой процедуры. Базовый адрес сегмента, содержащийся в дескрипторе, и смещение, задаваемое в команде JMP или CALL, определяют начальный адрес вызываемой процедуры. Схема такого вызова приведена на рисунке 37.

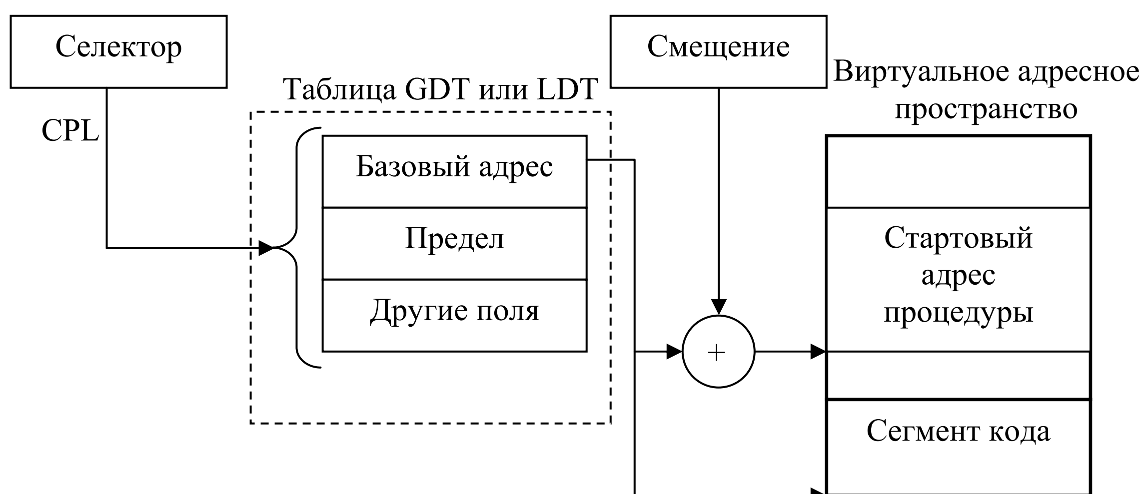

Рисунок 37 – Непосредственный вызов процедуры

Разрешение вызова происходит в зависимости от значения поля ***C*** в дескрипторе сегмента вызываемого кода.

При ***C*** равном 0 вызываемый сегмент не считается подчиненным, и вызов разрешается, только если уровень привилегий вызывающего кода совпадает с уровнем привилегий вызываемого сегмента (CPL=DPL). При ***C*** равном 1 сегмент считается подчинённым. Подчиненный сегмент можно вызывать с помощью указания его селектора в командах CALL или JMP из кода программ с равным или более низким уровнем привилегий (CPL≥DPL).

Но нужно иметь в виду, что вызываемый код будет в этом случае выполняться с привилегиями вызывающей программы. Например, если код операционной системы, хранящийся в сегменте с уровнем привилегий 0, будет вызван из пользовательского приложения с уровнем привилегий 3, то процедура операционной системы будет наследовать привилегии пользовательской программы и возможности этой процедуры по доступу к системным данным будут весьма ограничены. Тем не менее выполнить действия над пользовательскими данными вызванная таким способом процедура операционной системы сможет.

Косвенный вызов процедуры через шлюз. Очевидно, что оба рассмотренных выше способа вызова процедур не подходят для реализации системных вызовов. Первый способ в принципе не позволяет вызвать из пользовательской программы с третьим уровнем привилегий процедуру операционной системы, находящуюся в неподчиненном сегменте и имеющую более высокий уровень привилегий. С помощью второго способа могут быть вызваны процедуры операционной системы, находящиеся в подчиненном сегменте, однако они будут выполняться с пользовательским уровнем привилегий и не смогут обрабатывать системные данные, что нужно для большинства системных вызовов. Поэтому процессор Pentium предоставляет еще один способ вызова подпрограмм – через шлюз (вентиль), позволяющий пользовательскому коду вызывать привилегированные процедуры, которые будут работать со своим высоким уровнем привилегий. Шлюзы вызова обладают еще одним преимуществом – появляется возможность контроля точек входа в вызываемые процедуры. В обоих рассмотренных выше способах адрес точки входа в вызываемую процедуру определяется смещением, заданным в команде CALL вызывающей процедуры, то есть существует возможность задания некорректного значения смещения, в результате чего может произойти передача управления не на нужную команду или вообще в середину команды. Шлюзы вызова свободны от данного недостатка.

Набор точек входа в привилегированные кодовые сегменты определяется заранее, и эти точки входа описываются с помощью специальных дескрипторов – дескрипторов шлюзов вызова процедур. Дескрипторы этого типа принадлежат к системным дескрипторам, и хотя их структура отличается от структуры дескрипторов сегментов кода и данных, они также включены в таблицы LDT и GDT.

Схема вызова процедуры через шлюз приведена на рисунке 38. Селектор из поля команды CALL указывает на дескриптор шлюза в таблицах GDT или LDT. Для того чтобы получить доступ к процедуре через шлюз, описываемый данным дескриптором, вызывающий код должен иметь не меньший уровень прав, чем дескриптор шлюза. При этом вызываемый код может иметь любой уровень привилегий (в том числе и более высокий, чем у шлюза), который сохраняется при его выполнении. Это позволяет из пользовательской программы вызывать

процедуры операционной системы, работающие с высоким уровнем привилегий. При определении адреса входа в вызываемом сегменте смещение из поля команды CALL не используется, а используется смещение из дескриптора шлюза, что не дает возможности задаче самой определять точку входа в защищенный кодовый сегмент.

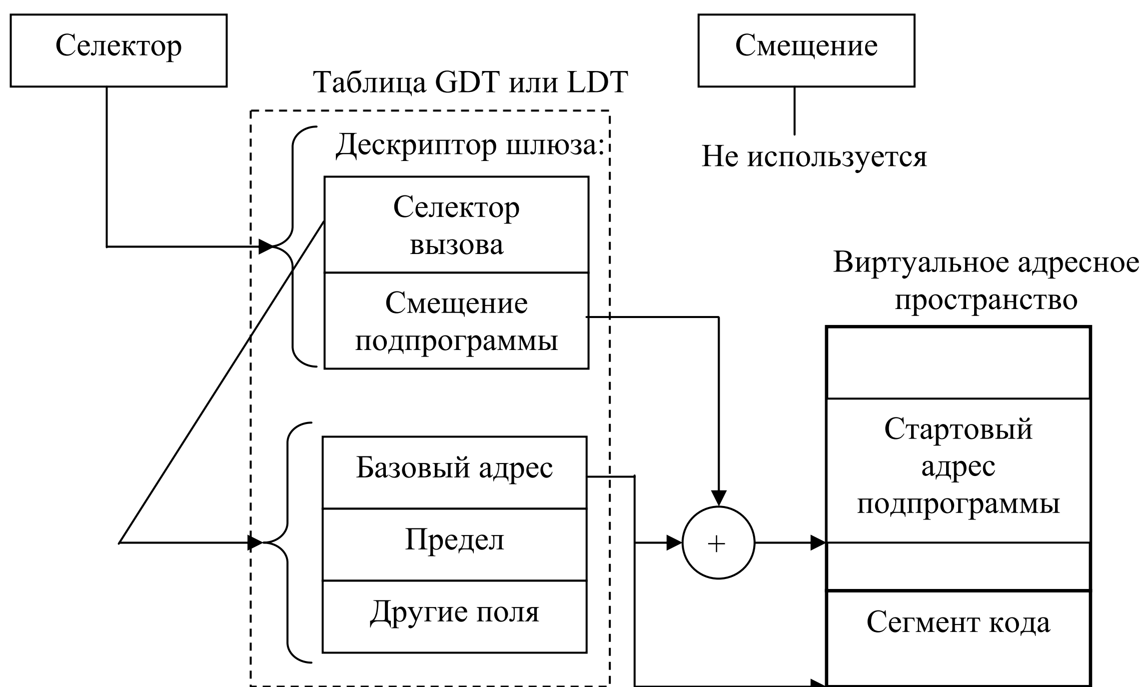

Рисунок 38 – Вызов подпрограммы через шлюз вызова

При вызове кодов, обладающих различными уровнями привилегий, возникает проблема передачи параметров между вызывающей и вызываемой процедурами. Для ее решения в процессоре предусмотрено существование стеков разных уровней, по одному стеку на каждый уровень привилегий. Используемый кодовым сегментом стек всегда соответствует текущему уровню привилегий кодового сегмента, то есть значению CPL. В сегменте контекста задачи TSS хранятся значения селекторов стека SS для уровней привилегий 0, 1 и 2. Если вызывается процедура, имеющая уровень привилегий, отличный от текущего, то при выполнении команды CALL создается новый стек. Для этого из сегмента TSS извлекается новое значение селектора стека, соответствующее новому уровню привилегий, которое загружается в регистр SS, из текущего стека в новый стек копируется столько 32-разрядных слов, сколько указано в поле счетчика слов дескриптора шлюза. В новом стеке также запоминается селектор старого стека, который используется при возврате в вызывающую процедуру.

Вызов задачи. Механизм вызова при переключении между задачами отличается от механизма вызова процедур. В этом случае селектор команды CALL должен указывать на дескриптор системного сегмента TSS. Сегмент TSS хранит контекст задачи, то есть информацию, которая нужна для восстановления выполнения прерванной в произвольный момент времени задачи. Контекст задачи включает значения регистров процессора, указатели на открытые файлы и некоторые другие, зависящие от операционной системы, переменные. Скорость переключения контекста в значительной степени влияет на производительность

многозадачной операционной системы.

Процессор Pentium производит аппаратное переключение контекстов задач, используя для этого сегменты специального типа TSS. Как и в случае вызова процедуры, имеются два способа вызова задачи – непосредственный вызов путем указания селектора дескриптора сегмента TSS нужной задачи в поле команды CALL и косвенный вызов через шлюз вызова задачи.

Однако условие, разрешающее непосредственный вызов задачи, отличается от условия непосредственного вызова процедуры: вызов возможен только в случае, если вызывающий код обладает уровнем привилегий, не меньшим, чем вызываемая задача.

При вызове через шлюз (который может располагаться и в таблице LDT) вызывающему коду достаточно иметь права доступа к шлюзу, а шлюз может указывать на дескриптор TSS в таблице GDT с равным или более высоким уровнем привилегий. Поэтому через шлюз вызова задачи можно выполнить переключение на более привилегированную задачу.

Непосредственный вызов задачи показан на рисунке 39.

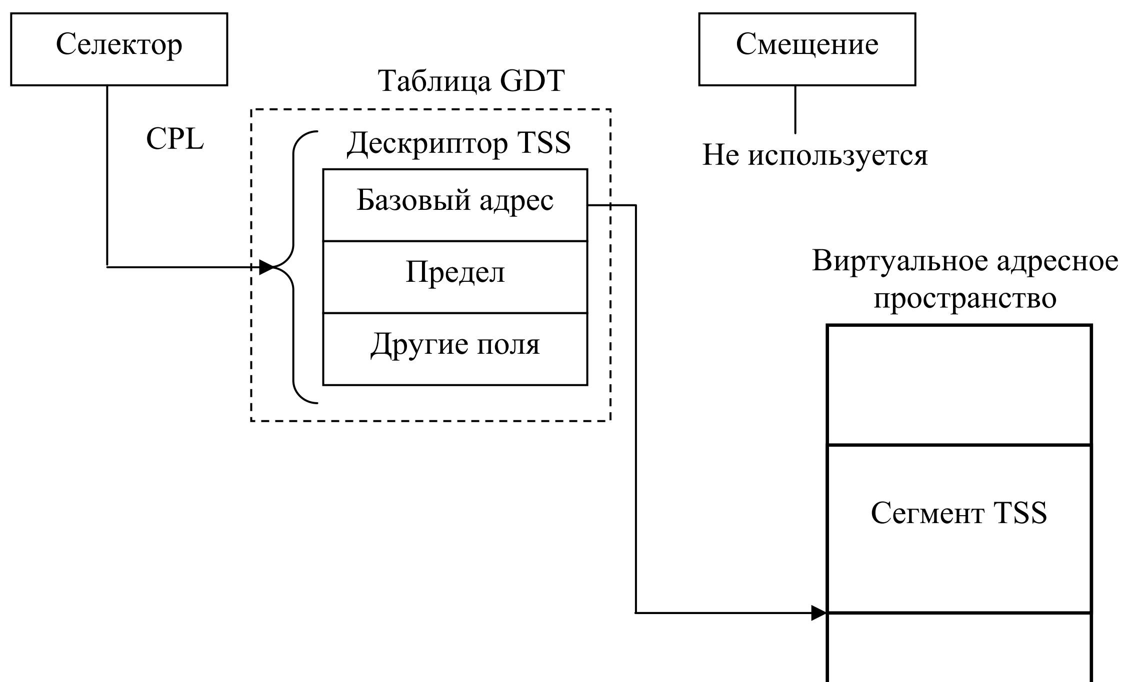

Рисунок 39 – Непосредственный вызов задачи

При переключении задач процессор выполняет следующие действия.

- Выполняется команда CALL, селектор которой указывает на дескриптор сегмента типа TSS. Происходит проверка прав доступа, успешная при CPL≤DPL.
- В TSS текущей задачи сохраняются значения регистров процессора. На текущий сегмент TSS указывает регистр процессора TR, содержащий селектор сегмента.
- В TR загружается селектор сегмента TSS задачи, на которую переключается процессор.
- Из нового TSS в регистр LDTR переносится значение селектора таблицы

LDT в таблице GDT задачи.

- Восстанавливаются значения регистров процессора (из соответствующих полей нового сегмента TSS).
- В поле селектора возврата нового сегмента TSS заносится селектор сегмента TSS снимаемой с выполнения задачи для организации возврата к ней в будущем.

Вызов задачи через шлюз происходит аналогично, добавляется только этап поиска дескриптора сегмента TSS по значению селектора дескриптора шлюза вызова.

4.6 Механизм прерываний

Процессор Pentium поддерживает векторную схему прерываний, с помощью которой может быть вызвано 256 процедур обработки прерываний (вектор имеет длину в один байт). Соответственно таблица процедур обработки прерываний имеет 256 элементов, которые в реальном режиме работы процессора состоят из дальних адресов (CS:IP) этих процедур, а в защищенном режиме – из дескрипторов. Контроллер прерываний в большинстве аппаратных платформ на основе процессоров Pentium реализует механизм опрашиваемых прерываний, поэтому общий механизм компьютера носит смешанный векторно-опрашиваемый характер. Прерывания, которые обрабатывает Pentium, делятся на следующие классы [11]:

- аппаратные (внешние) прерывания – источником таких прерываний является сигнал на входе процессора;
- исключения – внутренние прерывания процессора;
- программные прерывания, происходящие по команде INT.

Аппаратные прерывания бывают маскируемыми и немаскируемыми. Маскируемые прерывания вызываются сигналом INTR на одном из входов микросхемы процессора. При его возникновении процессор завершает выполнение очередной инструкции, сохраняет в стеке значение регистра признаков программы EFLAGS и адреса возврата, а затем считывает с входов шины данных байт вектора прерываний и в соответствии с его значением передает управление одной из 256 процедур обработки прерываний.

Маскируемость прерываний управляется флагом разрешения прерываний, IF (Interrupt Flag), находящимся в регистре EFLAGS процессора. При IF=1 маскируемые прерывания разрешены, а при IF=0 – запрещены. Для явного управления флагом IF в процессоре имеются чувствительные к уровню привилегий инструкции разрешения маскируемых прерываний STI (SeT Interrupt flag) и запрета маскируемых прерываний CLI (Clear Interrupt flag).

Немаскируемое аппаратное прерывание происходит при появлении сигнала NMI (Non Maskable Interrupt) на входе процессора. Этот сигнал всегда прерывает работу процессора, вне зависимости от значения флага IF. При обработке немаскируемого прерывания вектор не считывается, а управление всегда передается процедуре с номером 2, описываемой третьим элементом таблицы процедур обработки прерываний (нумерация в этой таблице начинается с нуля).

Немаскируемые прерывания предназначаются для реакции на «сверхважные» для компьютерной системы события, например сбой по питанию. В ходе процедуры обслуживания немаскируемого прерывания процессор не реагирует на другие запросы немаскируемых и маскируемых прерываний до тех пор, пока не будет выполнена команда IRET. Если при обработке немаскируемого прерывания возникает новый сигнал NMI, то он фиксируется и обрабатывается после завершения обработки текущего прерывания, то есть после выполнения команды IRET.

Исключения (exeprtions) делятся в процессоре Pentium на отказы (faults), ловушки (traps) и аварийные завершения (aborts).

Отказы соответствуют некорректным ситуациям, которые выявляются до выполнения инструкции, например, при обращении по адресу, находящемуся в отсутствующей в оперативной памяти странице (страничный отказ). После обработки исключения-отказа процессор повторяет выполнения команды, которую он не смог выполнить из-за отказа.

Ловушки обрабатываются процессором после выполнения инструкции, например при возникновении переполнения. После обработки процессор выполняет инструкцию, следующую за той, которая вызвала исключение.

Аварийные завершения соответствуют ситуациям, когда невозможно точно определить команду, вызвавшую прерывание. Чаще всего это происходит во время серьезных отказов, связанных со сбоями в работе аппаратуры компьютера. Для обработки исключений в таблице прерываний отводятся номера 0-31.

Программные прерывания в процессоре Pentium происходят при выполнении инструкции INT с однобайтовым аргументом, в котором указывается вектор прерывания. Общая длина инструкции INT – два байта, исключение составляет инструкция INT 3, которая целиком помещается в один байт – это удобно при отладке программ, когда инструкция INT заменяет первый байт любой команды, вызывая переход на процедуру отладки. Программные прерывания подобно ловушкам обрабатываются после выполнения соответствующей инструкции INT, а возврат происходит в следующую инструкцию. Программное прерывание может вызвать любую из 256 процедур обработки прерываний, указанных в таблице прерываний.

При одновременном возникновении запросов прерываний различных типов процессор Pentium разрешает противоречие с помощью приоритетов. Немаскируемые прерывания имеют более высокий приоритет, чем маскируемые. Приоритетность внутри маскируемых прерываний устанавливается не процессором, а контроллером прерываний (процессор не может этого сделать, так как для него все маскируемые запросы представлены одним сигналом INTR). Проверка некорректных ситуаций, порождающих исключения (в том числе и при выполнении одной команды), выполняется в процессоре в соответствии с определенной последовательностью.

Таблица прерываний в реальном режиме состоит из 256 элементов, каждый из которых имеет длину в 4 байта и представляет собой дальний адрес (CS:IP) процедуры обработки прерываний. Таблица прерываний реального режима всегда находится в фиксированном месте физической памяти – с начального адреса 00000

по адрес 003FF.

В защищенном режиме таблица прерываний носит название IDT (Interrupt Descriptor Table) и может располагаться в любом месте физической памяти. Ее начало (32-разрядный физический адрес) и размер (16 бит) можно найти в регистре системных адресов IDTR. Каждый из 256 элементов таблицы прерываний представляет собой 8-байтный дескриптор. В таблице прерываний могут находиться только дескрипторы определенного типа – дескрипторы шлюзов прерываний, шлюзов ловушек и шлюзов задач.

Шлюзы прерываний и ловушек специально вводятся для вызова процедур обработки прерываний. Если для вызова процедуры обработки прерывания используется шлюз задач, то происходит смена процесса, а по завершении обработки – возврат к прерванному процессу. Шлюзы прерываний и ловушек не вызывают смены контекста задачи, следовательно, процедуры обработки прерываний в этом случае вызываются быстрее, чем при использовании шлюза задачи.

4.7 Кэширование в процессоре Pentium

В процессоре Pentium кэширование используется в следующих случаях [11].

- *Кэширование дескрипторов сегментов в скрытых регистрах.* Для каждого сегментного регистра в процессоре имеется так называемый скрытый регистр дескриптора. В скрытый регистр при загрузке сегментного регистра помещается информация из дескриптора, на который указывает данный сегментный регистр. Информация из дескриптора сегмента используется для преобразования виртуального адреса в физический при чисто сегментной организации памяти либо для получения линейного виртуального адреса при страничном механизме. Доступ к скрытому регистру выполняется быстрее, чем поиск и извлечение информации из таблицы страниц, находящейся в оперативной памяти. Поэтому если очередное обращение будет относиться к одному из сегментов, дескриптор которого еще хранится в скрытом регистре (а вероятность этого велика), то преобразование адресов будет выполнено быстрее. Тем самым скрытые регистры играют роль кэша таблицы дескрипторов и ускоряют работу процессора.
- *Кэширование пар номеров виртуальных и физических страниц в буфере ассоциативной трансляции TLB (Translation Lookaside Buffer)* позволяет ускорять преобразование виртуальных адресов в физические при сегментно-страничной организации памяти. TLB представляет собой ассоциативную память небольшого объема, предназначенную для хранения интенсивно используемых дескрипторов страниц. В процессоре Pentium имеются отдельные TLB для инструкций и данных.
- *Кэширование данных и инструкций в кэш-памяти первого уровня.* Эта память, называемая также *внутренней* кэш-памятью, поскольку она размещена непосредственно на кристалле микропроцессора, имеет объем 16/32 Кбайт. В процессоре Pentium кэш первого уровня разделен на память для хранения данных и память для хранения инструкций. Согласование данных выполняется только методом сквозной записи.
- *Кэширование данных и инструкций в кэш-памяти второго уровня.* Эта

память называется также *внешней* кэш-памятью, поскольку она устанавливается в виде отдельной микросхемы на системной плате. Кэш-память второго уровня является общей для данных и инструкций и имеет объем 256/512 Кбайт. Поиск в кэше второго уровня выполняется в случае, когда констатируется промах в кэше первого уровня. Для согласования данных в кэше второго уровня может использоваться как сквозная, так и обратная запись.

Контрольные вопросы по разделу

1 Значения каких системных регистров процессора должен использовать программный модуль операционной системы, чтобы произвести обращение к индивидуальной части памяти текущего процесса?

2 Представьте, что для задач всех уровней привилегий используется один общий стек. К каким последствиям это может привести?

3 Почему в сегменте состояния задачи TSS хранятся значения селекторов стека для уровней привилегий 0, 1 и 2, но нет значения для селектора уровня 3?

4 В какой памяти – физической или виртуальной – задает положение сегмента при выключенном страничном механизме базовый адрес, хранимый в дескрипторе сегмента?

5 Зачем нужны шлюзы вызовов процедур и задач, если существует возможность непосредственного вызова?

6 Чем отличаются маскируемые и немаскируемые прерывания?

7 Перечислите переменные, характеризующие уровни привилегий в процессоре Pentium.

8 Зачем в процессоре Pentium используется двухуровневая структура страниц?

5 Ввод-вывод

5.1 Принципы аппаратуры ввода-вывода

Одной из главных задач операционной системы является обеспечение обмена данными между приложениями и периферийными устройствами компьютера, т.е. операционная система должна управлять всеми устройствами ввода-вывода.

Устройства ввода-вывода делятся на две категории: блочные и символьные устройства. Блочными называются устройства, хранящие информацию в виде блоков фиксированного размера, причём у каждого блока имеется адрес. Каждый блок может быть прочитан независимо друг от друга. Блочными устройствами являются жёсткие диски. Символьное устройство принимает или предоставляет поток символов без какой-либо блочной структуры. Это принтеры, сетевые карты, мыши. Однако классификация на блочные и символьные устройства не покрывает все возможные устройства, например, часы, суть работы которых состоит в инициировании прерываний в определённые моменты времени. Скорость работы устройств ввода-вывода колеблется от 10 байт в секунду да десятков гигабайт в секунду.

Обычно устройство ввода-вывода состоит из механической части и электронной части. Электронный компонент устройства называется контроллером устройства или адаптером, который принимает вид печатной платы, вставляемой в разъём.

Интерфейс низкого уровня между устройством и контроллером обеспечивает конвертирование последовательного потока битов в блок байтов и выполнение коррекции ошибок. После чего, этот блок байтов уже обслуживает операционная система.

У каждого контроллера есть несколько регистров, с помощью которых с ним может общаться центральный процессор. Записывая туда – процессор требует предоставить данные, и напротив, считывая оттуда информацию, процессор узнаёт о состоянии устройства. Помимо регистров у многих устройств есть буфер данных, из которых операционная система может читать и записывать туда (например, видеопамять).

Существуют два способа реализации доступа к управляющим регистрам и буферам данных устройств ввода-вывода.

1 Каждому управляющему регистру назначается номер порта ввода-вывода, 8-или 16-разрядное целое число. Таким образом работали самые древние компьютеры. И при такой схеме адресные пространства ОЗУ и устройств ввода-вывода не пересекаются. (Рисунок 40, а).

2 Отображение всех управляющих регистров периферийных устройств на адресное пространство памяти (Рисунок 40, б).

Ну и конечно, существуют гибридные схемы. Оба метода имеют сильные и слабые стороны.

Достоинства ввода-вывода, отображаемого на адресное пространство:

- для обращения к устройствам ввода-вывода не нужны специальные команды, что упрощает написание программ по сравнению с отдельным адресным

пространством;

- не требуется специального механизма защиты от пользовательских процессов, обращающихся к устройствам ввода-вывода, т.к. область памяти с портами исключается из адресного пространства пользователей;
- каждая команда процессора для обращения к памяти может использоваться и для работы с портами.

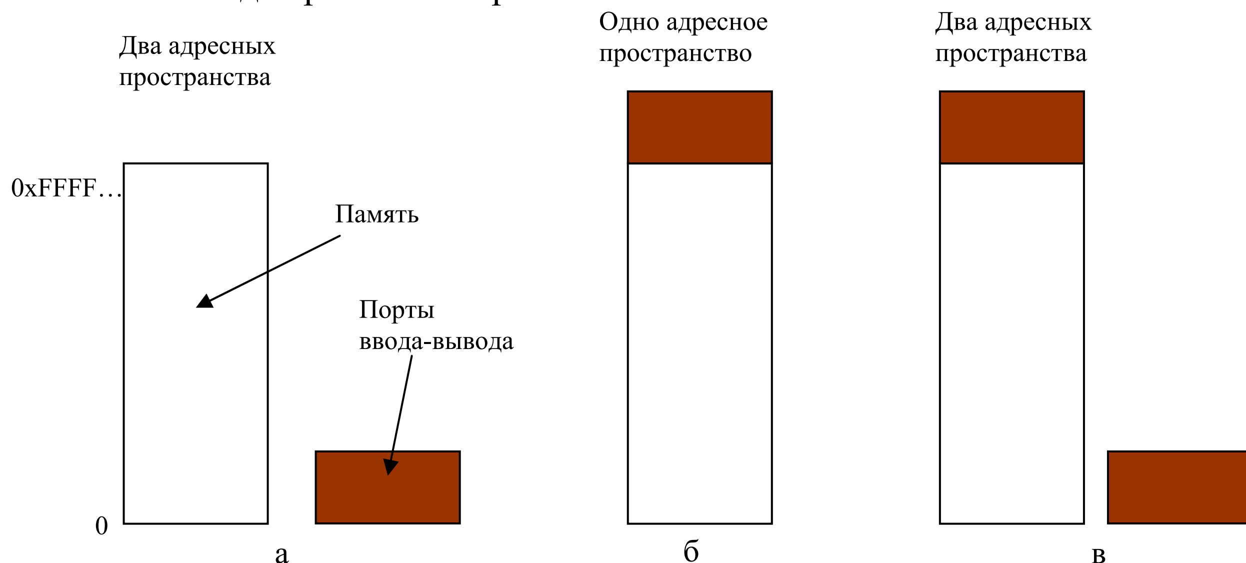

Рисунок 40 – Раздельные адресные пространства (а); отображаемый на адресное пространство ввод-вывод (б); гибрид (в)

Недостатки ввода-вывода, отображаемого на адресное пространство:

- регистры ввода-вывода нельзя кэшировать, т.к. в этом случае мы бы никогда не узнали состояния портов, поэтому увеличивается сложность управления избирательным кэшированием;
- все устройства ввода-вывода должны изучать все обращения к памяти центрального процессора, что в схемах с более чем одной шиной можно сделать только с помощью фильтрации адресов специальной микросхемой, что и сделано ещё на базе процессора Pentium I.

На практике центральный процессор не опрашивает по байту устройство ввода-вывода, а использует прямой доступ к памяти DMA (Direct Memory Access). Операционная система пользуется прямым доступом к памяти через аппаратный DMA-контроллер, если конечно он есть в конфигурации данного компьютера. Чтобы понять различие между DMA и доступом к устройству ввода-вывода напрямую рассмотрим, как происходит чтение с жесткого диска.

При отсутствии DMA.

1 Контроллер считывает с диска один или несколько секторов последовательно, пока весь блок не окажется в буфере контроллера.

2 Контроллер проверяет контрольную сумму – не было ли ошибок.

3 Контроллер инициирует прерывание.

4 Операционная система читает блок диска побайтно или пословно с контроллера.

При использовании DMA:

1 Центральный процессор программирует DMA-контроллер, устанавливая его регистры и указывая, какие данные и куда следует переместить. Даёт команду дисковому котроллеру, прочитать данные во внутренний буфер и проверить его содержимое.

2 DMA-контроллер начинает перенос данных, посылая дисковому контроллеру по шине запрос чтения.

3 Перенос данных из контроллера жесткого диска в ОЗУ.

4 По окончании записи контроллер диска посылает сигнал подтверждения DMA-контроллеру.

Шаги 2-4 повторяются, пока в память не будь считано необходимое количество данных. Операционной системе не нужно заниматься копированием блока диска в память, т.к. он уже находится там. Следовательно, разгружается центральный процессор.

Большое значение для осуществления ввода-вывода имеет прерывание. Прерывание (англ. interrupt) – сигнал, сообщающий процессору о совершении какого-либо асинхронного события. При этом выполнение текущей последовательности команд приостанавливается, и управление передаётся обработчику прерывания, который выполняет работу по обработке события и возвращает управление в прерванный код.

Структура прерываний представлена на рисунке 41.

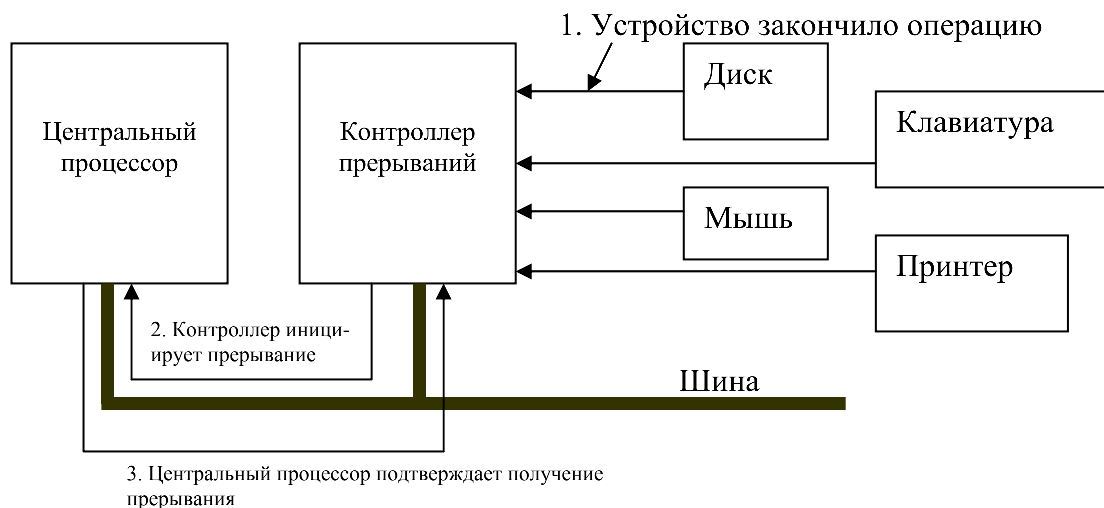

Рисунок 41 – Схема прерываний в компьютере

Здесь стоит отметить, что соединения между устройством и контроллером прерываний в действительности являются специальными линиями шины, а не выделенными проводами.

Когда устройство ввода-вывода заканчивает свою работу, оно инициирует прерывание. При отсутствии других запросов прерывания контроллер обрабатывает прерывание немедленно. В противно случае прерывание игнорируется, а устройство ввода-вывода продолжает удерживать сигнал о прерывании для контроллера. Для обработки прерывания контроллер выставляет на адресную шину номер устройства, требующего к себе внимания, и устанавливает сигнал прерывания на соответствующий контакт процессора. Этот сигнал заставляет процессор приостановить текущую работу и начать выполнять обработку прерывания. Номер,

выставленный на адресную шину, используется в качестве индекса в таблице, называемой вектором прерываний (о чём уже упоминалось ранее), из которой извлекается новое значение счетчика команд. Новый счетчик команд указывает на начало соответствующей процедуры обработки прерывания. Вскоре после начала своей работы процедура обработки прерываний подтверждает получение прерывания, что разрешает контроллеру издавать новые прерывания.

Прежде, чем начать обработку прерываний, необходимо сохранить определенную информацию, например, счетчик команд и регистры центрального процессора. Данная информация сохраняется в стеке.

5.2 Принципы программного обеспечения ввода-вывода

Существует несколько задач для программного обеспечения ввода-вывода [14].

- *Независимость от устройств.* Что означает возможность написания программ, способных получать доступ к любому устройству ввода-вывода, без предварительного указания конкретного устройства.
- *Единообразие именования.* Имя файла или устройства должно быть просто текстовой строкой или целым числом и никоим образом не зависеть от физического устройства.
- *Обработка ошибок.* Ошибки должны обрабатываться как можно ближе к аппаратуре. Если контроллер обнаружил ошибку, он должен исправить её сам.
- *Способ переноса данных*: синхронный (блокирующий) или асинхронный (управляемый прерываниями). Если мы работаем по прерываниям, т.е. асинхронный способ, то операционная система делает их для пользователя блокирующими – т.е. программа делает системный вызов и ожидает ответа.
- *Буферизация.* Включает копирование данных в значительных размерах и увеличивает производительность операций ввода-вывода.

Важно понятие выделенных устройств и устройств коллективного использования. К первым может иметь доступ только один пользователь в один и тот же момент времени, а ко вторым – несколько пользователей.

Существуют три различных способа осуществления операций ввода-вывода:

- программный ввод-вывод;
- управляемый прерываниями ввод-вывод;
- ввод-вывод с использованием DMA.

Суть программного ввода-вывода рассмотрим на примере печати строки символов на принтере. Первоначально процесс пользователя собирает эту строку в буфере. Затем процесс получает принтер во временное пользование. После этого процесс просит операционную систему распечатать строку символов. Операционная система копирует буфер в пространство ядра, и как только принтер доступен для печати, копирует первый символ в регистр принтера и смещает указатель на следующий символ. И так до тех пор, пока все символы не перенесутся в буфер принтера. По окончании печатается вся строка, и принтер снова становится

доступным для печати. Упрощённо это можно представить в виде программы на языке С (Листинг 5).

```
copy_from_user(buffer,p,count);           /* p –буфер ядра */
for(i=0; i<count;i++){                    /* цикл символов */
      while (*printer_status_reg!=READY); /* цикл ожидания готовности */
      *printer_data_reg=p[i];             /* печать символа */
}
return_to_user();
```

Листинг 5 – Печать строки при помощи программного ввода-вывода

Существенный аспект данного способа проиллюстрированный в примере состоит в том, что после печати каждого символа процессор в цикле опрашивает готовность устройство, т.е. происходит активное ожидание. Программный ввод-вывод легко реализуется, но его недостаток – процессор занимается на все время операции ввода-вывода. Такой подход приемлем только в примитивных встроенных системах.

Рассмотрим тот же пример принтера для управляемого прерываниями ввода-вывода. Для этого обратимся к программе на языке С (Листинг 6).

```
copy_from_user(buffer,p,count);
enable_interrupts();
while (*printer_status_reg!=READY);
*printer_data_reg=p[0];
scheduler(); (планировщик)
```

а

```
if (count==0) {
      unblock_user();
} else {
      *printer_data_reg=p[i];
      count=count-1;
      i=i+1;
}
return_from_interrupt();
```

б

Листинг 6 – Печать строки при помощи ввода-вывода, управляемого прерываниями: программа, выполняемая при обращении к системному вызову (а); процедура обработки прерываний (б)

Когда выполняется системный вызов печати строки, копируется буфер в ядро. Разрешаются прерывания. Первый символ копируется на принтер. Затем процессор вызывает планировщик и может заниматься чем угодно, например выполнением другого процесса, а этот процесс заблокирован на всё время выполнения печати.

Когда символ передался в принтер и тот снова готов принять следующий, он инициирует прерывание. Текущий процесс останавливается и запускается процедура обработки прерывания. В которой – если напечатаны все символы, то процесс отправивший их на печать разблокируется, иначе печатает следующий символ и выходит из прерывания. Очевидный недостаток в том, что прерывания происходят при печати каждого символа. А обработка прерывания занимает определенное время.

И, наконец, рассмотрим способ ввода-вывода с использованием DMA. Идея в том, чтобы позволить контроллеру DMA поставлять символы принтеру по одному, не беспокоя при этом центральный процессор. По существу этот метод отличается

от предыдущего только тем, что всю работу выполняет DMA-контроллер вместо центрального процессора. А прерывание одно – на весь буфер, отправленный на печать.

Программное обеспечение ввода-вывода обычно организуется в виде четырех уровней, показанных на рисунке 42.

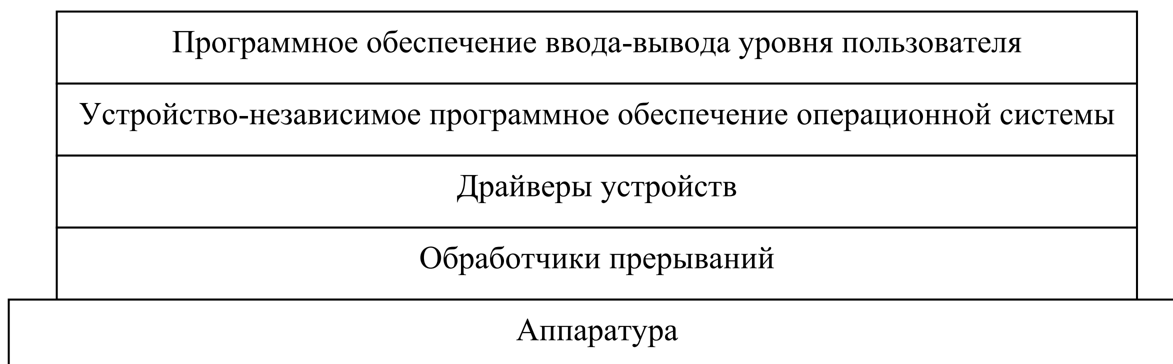

Рисунок 42 – Программные уровни ввода-вывода

Обычно при начале операции ввода-вывода драйвер устройства блокирует сам себя. Когда от аппаратуры приходит прерывание, свидетельствующее об окончании работы, начинает работу обработчик прерываний. По окончании необходимой работы он может разблокировать драйвер, запустивший его.

Чтобы получить доступ к аппаратной части устройства, т.е. к регистрам контроллера, драйвер устройства должен быть частью ядра операционной системы. Драйверы устройств обычно располагаются под остальной частью операционной системы, так как показано на рисунке 43. Иерархически структура отображена для наглядности, т.к. на самом деле весь обмен информацией между драйверами и контроллерами устройств идёт по шине.

Операционная система обычно классифицирует драйверы по нескольким категориям в соответствии с типами обслуживаемых ими устройств. К наиболее общим категориям относятся блочные устройства, например, диски, содержащие блоки данных, к которым возможна независимая адресация, и символьные устройства, такие как клавиатуры и принтеры, формирующие или принимающие поток символов.

В большинстве операционных систем определен стандартный интерфейс, который должны поддерживать все блочные драйверы, и второй стандартный интерфейс, поддерживаемый всеми символьными драйверами. Эти интерфейсы включают наборы процедур, которые могут вызываться остальной операционной системой для обращения к драйверу. К этим процедурам относятся, например, процедуры чтения блока (блочного устройства) или записи символьной строки (для символьного устройства).

У драйвера устройства есть несколько функций. Наиболее очевидная функция драйвера состоит в обработке абстрактных запросов чтения и записи независимого от устройств программного обеспечения, расположенного над ними. Но кроме этого они должны также выполнять еще несколько функций. Например, драйвер должен

при необходимости инициализировать устройство. Ему также может понадобиться управлять энергопотреблением устройства и регистрацией событий.

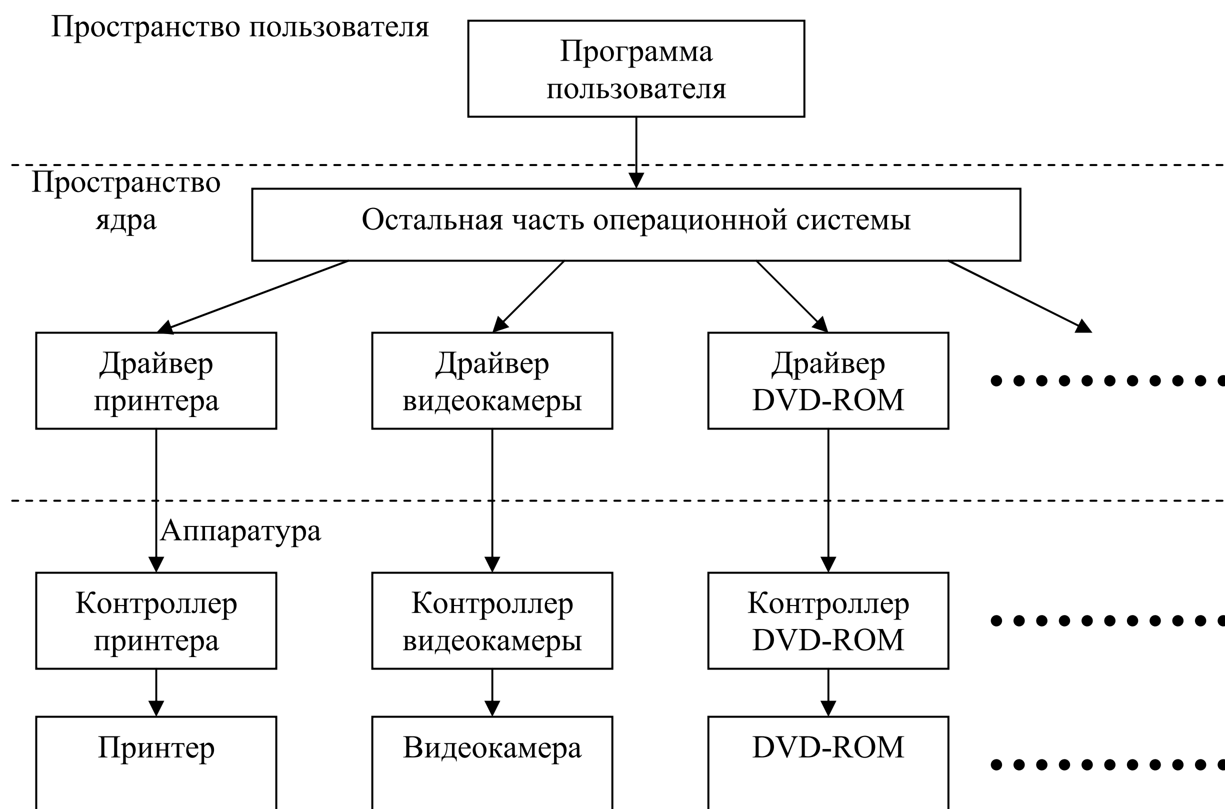

Рисунок 43 – Логическое расположение драйверов устройств

Управление устройством подразумевает выдачу ему серии команд, которые драйвер записывает в регистры устройства. После этого устройство выполняет какие-то операции, по окончании которых драйвер проверяет их безошибочность. Если все в порядке, то данные передаются по уровню выше в независимое от устройств программное обеспечение, которое обычно обладает следующими функциями:

- единообразный интерфейс для драйверов устройств;
- буферизация;
- сообщения об ошибках;
- захват и освобождения выделенных устройств;
- размер блока, независящий от устройства.

Основная задача независимого от устройств программного обеспечения состоит в выполнении функций ввода-вывода, общих для всех устройств, и предоставлении единообразного интерфейса для программ уровня пользователя.

Контрольные вопросы по разделу

1. Чем отличаются блочные и символьные устройства?
2. Какие преимущества отображения всех управляющих регистров

периферийных устройств на адресное пространство памяти вы можете назвать?

3 Зачем необходим прямой доступ к памяти DMA?

4 Перечислите задачи, которые должно решать программное обеспечение ввода-вывода?

5 Назовите три различных способа осуществления ввода-вывода? Каковы их преимущества и недостатки?

6 Объясните, как осуществляется ввод-вывод, управляемый прерываниями.

7 Какова сущность программного ввода-вывода?

8 Какими функциями обладает независимое от устройств программное обеспечение?

6 Файловые системы

6.1 Основы файловых систем

Одной из основных задач операционной системы является предоставление удобств пользователю при работе с данными, хранящимися на дисках. Для этого операционная система подменяет физическую структуру хранящихся данных некоторой удобной для пользователя логической моделью. Логическая модель файловой системы материализуется в виде дерева каталогов, в символьных составных именах файлов, в командах работы с файлами. Базовым элементом этой модели является файл, который так же, как и файловая система в целом, может характеризоваться как логической, так и физической структурой.

Файл – это именованная область внешней памяти, в которую можно записывать и из которой можно считывать данные. Файлы хранятся в памяти, на зависящей от энергопитания, обычно – на магнитных дисках. Одним из исключений является так называемый электронный диск, когда в оперативной памяти создается структура, имитирующая файловую систему.

Основные цели использования файла [11].

- *Долговременное и надежное хранение информации.* Долговременность достигается за счет использования запоминающих устройств, не зависящих от питания, а высокая надежность определяется средствами защиты доступа к файлам и общей организацией программного кода операционной системы, при которой сбои аппаратуры чаще всего не разрушают информацию, хранящуюся в файлах.
- *Совместное использование информации.* Файлы обеспечивают естественный и легкий способ разделения информации между приложениями и пользователями за счет наличия понятного человеку символьного имени и постоянства хранимой информации и расположения файла. Пользователь должен иметь удобные средства работы с файлами, включая каталоги-справочники, объединяющие файлы в группы, средства поиска файлов по признакам, набор команд для создания, модификации и удаления файлов. Файл может быть создан одним пользователем, а затем использоваться совсем другим пользователем, при этом создатель файла или администратор могут определить права доступа к нему других пользователей. Эти цели реализуются в операционной системе файловой системой.

Файловая система – это часть операционной системы, включающая:

- совокупность всех файлов на диске;
- наборы структур данных, используемых для управления файлами, такие, например, как каталоги файлов, дескрипторы файлов, таблицы распределения свободного и занятого пространства на диске;
- комплекс системных программных средств, реализующих различные операции над файлами, такие как создание, уничтожение, чтение, запись, именование и поиск файлов.

Файловая система позволяет программам обходиться набором достаточно простых операций для выполнения действий над некоторым абстрактным объектом,

представляющим файл. При этом программистам не нужно иметь дело с деталями действительного расположения данных на диске, буферизацией данных и другими низкоуровневыми проблемами передачи данных с долговременного запоминающего устройства. Все эти функции файловая система берет на себя. Файловая система распределяет дисковую память, поддерживает именование файлов, отображает имена файлов в соответствующие адреса во внешней памяти, обеспечивает доступ к данным, поддерживает разделение, защиту и восстановление файлов.

Файловая система играет роль промежуточного слоя, экранирующего все сложности физической организации долговременного хранилища данных, и создающего для программ более простую логическую модель этого хранилища, а также предоставляя им набор удобных в использовании команд для манипулирования файлами.

Задачи, решаемые файловой системой, зависят от способа организации вычислительного процесса в целом. Самый простой тип – это файловая система в однопользовательских и однопрограммных операционных системах (например, MS-DOS). Основные функции в такой файловой системе нацелены на решение следующих задач:

- именование файлов;
- программный интерфейс для приложений;
- отображения логической модели файловой системы на физическую организацию хранилища данных;
- устойчивость файловой системы к сбоям питания, ошибкам аппаратных и программных средств.

Задачи файловой системы усложняются в операционных однопользовательских мультипрограммных системах. К перечисленным выше задачам добавляется новая задача совместного доступа к файлу из нескольких процессов. Файл в этом случае является разделяемым ресурсом, а значит, файловая система должна решать весь комплекс проблем, связанных с такими ресурсами. В частности, должны быть предусмотрены средства блокировки файла и его частей, исключение тупиков, согласование копий и т. п. В многопользовательских системах появляется еще одна задача: защита файлов одного пользователя от несанкционированного доступа другого пользователя.

Файловые системы поддерживают несколько функционально различных типов файлов, в число которых, как правило, входят обычные файлы, файлы-каталоги, специальные файлы, именованные конвейеры, отображаемые в память файлы и другие.

Обычные файлы, или просто файлы, содержат информацию произвольного характера, которую заносит в них пользователь или которая образуется в результате работы системных и пользовательских программ.

Каталоги – это особый тип файлов, которые содержат системную справочную информацию о наборе файлов, сгруппированных пользователями по какому-либо неформальному признаку.

Специальные файлы – это фиктивные файлы, ассоциированные с устройствами ввода-вывода, которые используются для унификации механизма доступа к файлам и внешним устройствам.

Пользователи обращаются к файлам по символьным именам. Однако способности человеческой памяти ограничивают количество имен объектов, к которым пользователь может обращаться по имени. Иерархическая организация пространства имен позволяет значительно расширить эти границы. Именно поэтому большинство файловых систем имеет иерархическую структуру, в которой уровни создаются за счет того, что каталог более низкого уровня может входить в каталог более высокого уровня (Рисунок 44).

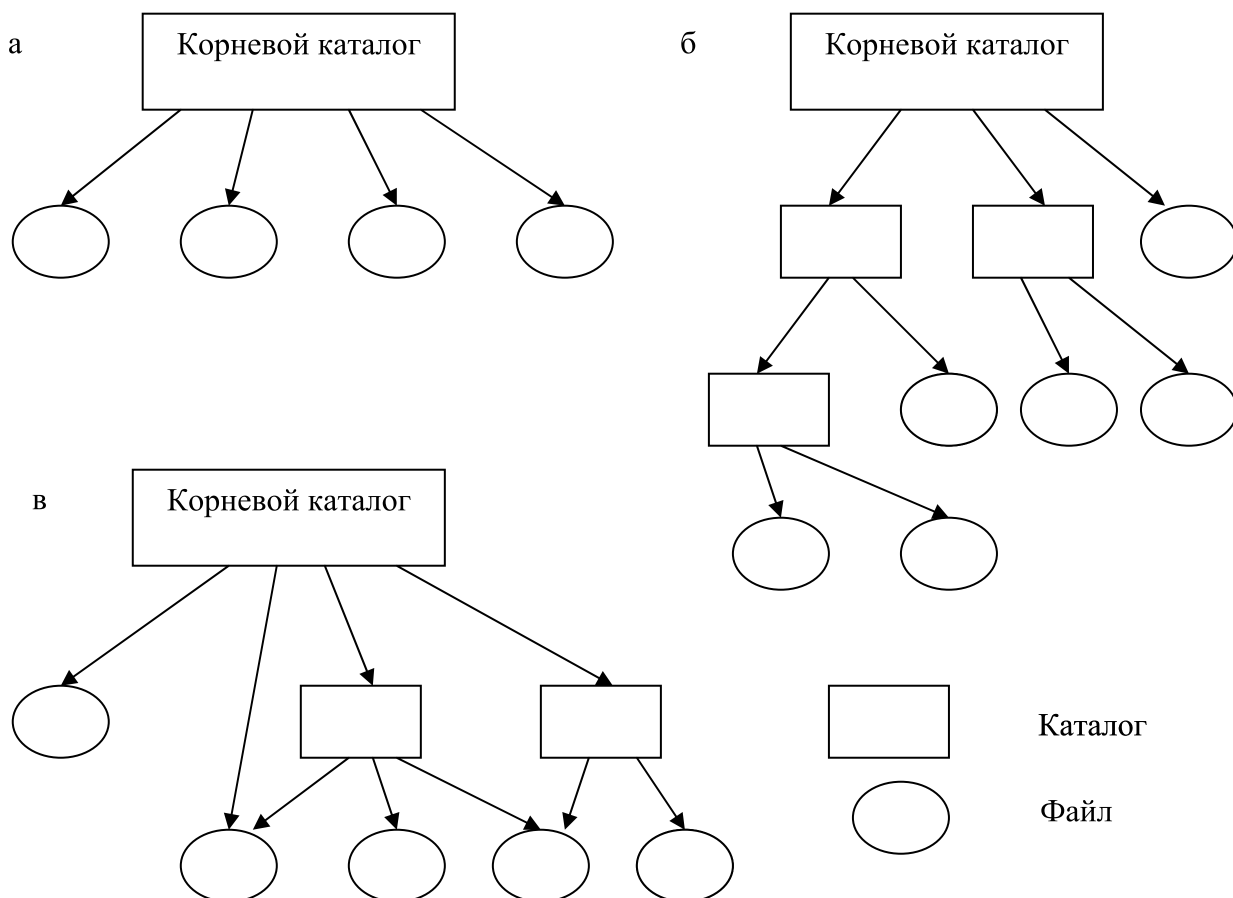

Рисунок 44 – Иерархия файловых систем

Каталоги образуют дерево, если файлу разрешено входить только в один каталог (Рисунок 44, б), и сеть – если файл может входить сразу в несколько каталогов (Рисунок 44, в). Например, в MS-DOS и Windows каталоги образуют древовидную структуру, а в UNIX – сетевую. В древовидной структуре каждый файл является листом. Каталог самого верхнего уровня называется корневым каталогом, или корнем (root). Частным случаем иерархической структуры является одноуровневая организация, когда все файлы входят в один каталог (Рисунок 44, а).

Все типы файлов имеют символьные имена. В иерархически организованных файловых системах обычно используются три типа имен файлов: простые, составные и относительные. Простое, или короткое, символьное имя идентифицирует файл в пределах одного каталога. Полное имя представляет собой цепочку простых символьных имен всех каталогов, через которые проходит путь от

корня до данного файла. Относительное имя файла определяется через понятие «текущий каталог».

При наличии нескольких устройств внешней памяти организация хранения файлов возможна двумя способами:

- на каждом устройстве размещается автономная файловая система (пример, MS-DOS, Windows);
- объединение в единую файловую систему с единым деревом каталогов, т.е. монтирование (UNIX).

Понятие «файл» включает не только хранимые им данные и имя, но и атрибуты. Атрибуты – это информация, описывающая свойства файла. (тип файла, владелец файла, признак «только для чтения» и т.д.). Пользователь может получать доступ к атрибутам, используя средства, предоставленные для этих целей файловой системой. Обычно разрешается читать значения любых атрибутов, а изменять – только некоторые. Например, пользователь может изменить права доступа к файлу (при условии, что он обладает необходимыми для этого полномочиями), но изменять дату создания или текущий размер файла ему не разрешается.

Значения атрибутов файлов могут непосредственно содержаться в каталогах, как это сделано в файловой системе FAT. Другим вариантом является размещение атрибутов в специальных таблицах, когда в каталогах содержатся только ссылки на эти таблицы. Такой подход реализован, например, в файловой системе UFS. В этой файловой системе структура каталога очень простая. Запись о каждом файле содержит короткое символьное имя файла и указатель на индексный дескриптор файла, так называется в UFS таблица, в которой сосредоточены значения атрибутов файла.

Файловые системы хранятся на дисках. Сектор 0 диска называется главной загрузочной записью MBR (Master Boot Record) и используется для загрузки компьютера. В конце MBR содержится таблица разделов, в которой хранятся начальные и конечные адреса каждого раздела. При загрузке компьютера BIOS считывает и исполняет MBR-запись, после чего определяется активный раздел и загрузчик в MBR-записи исполняет его. Программа, находящаяся в загрузочном блоке раздела, загружает операционную систему. Возможная структура файловой системы представлена на рисунке 45.

Суперблок содержит ключевые параметры файловой системы, включающая в себя количество блоков в файловой системе и другую административную информацию. I-узлы – массив структур данных, содержащих информацию о файлах.

Важным моментом в реализации хранения файлов является учет соответствия блоков диска файлам. Для определения того, какой блок какому файлу принадлежит, в различных операционных системах применяются различные методы.

Реализация файлов возможна следующими способами.

1 *Непрерывные файлы.* Файлы представляют собой непрерывные наборы соседних блоков диска. Преимущества: простота реализации плюс высокая производительность, т.к. весь файл может быть прочитан с диска за одну операцию. Недостаток: в результате фрагментации необходимо будет знать конченый размер

файла перед записью или постоянно производить дефрагментацию. Непрерывные файлы возможно применять на CD и DVD дисках.

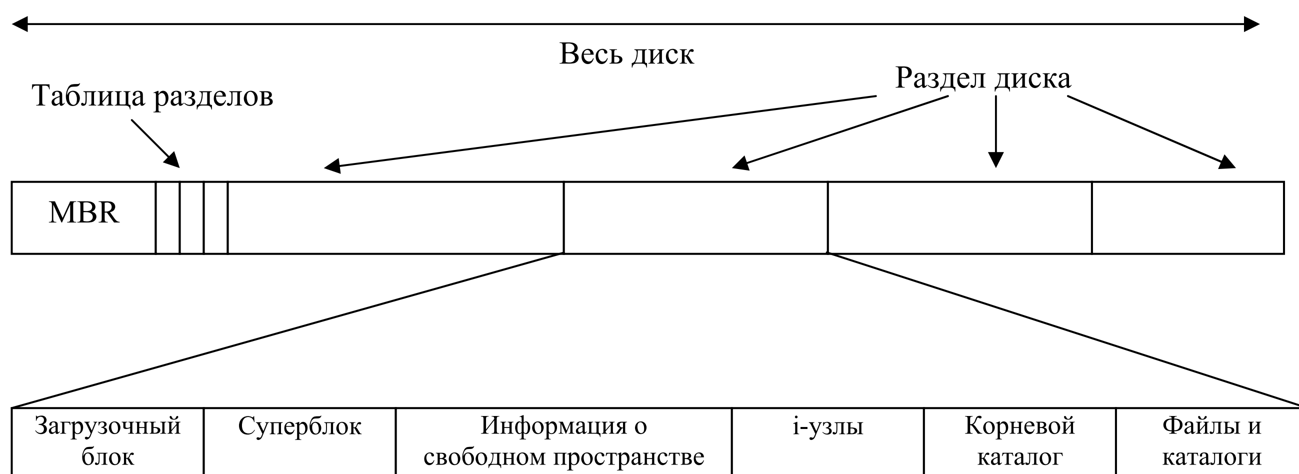

Рисунок 45 – Возможная структура файловой системы

2 *Связные списки.* Файл состоит из блоков диска, как показано на рисунке 46. Первое слово каждого блока используется как указатель на следующий блок. Недостатки: если мы хотим прочитать информацию в конце файла, нам необходимо его читать сначала, что очень медленно. Кроме того теряется место на содержание указателей на следующий блок.

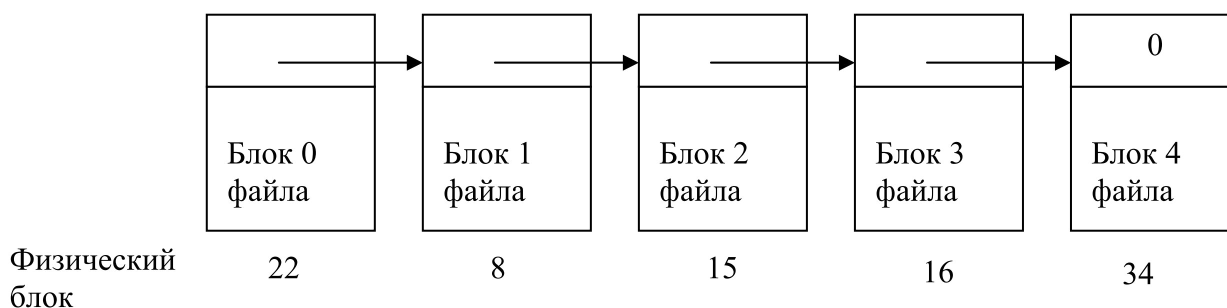

Рисунок 46 – Размещение файла в виде связного списка блоков диска

3 *Связный список при помощи таблицы в памяти.* В отличии от предыдущего способа все указатели на следующий блоки хранятся в отдельной таблице. Недостаток: вся таблица, которая называется FAT-таблицей, должна находиться в памяти. Что очень много, т.к. для 20-гигабайтного диска с блоками размером 1 Кбайт потребовалась бы таблица из 20 миллионов записей. Каждая из которых не менее 4 байт. А это 80 Мбайт постоянно занятой оперативной памяти.

4 *I-узлы.* В этом случае с каждым файлом связывается структура данных (i-узел – index-узел), содержащей атрибуты файлов и адреса блоков файла. Перед работой с файлом i-узел читает в память все адреса блоков. Преимущество: мало занимаемая ОЗУ.

6.2 Файловая система FAT

Файловая система FAT (File Allocation Table – Файловая таблица распределения) является одной из простейших систем. Основная концепция файловой системы FAT заключается в том, что каждому файлу и каталогу выделяется структура данных, называемая записью каталога. В этой структуре хранится имя файла, его размер, начальный адрес содержимого файла и другие метаданные. Содержимое файлов и каталогов хранится в блоках данных, называемых кластерами. Если файлу или каталогу выделяется более одного кластера, остальные кластеры находятся при помощи структуры данных, называемой FAT. Существуют 3 версии FAT: FAT12, FAT16 и FAT32. Они отличаются между собой размером записей в структуре FAT. На рисунке 47 показана общая схема между структурами данных [5].

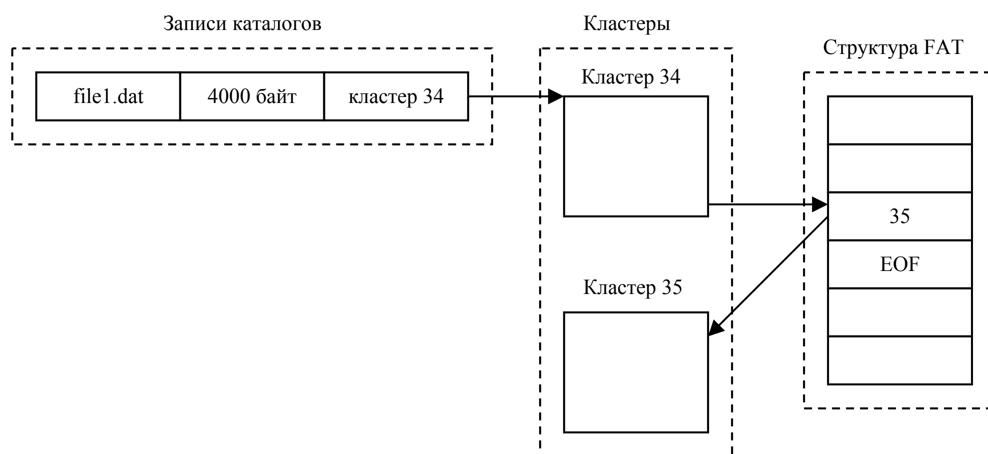

Рисунок 47 – Отношения между записями каталогов, кластерами и FAT

Файловая система FAT делится на три физические области.

- Зарезервированная область, в которой хранятся данные из категории файловой системы. Размер её определяется в загрузочном секторе. В FAT12 и FAT16 занимает всего 1 сектор.
- Область FAT – содержит основные и резервные структуры FAT. Она начинается в секторе, следующем за зарезервированной область, а её размер определяется количеством и размером структур FAT.
- Область данных содержит кластеры, выделяемые для хранения файлов и содержимого каталогов (Рисунок 48).

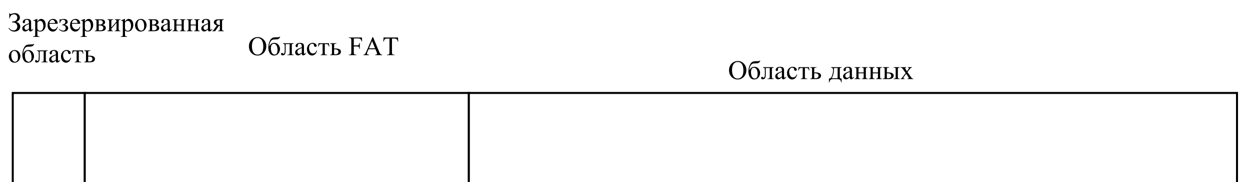

Рисунок 48 – Физическая структура файловой системы FAT

Для того, чтобы открыть файл операционная система должна прочитать соответствующую запись каталога. Первоначальная каталоговая запись системы FAT представлена на рисунке 49.

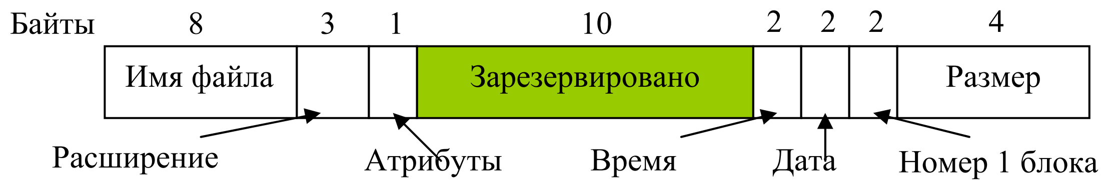

Рисунок 49 – Формат каталоговой записи в системе FAT

FAT-12, 16 и 32 различаются размерами минимальных блоков (кластеров), общим максимальным объемом диска и разрядностью указателей на эти блоки (12, 16, 28 разряда). FAT-12 применялась на гибких дисках. Размер дискового раздела мог составлять 2 Мб, а размер блока 512 байт, 1 Кб, 2 Кб, 4 Кб. FAT-16 вы можете использовать и сейчас, например отформатировав флеш-носитель небольшого размера. Кластеры здесь размером 8, 16 или 32 Кб. Максимальный размер дискового раздела (логический диск) – 2 Гб, максимальный размер диска – 8 Гб. Таблица FAT занимает в памяти 128Кб. В FAT32 размер разделов ограничен 2Тб (2048 Гб). Размеры кластеров остались прежними. FAT32 широко используется и по сей день. При этом изменился формат каталоговой записи, который представлен на рисунке 50.

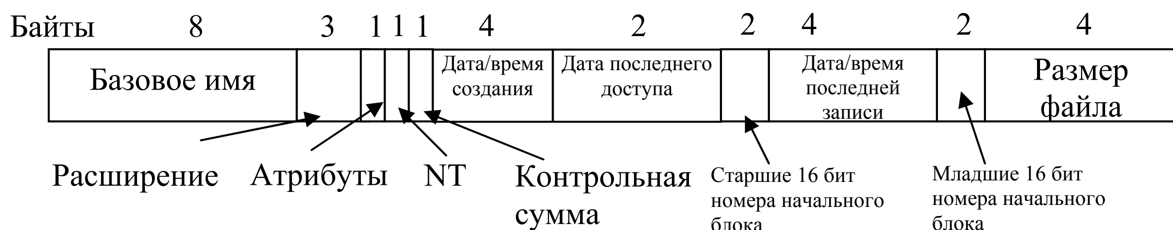

Рисунок 50 – Формат каталоговой записи в системе FAT32

Если у файла есть также длинное имя, оно хранится в одной или нескольких каталоговых записях, предшествующих описателю файла (Рисунок 50). Каждая такая запись содержит до 13 символов формата Unicode. Элементы имени хранятся в обратном порядке, начинаясь сразу перед описателем файла в формате MS-DOS и последующими фрагментами перед ним. Формат каждого фрагмента имени представлен на рисунке 51.

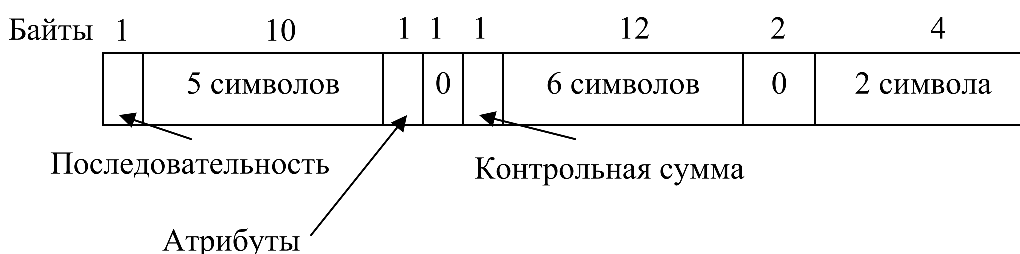

Рисунок 51 – Формат каталоговой записи с фрагментом длинного имени файла

Операционная система отличает стандартные каталоговые записи от записей с фрагментом длинного файла по полю Attributes (атрибуты). Для фрагмента длинного имени это поле содержит значение 0x0F, что соответствует невозможной комбинации атрибутов для описателя файла в MS-DOS. Старые программы, написанные для работы в MS-DOS, читая каталог, просто игнорируют такие описатели как неверные.

Реализация файловой системы FAT-32 концептуально близка к реализации файловой системы FAT-16. Однако вместо массива из 65 536 элементов в ней используется столько, сколько нужно, чтобы покрыть весь раздел диска. Если диск содержит миллион блоков, то и таблица будет состоять из миллиона элементов. Для экономии памяти система Windows 98 не хранит их все сразу в памяти, а использует окно, накладываемое на таблицу.

6.3 Файловая система NTFS

Основными целями при проектировании NTFS (New Technology File System – Файловая система новой технологии) были надежность, безопасность и поддержка носителей информации большой емкости. Основные особенности файловой системы NTFS следующие [5].

- *Способность восстановления данных.* Файловая система восстанавливается при отказе системы и сбоев дисков. Это достигнуто по средствам использования механической транзакции, при котором осуществляется журналирование файловой операции.
- *Безопасность.* Файловая система поддерживает объектную модель безопасности и рассматривает все тома, каталоги, файлы как самостоятельные объекты. NTFS обеспечивает безопасность на уровне файлов, это означает, что право доступа к файлам зависит от учетной записи пользователя, и тех групп к которым он принадлежит.
- *Расширенная функциональность.* NTFS проектировалась с учетом возможного расширения. В ней реализованы такие возможности, как эмуляция других операционных систем, параллельная обработка потоков данных и создание файловых атрибутов определенных пользователем.
- *Поддержка POSIX (Portable Operating System for computing environments).* Международный стандарт машинно-независимого интерфейса вычислительной среды. В нем основное внимание уделяется взаимодействию прикладных программ с операционной системой. Написанная прикладная программа позволяет создавать программы легко переносимые из одной операционной системы в другую.
- *Эффективная поддержка больших дисков и файлов.* Максимальный размер тома NTFS составляет 2^{64} байт = 1 Экзобайт = 16000 млр. Гб. Максимальный размер файла составляет 2^{32} кластера = 2^{64} байт. Размер кластера может меняться от 512 байт до 64 Кбайт. NTFS поддерживает длинные имена файлов, набор символов Unicode и имена 8.3. Количество файлов в корневом и не корневом каталоге не ограниченно.

NTFS не обладает жестко заданной структурой. Вся файловая система считается областью данных, и любой сектор может быть выделен файлу. Единственное фиксированное требование – это первые сектора тома содержат загрузочный сектор и загрузочный код.

«Сердцем» NTFS является главная файловая таблица MFT (Master File Table – Общая таблица файлов), содержащая информацию обо всех файлах и каталогах. Каждый файл или каталог представлен как минимум одной записью таблицы, причём записи сами по себе очень просты. Их размер составляет 1 Кбайт, но только первые 42 байта имеют определенное предназначение. В остальных байтах хранятся атрибуты – небольшие структуры данных, выполняющие строго специализированную функцию. Например, один атрибут используется для хранения имени файла, а другой – для хранения его содержимого. На рисунке 52 показана основная структура записи MFT с заголовком и тремя атрибутами.

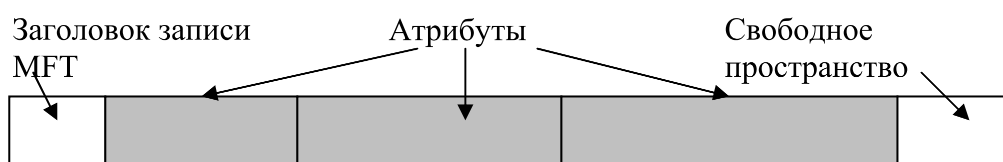

Рисунок 52 – Структура записи MFT

Количество атрибутов зависит от версии NTFS и характеристик описываемого объекта. Атрибут – это объект, содержащий данные определенного типа. Существуют атрибуты для имени файла, даты, времени и даже для содержимого файлов. В этом проявляется одно из отличий NTFS от других файловых систем. Как правило, файловые системы читают и записывают содержимое файлов, а NTFS читает и записывает атрибуты, одна из разновидностей которых передаёт содержимое файлов. В таблице 4 перечислены некоторые стандартные типы атрибутов и соответствующие им идентификаторы. Не все типы атрибутов и идентификаторы существуют для каждого файла.

Таблица 4 – Некоторые стандартные типы атрибутов в записях MFT

Идентификатор типа	**Имя**	**Описание**
16	$STANDARD_INFORMATION	Общая информация (флаги; время создания, последнего обращения и модификации; владелец и идентификатор системы безопасности)
32	$ATTRIBUTE_LIST	Список других атрибутов файла
48	$FILE_NAME	Имя файла в Unicode; время создания, последнего обращения и модификации
64	$VOLUME_VERSION	Информация о томе. Существует только в версии 1.2
80	$SECURITY_DESCRIPTOR	Время обращения и свойства безопасности файла
96	$VOLUME_NAME	Имя тома
112	$VOLUME_INFORMATION	Версия файловой системы и другие флаги
128	$DATA	Содержимое файла
144	$INDEX_ROOT	Корневой узел индексного дерева

Атрибут состоит из заголовка и содержимого. Заголовок определяет тип атрибута, его размер и имя, содержит флаги, указывающие на сжатие или шифрование.

Содержимое атрибутов имеет произвольный формат и произвольный размер. Естественно неудобно сохранять такое количество данных в 1Кбайтных записях MFT. Для решения этой проблемы в NTFS предусмотрена возможность хранения содержимого атрибутов в двух местах: резидентные атрибуты хранятся в MFT записях с заголовками, нерезидентные атрибуты хранятся во внешних кластерах файловой системы. Что представлено на рисунке 53.

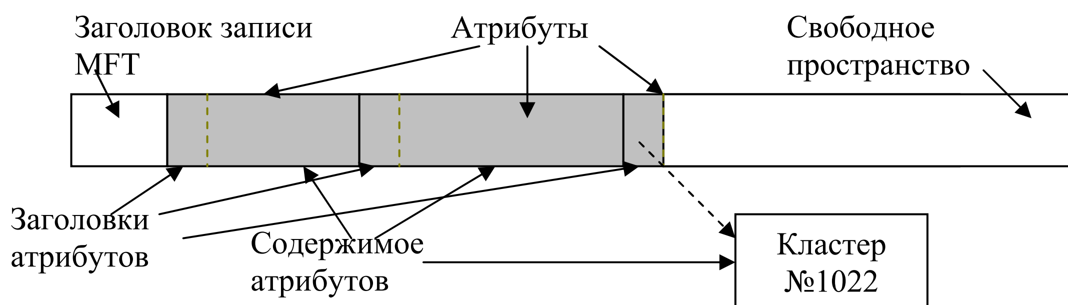

Рисунок 53 – Структура записи MFT с заголовками и содержимым атрибутов

Теоретически файл может содержать до 65536 атрибутов (из-за 16-разрядных идентификаторов), поэтому для хранения всех заголовков атрибутов одной записи MFT может быть недостаточно. Поэтому создается базовая MFT-запись и ссылается на другие MFT-записи. Чтобы уменьшить объем места, занимаемого файлом, NTFS может сохранять значения некоторых нерезидентных атрибутов в разреженном формате, т.е. заполненные нулями кластеры не записываются на диск. NTFS позволяет хранить атрибуты в сжатом виде, а также применяться шифрование атрибутов.

Таким образом структура раздела файловой системы NTFS имеет вид, представленный на рисунке 54. Первые 12 % диска отводятся под так называемую MFT зону – пространство, в которое растут MFT записи. Запись каких-либо данных в эту область невозможна. MFT-зона всегда держится пустой – это делается для того, чтобы самый главный, служебный файл (MFT) не фрагментировался при своем росте. Остальные 88% диска представляют собой обычное пространство для хранения файлов.

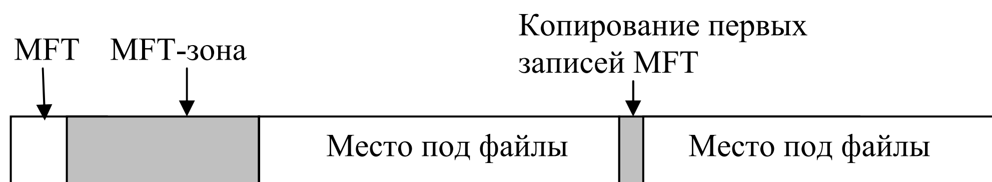

Рисунок 54 – Структура раздела NTFS

Для упрощения поиска в NTFS используются индексные структуры данных, в которых атрибуты сортируются в виде В-деревьев. Деревом называется совокупность структур данных, называемых узлами; узлы связываются между собой, начиная с корневого узла. Пример приведен на рисунке 55, а.

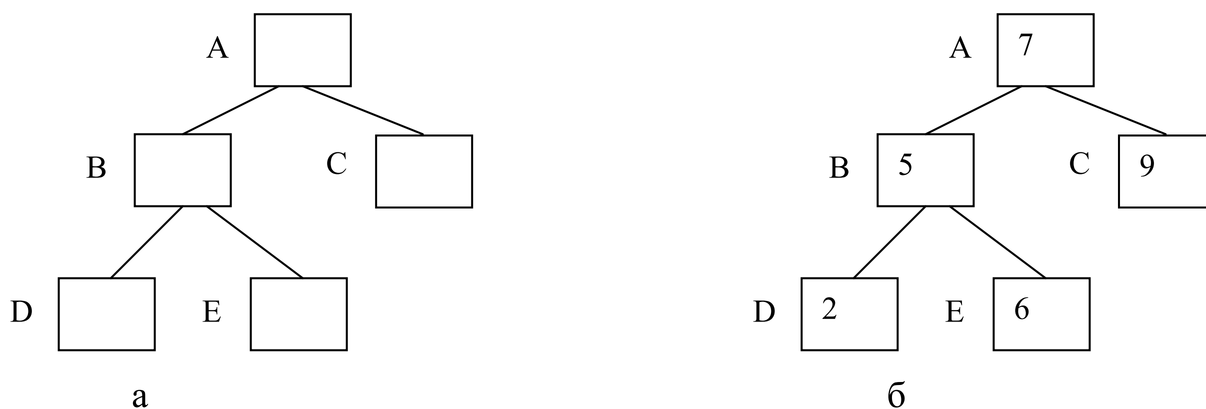

Рисунок 55 – а) дерево с пятью узлами, б) то же дерево после сортировки узлов

Родительским узлом называется узел, от которого идут связи к другим узлам. Дочерним – к кому идут. Облегчение поиска заключается в том, что если нужно найти какое-то значение производится перебор не всех значений, а по дереву. Т.е. если больше корневого узла направо, если меньше налево. В NTFS используется схожая структура индексации, в результате процедура добавления и удаления файла несколько сложна. То же касается и восстановления данных.

Файловая система NTFS достаточно сложна и полное описание функционирования системы и её возможности займёт отдельную книгу, но дополнительную информацию вы можете почерпнуть в публикациях [5, 8, 9].

6.4 Файловые системы Ext2, Ext3 и UFS

Файловая система UFS (Unix File System) является основой для многих других файловых систем, в том числе и популярных в Linux Ext2 и Ext3. Несмотря на различия, эти системы имеют общую структуру, которая и будет рассмотрена ниже.

Основными целями при проектировании этих файловых систем были быстрота и надежность. Копии важных структур данных дублируются в файловой системе, а все данные, ассоциированные с файлом, локализуются, чтобы свести к минимуму перемещение головок жесткого диска во время чтения. Файловая система начинается с необязательной зарезервированной области, а оставшаяся часть делится на секции, называемые группами блоков или группами цилиндров. Все группы, за исключением последней, содержат одинаковое количество блоков, используемых для хранения имен файлов, метаданных и содержимого файлов.

В начале файловой системы находится суперблок с основной информацией о строении. Содержимое каждого файла хранится в блоке, который представляет собой группу смежных секторов. Блоки также могут делиться на фрагменты, которые используются для хранения завершающих байтов файла. Метаданные каждого файла и каталога хранятся в i-узлах. Имена файлов хранятся в записях каталогов, содержащихся в выделенных каталогу блоках. На рисунке 56 представлено отношение между записями каталогов, индексных узлов и блоков данных.

Файловые системы этого типа обладают дополнительными функциями, разделенными на три категории в зависимости от того, что должна делать

операционная система, обнаружив файловую систему с функциями, которые она не поддерживает.

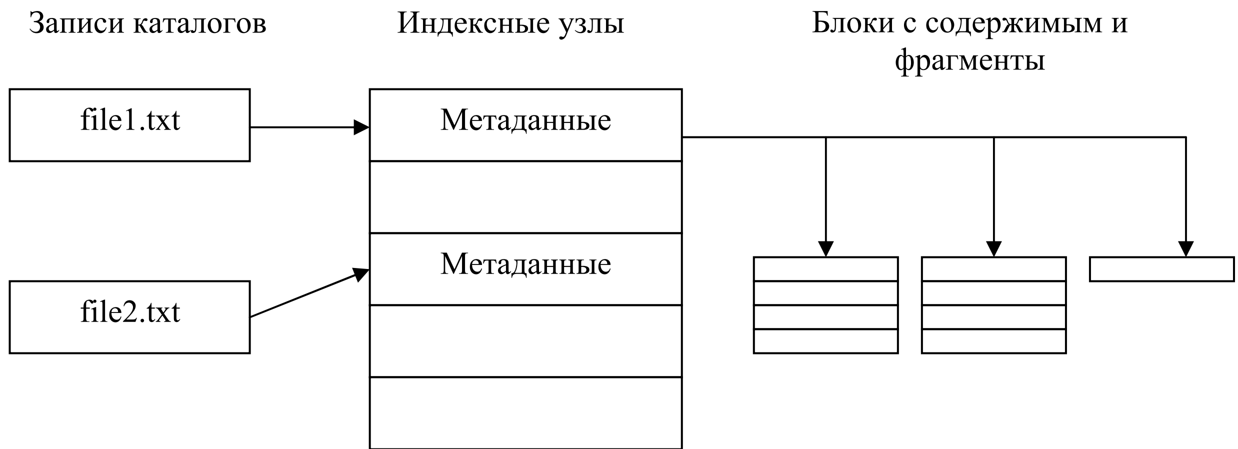

Рисунок 56 – Отношения между записями каталогов, индексными узлами и блоками данных

- *Функции совместимости.* Даже если операционная система не поддерживает какие-то функции, она может смонтировать файловую систему и продолжить работу в обычном режиме.
- *Несовместимые функции.* Столкнувшись с ними операционная система не должна монтировать файловую систему.
- *Совместимые только в режиме чтения.*

В файловой системе ufs на логическом диске (разделе реального диска) находится последовательность секций файловой системы (Рисунок 57).

Рисунок 57 – Структура расположения данных в файловой системе UFS

Суперблок содержит список свободных блоков и свободные i-узлы (information nodes – информационные узлы). В файловых системах ufs для повышения устойчивости поддерживается несколько копий суперблока (как видно из рисунка 57 по одной копии на группу цилиндров). Каждая копия суперблока имеет размер 8196 байт, и только одна копия суперблока используется при монтировании файловой системы. Однако, если при монтировании устанавливается, что первичная копия суперблока повреждена или не удовлетворяет критериям целостности информации, используется резервная копия.

Блок группы цилиндров содержит число i-узлов, специфицированных в списке i-узлов для данной группы цилиндров, и число блоков данных, которые связаны с этими i-узлами. Размер блока группы цилиндров зависит от размера файловой

системы. Для повышения эффективности файловая система ufs старается размещать i-узлы и блоки данных в одной и той же группе цилиндров.

Список i-узлов содержит список i-узлов, соответствующих файлам данной файловой системы. Максимальное число файлов, которые могут быть созданы в файловой системе, определяется числом доступных i-узлов. В i-узле хранится информация, описывающая файл: режимы доступа к файлу, время создания и последней модификации, идентификатор пользователя и идентификатор группы создателя файла, описание блочной структуры файла и т.д.

Блоки данных – в этой части файловой системы хранятся реальные данные файлов. В случае файловой системы ufs все блоки данных одного файла пытаются разместить в одной группе цилиндров. Размер блока данных определяется при форматировании файловой системы и может быть установлен в 512, 1024, 2048, 4096 или 8192 байтов.

Контрольные вопросы по разделу

1. Перечислите функции файловой системы.
2. Чем отличаются функции многопользовательских многопрограммных файловых систем от однопользовательской однопрограммной?
3. Чем отличается иерархия файловых систем UNIX и Windows?
4. Что такое MBR и зачем она нужна?
5. В чём недостаток реализации файлов связными списками?
6. Объясните назначение структуры FAT и её нахождение в памяти?
7. Как хранятся длинные имена файлов в системе FAT-32?
8. Перечислите основные преимущества системы NTFS.
9. Что такое MFT-таблица и MFT-запись?
10. Объясните назначение суперблока в системе UFS.

7 Безопасность операционных систем

7.1 Основы безопасности

С позиций безопасности у компьютерной системы в соответствии с наличествующими угрозами есть три главные задачи:

- конфиденциальность данных;
- целостность данных;
- доступность системы.

Первая задача, конфиденциальность данных, заключается в том, что секретные данные должны оставаться секретными. Вторая задача, целостность данных, означает, что неавторизованные пользователи не должны иметь возможности модифицировать данные без разрешения владельца. Третья задача, доступность системы, означает, что никто не может вывести систему из строя.

Каждой задаче соответствует угроза:

- демонстрация данных;
- порча и подделка данных;
- отказ обслуживания.

В литературе по безопасности человека, сующего свой нос в чужие дела, называют злоумышленником. Злоумышленники бывают активные и пассивные. Пассивные злоумышленники просто пытаются прочитать данные, которые нельзя читать. Активные пытаются незаконно изменить данные. Наиболее распространёнными категориями злоумышленников являются:

- случайные любопытные пользователи, не применяющие специальных технических средств;
- члены организации, занимающиеся шпионажем. Студенты, программисты и т.п.;
- совершающие попытки личного обогащения;
- занимающиеся коммерческим и военным шпионажем.

Помимо угроз со стороны злоумышленников существует опасность случайной потери данных:

- форс-мажор: пожар, землетрясение и т.п.;
- аппаратные и программные ошибки: сбои центрального процессора, ошибки при передачи данных, ошибки в программах;
- человеческий фактор: неправильный ввод данных, удаление и т.п.

Для защиты операционных систем используются различные методы криптографии. Задача криптографии заключается в шифровании данных. Схематично процесс шифрования представлен на рисунке 58.

Секретность зависит от параметров алгоритма шифрования (ключей). ***E*** – шифрование, ***D*** – дешифрование.

Существуют алгоритмы шифрования с секретным и открытыми ключами. Алгоритмы с секретным ключом эффективны и используются в случаях наличия у получателя и отправителя секретного ключа. В шифровании с открытым ключом для шифрования и дешифрования используются различные ключи. Например,

алгоритм RSA. Алгоритмы с открытым ключом широко используются для цифровой подписи платежей.

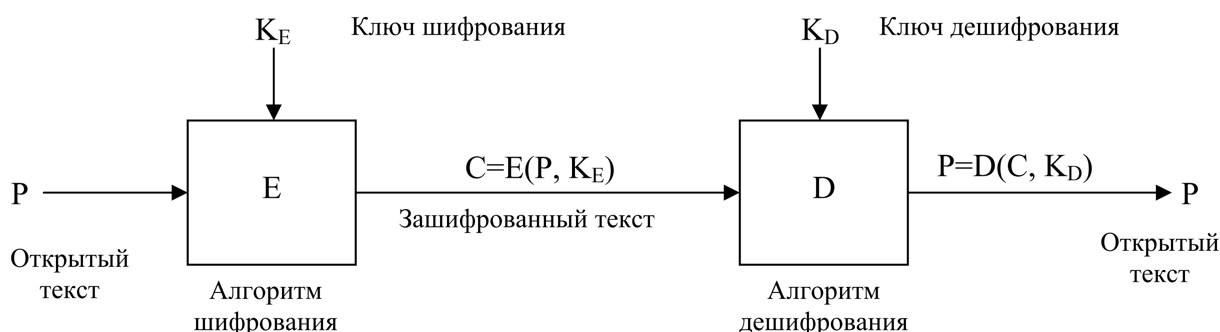

Рисунок 58 – Открытый текст и зашифрованный текст

7.2 Аутентификация пользователей

Аутентификация (authentication) предотвращает доступ к сети нежелательных лиц и разрешает вход для легальных пользователей. Термин «аутентификация» в переводе с латинского означает «установление подлинности». Аутентификацию следует отличать от идентификации. Идентификаторы пользователей используются в системе с теми же целями, что и идентификаторы любых других объектов, файлов, процессов, структур данных, но они не связаны непосредственно с обеспечением безопасности. Идентификация заключается в сообщении пользователем системе своего идентификатора, в то время как аутентификация – это процедура доказательства пользователем того, что он есть тот, за кого себя выдает, в частности, доказательство того, что именно ему принадлежит введенный им идентификатор.

Возможны три способа аутентификации:

- аутентификация с использованием паролей;
- аутентификация с использованием физического объекта;
- аутентификация с использованием биометрических данных.

Наиболее широко применяется аутентификация с использованием паролей, которая легко реализуется. Обычно эту защиту обходят простым перебором комбинаций паролей и имен пользователей. Поэтому в качестве требований к паролям обычно выступают следующие:

- пароль должен содержать как минимум семь символов;
- пароль должен содержать как строчные, так и прописные символы;
- пароль должен содержать как минимум одну цифру или специальный символ;
- пароль не должен представлять собой слово, содержащееся в словаре, имя собственное и т.д.

При использовании многоразового пароля особенно при сетевом доступе есть большой недостаток – его может перехватить злоумышленник и использовать в своих целях. Более надежными оказываются схемы, использующие одноразовые пароли, которые как правило рассчитываются на аутентификацию удалёнными

пользователями. Генерация одноразовых паролей может выполняться либо программно, либо аппаратно. Независимо от того, какую реализацию системы аутентификации на основе одноразовых паролей выбирает пользователь, он, как и в системах аутентификации с использованием многоразовых паролей, сообщает системе свой идентификатор, однако вместо того, чтобы вводить каждый раз один и тот же пароль, он указывает последовательность цифр, сообщаемую ему аппаратным или программным ключом. Через определенный небольшой период времени генерируется другая последовательность – новый пароль.

Еще один вариант идеи паролей заключается в том, что для каждого нового пользователя создается длинный список вопросов и ответов, который хранится на сервере в надежном виде (например, в зашифрованном виде). Вопросы должны выбираться так, чтобы пользователю не нужно было их записывать. При регистрации сервер задает один из этих вопросов, выбирая его из списка случайным образом, и проверяет ответ. Однако чтобы такая схема могла работать, потребуется большое количество пар вопросов и ответов.

Другой вариант называется «оклик-отзыв». Он работает следующим образом. Пользователь выбирает алгоритм, идентифицирующий его как пользователя, например $\boldsymbol{x^2}$. Когда пользователь входит в систему, сервер посылает ему некое случайное число, например 7. В ответ пользователь посылает серверу 49. Алгоритм может отличаться утром и вечером, в различные дни недели и т. д.

Способ аутентификации с использованием физического объекта заключается в проверке некоторого физического объекта, который есть у пользователя. К таким объектам относятся пластиковые карты (например, для банкомата) или смарт-карты, которые содержат в себе примитивную операционную систему. Отличие смарт-карты от обычной пластиковой в том, что на неё есть центральный процессор.

Со смарт-картами могут применяться различные схемы аутентификации. Простой протокол «оклик-отзыв» работает следующим образом. Сервер посылает 512-разрядное случайное число смарт-карте, которая добавляет к нему 512-разрядный пароль, хранящийся в электрически стираемом программируемом ПЗУ. Затем сумма возводится в квадрат, и средние 512 бит посылаются обратно на сервер, которому известен пароль пользователя, поэтому сервер может произвести те же операции и проверить правильность результата. Эта последовательность показана на рисунке 59.

Если даже злоумышленник видит оба сообщения, он не может определить по ним пароль. Сохранять эти сообщения также нет смысла для взломщика, так как в следующий раз сервер пошлет пользователю другое 512-разрядное случайное число. Конечно, вместо возведения в квадрат может применяться (и, как правило, применяется) более хитрый алгоритм.

Недостаток любого фиксированного криптографического протокола состоит в том, что со временем он может быть взломан, в результате чего смарт-карта станет бесполезной. Избежать этого можно, если хранить в памяти карты не сам криптографический протокол, а интерпретатор Java. При этом настоящий криптографический протокол будет загружаться в карту в виде двоичной программы Java и исполняться на ней. Таким образом, как только один протокол взломан, можно мгновенно перейти на использование другого протокола.

Недостаток такого подхода заключается в том, что и без того не отличающаяся высокой производительностью смарт-карта будет работать еще медленнее, однако с развитием технологий этот метод приобретает все большую гибкость.

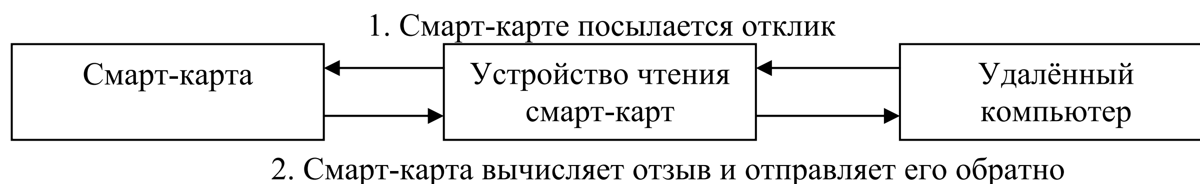

Рисунок 59 – Использование смарт-карты для аутентификации

Третий способ аутентификации основан на измерении физических характеристик пользователя, которые трудно подделать, таких как клавиатурный почерк, длина пальцев, отпечатки пальцев, рисунок сетчатки глаза, голос и т.д. Некоторые методы описаны в работе [1].

Помимо аутентификации могут использоваться контрмеры, направленные на создание ловушек для взломщиков.

7.3 Атаки изнутри операционной системы

Зарегистрировавшись на компьютере, взломщик может начать причинение ущерба. Если на компьютере установлена надежная система безопасности, возможно, взломщик сможет навредить только тому пользователю, чей пароль он взломал, но часто начальная регистрация может использоваться в качестве ступеньки для последующего взлома других учетных записей.

1 Троянские кони

Это вредоносная программа, используемая злоумышленником для сбора информации, её разрушения или модификации, нарушения работоспособности компьютера или использования его ресурсов в неблаговидных целях. К этим целям может относится и осуществление банковский операций через защищенные программы.

Чтобы троянский конь заработал, необходимо запустить его. Обычно это происходит, когда люди скачивают с Интернета игру, новый проигрыватель mp3, специальную программу для просмотра порнографии и т.д. Т.е. при применении троянского коня взлом компьютера не нужен, ведь вредоносную программу запускает сам пользователь.

2 Фальшивая программа регистрации

Ввод пользователя и пароля в окне взломщика. Для того, чтобы взломщик не смог подменить окно смены пользователя в Windows используется от перехвата комбинация CTRL+ALT+DEL. Однако и для этой комбинации есть способы взлома, но в этом случае необходимо внедрить DLL в процесс Winlogon.

3 Логические бомбы

Логическая бомба – это написанная одним из сотрудников программа, тайно установленная в операционную систему, которая при внезапно увольнении сотрудника через некоторое время начинает действовать: например,

форматирование жесткого диска, удалении файлов в случайном порядке или шифровании важных файлов.

4 Потайные двери

Внедренные программистом процедуры проверки регистрации. Т.е. изменяется кусок программы, в которые добавляется к примеру, пользователь, зашитый в исполняемый код.

5 Переполнение буфера

Практически все операционные системы написаны на языке С, программы на котором компилируются гораздо эффективнее, чем другие языки. Однако ни один из компиляторов С не проверяет границы массива – за этим должен следить программист. Пример программного кода представлен в листинге 7.

```
int i;
BYTE buf[10000];
i=20000;
buf[i]=0;
```

Листинг 7 – Пример обращение к памяти за границей массива

В результате выполнения программы изменяется байт совершенно в другом месте. Это свойство языка С позволяет произвести атаку следующего типа (Рисунок 60).

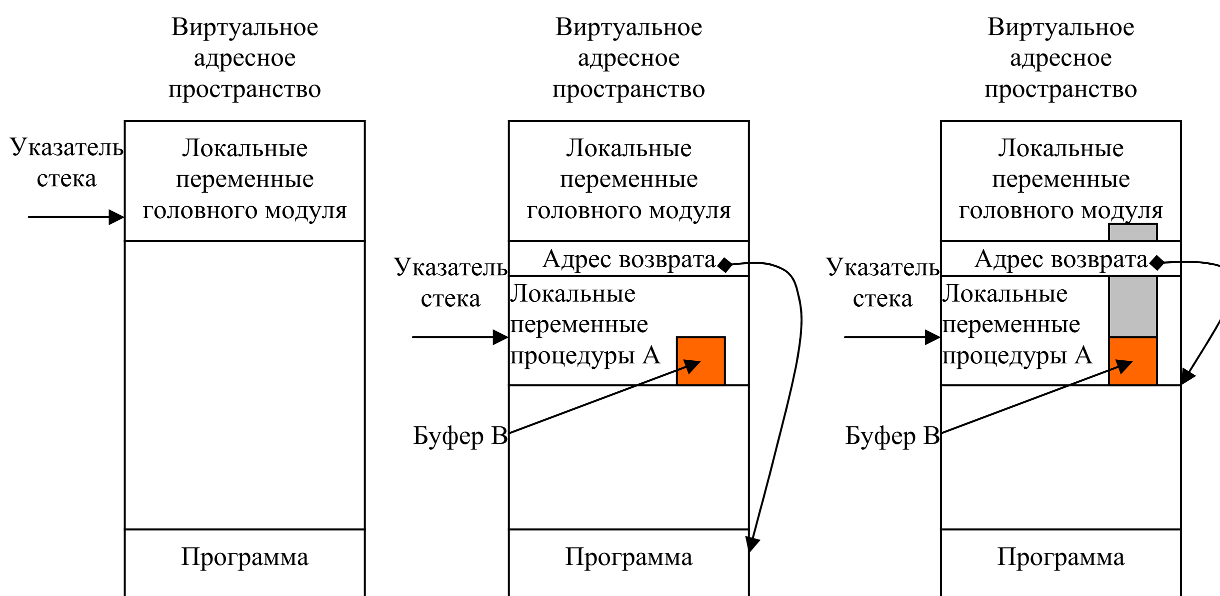

Рисунок 60 – Работает головная программа (а); вызывается процедура ***А*** (б); переполнение буфера (в)

На рисунке 60, а показана работающая программа со своими переменными в стеке. В некоторый момент она вызывает процедуру ***А*** (Рисунок 60, б). В стек помещается адрес возврата и управление передается процедуре ***А***. Для каких-либо целей в процедуре зарезервирован буфер ***В*** (например, для имени файла), однако пользователь вводит буфер намного больше. И если процедура не проверяет размер, то адрес возврата в программу может оказаться другим. Если предположить, что взломщик знает размер стека, то он может сделать буфер не из случайных байтов, а

из команд, а адрес возврата точно подгадать так, как показано на рисунке 56, в. В результате выполнить свой программный код при отсутствии на это прав. Защита от атак подобного рода заключается в проверке длины всех поставляемых пользователем строк перед копированием в буфер.

7.4 Атаки операционной системы снаружи

В связи с распространением Интернет и локальных сетей всё большую угрозу представляют атаки операционной системы с удалённых компьютеров. Почти во всех случаях атака состоит из того, что по сети передаётся компьютерная программа, при выполнении которой атакуемой машине наносится ущерб.

Особое внимание здесь стоит уделять вирусам. Вирус – это программа, которая может размножаться, присоединяя свой код к другой программе. Возможные сценарии нанесения ущерба вирусом:

- безвредные действия, вроде вывода сообщений на экран, изображений, воспроизведения звука и т.п.;
- порча информации, возможно шифрования файлов с целью шантажа;
- атака с целью отказа в обслуживании. Использование вирусом всех ресурсов компьютера, например, процессорного времени, заполнения жесткого диска;
- порча аппаратного обеспечения компьютера.

Для распространения вируса, автор обычно вставляет его в какую-нибудь пиратскую копию программы и выкладывает её в Интернет. После чего пользователи начинают загружать программу на свой компьютер, где вирус размножается и выполняет действия, ради которых он и писался. На настоящий момент выделяют 7 основных видов вирусов.

1 *Вирусы-компаньоны.* Эти вирусы не заражают программу, а запускаются вместо какой-либо программы. Например, в системе MS-DOS или в командной строке Windows (или в пункте Пуск→Выполнить) пользователь вводит команду prog. При этом операционная система ищет сначала prog.com, а если его нет, то prog.exe. Поэтому автор вируса может назвать программу схожую с известной exe, но с расширением com и запустить его первым. Закончив выполняться, вирус запускает оригинал.

2 *Вирусы, заражающие исполняемые файлы*. Самый простой вирус записывает себя поверх исполняемой программы, поэтому их называют перезаписывающими. Однако их легко обнаружить при записке программы. Паразитическими вирусами называются вирусы прицепляющиеся к программам, позволяя им нормально выполняться, что представлено на рисунках 61, б и в.

Полостные вирусы, представленные на рисунке 61, г используют тот принцип, что пытаются записать себя целиком в свободные участки исполняемого файла.

3 *Резидентные вирусы.* Если первые два типа вируса по выполнении собственных задач завершают существование в памяти, то резидентные всегда находятся в памяти. Типичный резидентный вирус перехватывает один из векторов прерывания (например, от устройства ввода-вывода или системного вызова), сохраняет старое значение в своей переменной и подменяет его адресом своей

процедуры. При каждом срабатывании прерывания вирус инфицирует программы.

4 *Вирусы, поражающие загрузочный сектор.* Вирус сначала копирует исходное содержимое загрузочного сектора, чтобы иметь возможность загружать операционную систему. При загрузке компьютера вирус копирует себя в оперативную память, а после загружает операционную систему, оставаясь при этом резидентным в памяти. Вирус перехватывает себе все векторы прерываний (от принтера, диска, часов и т.д.) и работает через них, контролируя все системные вызовы.

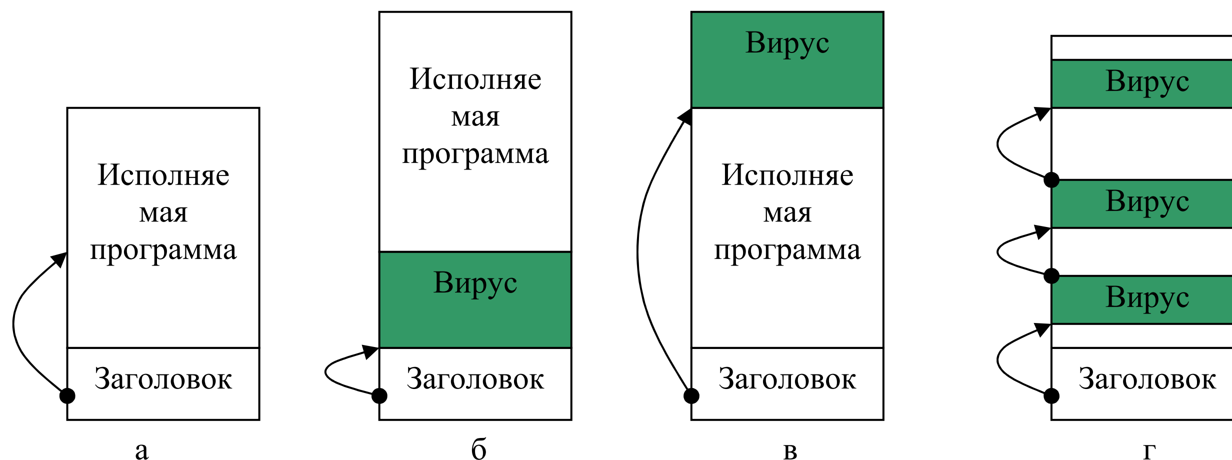

Рисунок 61 – Исполняемый файл (а); с вирусом в начале (б); с вирусом в конце (в); с вирусом, распределенным по свободным участкам программы (г)

5 *Вирусы драйверов устройств.* Вирус в этом случае инфицирует драйверы устройств, получая возможность перехватывать вектор системных прерываний.

6 *Макровирусы.* Макросы в Word и Excel также могут использоваться злоумышленниками для атаки операционных систем. Открытие документа с макросом вызывает выполнение его. А макрос может заражать другие документы Word, удалять файлы и т.д.

7 *Вирусы, заражающие исходные тексты программ.* Эти вирусы работают с исходными текстами программ, что делает их переносимыми для различных платформ.

Следует отличать вирусы от Интернет-червей, первые размножаются при запуске пользователем, а вторые используют дыры в операционной системе для самостоятельного проникновения и размножения.

Для борьбы с вирусами используются антивирусные программы, такие как DrWeb (http://www.drweb.com), Антивирус Касперского (http://www.kaspersky.ru) и многие другие, а также ряд научных трудов, например [4].

Однако одними вирусами атаки снаружи операционной системы не ограничиваются [6, 7]. Помимо всевозможных «дыр» в защите конкретных операционных систем, сетевых устройств и файрволов существует ещё угроза атаки отказа в обслуживании или DOS-атака (Denial of Service), которая в основном используется для атаки на Интернет-сервера. Атака такого рода препятствует или полностью блокирует ответы законным пользователям. Существует несколько типов DOS-атак.

- *Захват полосы пропускания.* Атакующий занимает всю производительность сети, например, переправляя трафик с других сайтов.
- *Истощение ресурсов.* Направлена на системные ресурсы – процессорного времени, памяти и т.п. Что приводит к краху операционной системы.
- *Ошибки программирования.* Они заключаются в неспособности приложения, операционной системы или логической микросхемы обрабатывать исключительные ситуации. Это достигается передачей несанкционированных данных.
- *Маршрутизация и атака DNS.* Основываются на манипуляции записями таблицы маршрутизации, что приводит к отказу в обслуживании Легитимных систем или сетей.

Механизмам атаки и защиты операционных систем посвящен ряд трудов, в которых рассмотрены как конкретные системы, так и принципы в целом.

Контрольные вопросы по разделу

1 Перечислите основные задачи и угрозы безопасности.

2 Что такое аутентификация? Чем аутентификация отличается от идентификации?

3 Каковы недостатки аутентификации по многоразовому паролю?

4 В чём заключается вариант использования защиты по паролю «отклик-отзыв»?

5 Чем различаются атаки изнутри и снаружи операционной системы?

6 Перечислите основные атаки изнутри операционной системы.

7 Поясните, как работает атака с использованием переполнения буфера. Почему данная атака возможна?

8 Какой ущерб могут причинить вирусы? Перечислите существующие виды вирусов?

9 Чем вирусы отличаются от Интернет-червей?

10 Что такое DOS-атака? Какие существуют типы DOS-атак?

8 Обзор современных операционных систем

8.1 Операционная система Windows 2000

Windows 2000 является очень удачной операционной системой, которая во многих местах используется и по сей день. Начиная с этой версии Windows, были совмещены две технологии, развивавшиеся ранее параллельно, представленные в операционных системах Windows 98 и Windows NT 4.0. С этой версии в Windows, отсутствует режим MS-DOS. После выхода Windows 2000 стало традицией распространять средства для разработчиков Software Development Kit (SDK) и Driver Development Kit (DDK), которые можно найти по адресу msdn.microsoft.com. Windows 2000 представляет собой чрезвычайно сложную систему, состоящую из около 30 млн. строк на С. А дальнейшие ОС: XP, 2003, Vista занимают ещё больший объём.

В Windows 2000 есть свой набор системных вызовов, которые она может выполнять. Однако взамен их использования для программистов корпорация Microsoft предоставляет набор функциональных вызовов Win32 API (Application Programming Interface). Эти вызовы опубликованы и полностью документированы. Они представляют собой библиотечные процедуры, которые либо обращаются к системным вызовам, чтобы выполнить некоторую работу, либо выполняют работу прямо в пространстве пользователя. При этом сохраняется поддержка старых API функций с предыдущих версий операционной системы (Рисунок 62).

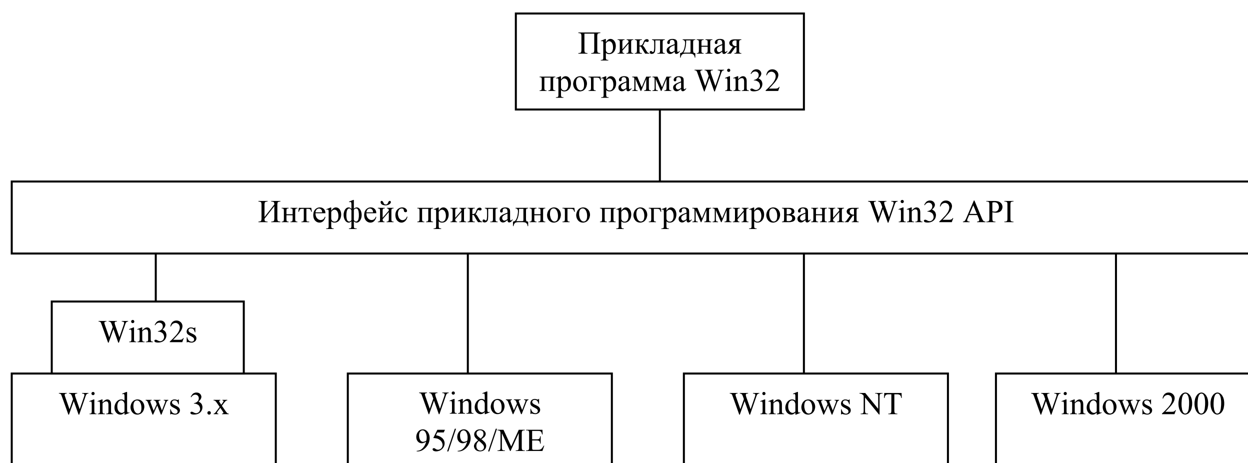

Рисунок 62 – Интерфейс Win32 API, Win32s – дополнительная библиотека, преобразующая подмножество 32-разрядных вызовов в 16-разрядные

В отличие от других операционных систем, например UNIX, Win32 API предоставляет всеобъемлющий интерфейс, позволяющий выполнить одно и тоже действие несколькими способами. Так в UNIX все системные вызовы формируют минимальный интерфейс, ни один из них нельзя удалить.

Важным понятием в операционной системе Windows 2000 является реестр – центральная база данных, в которой находится почти вся информация, необходимая для загрузки и конфигурирования системы, настройки её под конкретного пользователя. Хотя реестр является одной из наиболее запутанных частей Windows,

его идея очень проста. Он состоит из набора каталогов, каждый из которых содержит либо подкаталоги, либо записи, по своей структуре напоминая файловую систему. Информация разбита по корневым каталогам, называемым ключами. Не вдаваясь в подробности, отметим предназначение корневых ключей. Для просмотра реестра можно пользоваться утилитой regedit.

HKEY_LOCAL_MACHINE – содержит всю информацию о локальной системе: описания аппаратуры, о драйверах, именах пользователей и паролях, политике безопасности, настройки производителей программного обеспечения для своих программ, информацию о загрузке системы.

HKEY_USERS – содержит профили для каждого пользователя.

HKEY_CLASSES_ROOT – содержит настройки для управления объектами COM (Component Object Model – модель компонентных объектов), а также занимается установкой соответствий между расширениями файлов и программами.

HKEY_CURRENT_CONFIG – представляет собой ссылку на подключ, содержащий информацию о текущей конфигурации аппаратного обеспечения.

HKEY_CURRENT_USER – указывает на настройки текущего пользователя.

HKEY_PERFORMANCE_DATA – данный ключ не виден в утилите просмотра regedit. Ключ предоставляет окно в операционную систему. Сама система содержит сотни счетчиков для мониторинга производительности системы. К таким счетчикам можно получить доступ через этот ключ реестра.

8.1.1 Структура Windows 2000

Операционная система Windows 2000 состоит из двух основных частей: самой операционной системы, работающей в режиме ядра, и подсистем окружения, работающих в режиме пользователя. Ядро является традиционным ядром в том смысле, что оно управляет процессами, памятью, файловой системой и т. д. Подсистемы окружения представляют собой нечто необычное, так как они являются отдельными процессами, помогающими пользователю выполнять определенные системные функции.

Система разделена на несколько уровней, каждый из которых пользуется службами лежащего ниже уровня. Эта структура проиллюстрирована на рисунке 63 (квадратики, помеченные символом «D», обозначают драйверы устройств, а сервисные процессы являются системными демонами). Один из уровней разделен горизонтально на множество модулей. У каждого модуля есть определенная функция, а также четко определенный интерфейс для взаимодействия с другими модулями.

Кратко опишем данную схему.

HAL (Hardware Abstraction Layer) предназначена для скрытия аппаратных различий. Работа уровня HAL заключается в том, чтобы предоставлять всей остальной системе абстрактные аппаратные устройства, свободные от индивидуальных отличительных особенностей, которыми так богато реальное аппаратное обеспечение. Эти устройства представляются в виде машинно-независимых служб (процедурных вызовов и макросов), которые могут использоваться остальной операционной системой и драйверами. Поскольку

драйверы и ядро пользуются службами HAL (идентичными на всех операционных системах Windows 2000, независимо от аппаратного обеспечения) и не обращаются напрямую к устройствам, требуется значительно меньше изменений для их переноса на другую платформу. Перенос самого уровня HAL довольно прост, так как весь машинно-зависимый код сконцентрирован в одном месте, а цель переделки четко определена, то есть заключается в реализации всех служб уровня HAL.

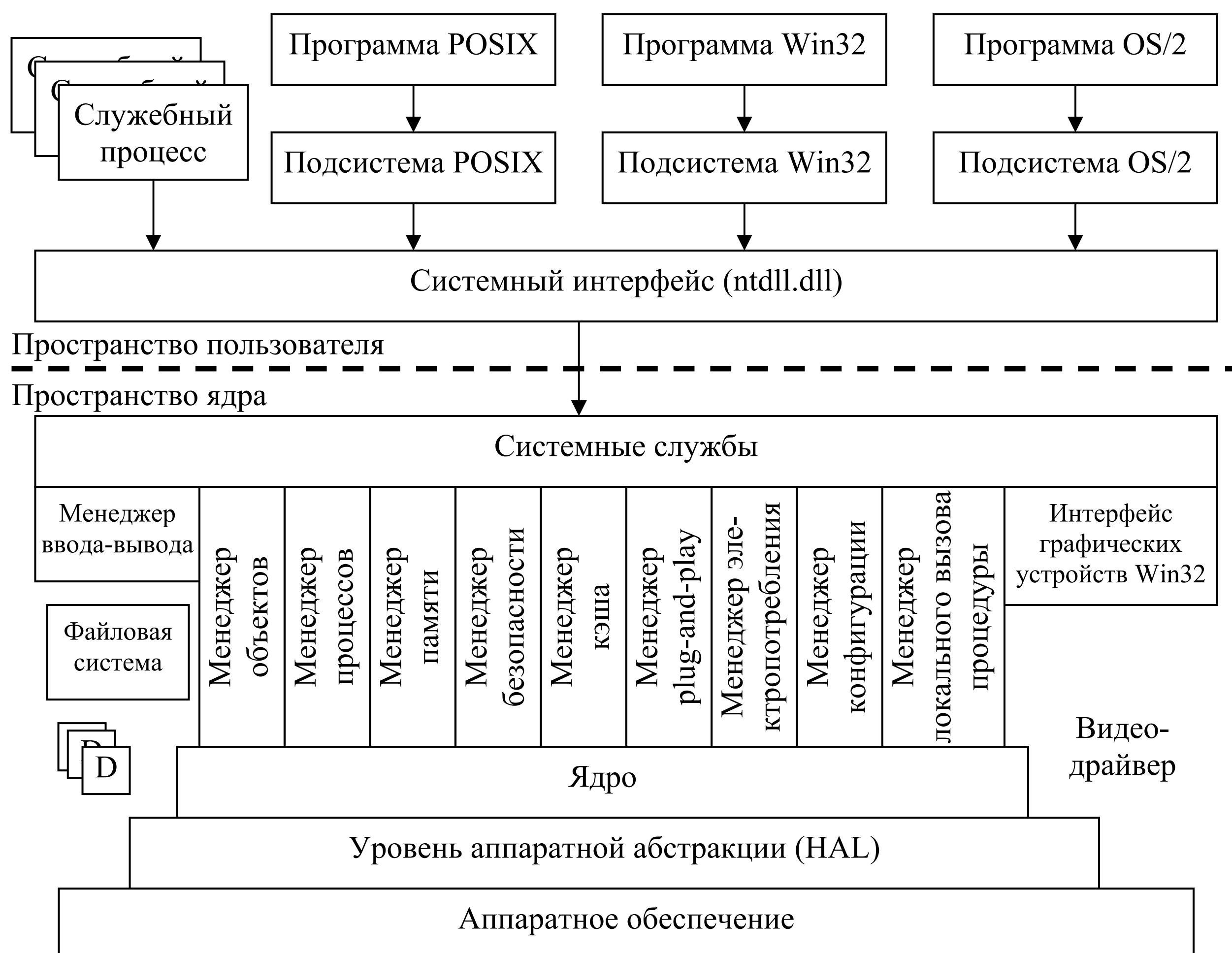

Рисунок 63 – Структура Windows 2000 (слегка упрощенная)

К службам уровня HAL относятся: доступ к регистрам устройств, адресация к устройствам, независящим от шины, обработка прерываний и возврат из прерываний, операции DMA (Direct Memory Access – прямой доступ к памяти), управление таймерами, часами реального времени, спин-блокировками нижнего уровня и синхронизация многопроцессорных конфигураций, интерфейс с BIOS и доступ к CMOS-памяти. Уровень HAL не предоставляет абстракций или служб для специфических устройств ввода-вывода – клавиатур, мышей или дисков, а также блоков управления памятью MMU.

Поскольку уровень HAL является в большой степени машинно-зависимым, он должен в совершенстве соответствовать системе, на которой установлен, поэтому набор различных уровней HAL поставляется на компакт-диске Windows 2000. Во время установки системы из них выбирается подходящий уровень и копируется на жесткий диск в системный каталог \winnt\system32 в виде файла hal.dll. При всех

последующих загрузках операционной системы используется эта версия уровня HAL. Если удалить этот файл, то система загрузиться не сможет.

Хотя эффективность уровня HAL является довольно высокой, для мультимедийных приложений ее может быть недостаточно. По этой причине корпорация Microsoft также производит пакет программного обеспечения, называемый DirectX, расширяющий функциональность уровня HAL дополнительными процедурами и предоставляющий пользовательским процессам прямой доступ к аппаратному обеспечению.

Над уровнем аппаратных абстракций располагается уровень, содержащий то, что корпорация Microsoft называет ядром, а также драйверы устройств. Назначение ядра заключается в том, чтобы сделать всю остальную часть операционной системы независимой от аппаратуры и, таким образом, легко переносимой на другие платформы. Оно начинается там, где заканчивается уровень HAL. Ядро получает доступ к аппаратуре через уровень HAL.

Ядро предоставляет абстрактную модель аппаратуры более высокого уровня, управляет переключениями потоков, предоставляет низкоуровневую поддержку двум классам объектов – управляющим объектам и объектам диспетчеризации. Эти объекты не являются объектами, к которым пользовательские процессы получают дескрипторы, но представляют собой внутренние объекты, на основе которых исполняющая система строит объекты пользователя.

Управляющие объекты – это объекты, управляющие системой, включая примитивные объекты процессов, объекты прерываний и два несколько странных объекта, называемых DPC и APC. Объект DPC (Deferred Procedure Call – отложенный вызов процедуры) используется, чтобы отделить часть процедуры обработки прерываний, для которой время является критичным, от той ее части, для которой время некритично. Объект APC (Asynchronous Procedure Call – асинхронный вызов процедуры) похож на отложенный вызов процедуры DPC, но отличается тем, что асинхронный вызов процедуры выполняется в контексте определенного процесса. Еще один тип объектов ядра – объекты диспетчеризации. К ним относятся семафоры, мьютексы, события, таймеры и другие объекты, изменения состояния которых могут ждать потоки.

Над ядром и драйверами устройств располагается верхняя часть операционной системы, называемая исполняющей системой (а также иногда супервизором или диспетчером). Исполняющая система состоит из 10 компонентов, каждый из которых представляет собой просто набор процедур, работающих вместе для выполнения некоторой задачи. Между отдельными компонентами нет жестких границ, и различные авторы, описывающие исполняющую систему, могут даже по-разному группировать составляющие ее процедуры в компоненты. Следует заметить, что компоненты одного уровня могут вызывать друг друга, и на практике они этим довольно активно занимаются.

Менеджер объектов управляет всеми объектами, известными операционной системе. К ним относятся процессы, потоки, файлы, каталоги, семафоры, устройства ввода-вывода, таймеры и многое другое. *Менеджер ввода-вывода* формирует каркас для управления устройствами ввода-вывода и предоставляет общие службы ввода-вывода. Он предоставляет остальной части системы независимый от устройств ввод-

вывод, вызывая для выполнения физического ввода-вывода соответствующий драйвер. *Менеджер процессов* управляет процессами и потоками, включая их создание и завершение. *Менеджер памяти* реализует архитектуру виртуальной памяти со страничной подкачкой по требованию операционной системы Windows 2000. Он управляет преобразованием виртуальных страниц в физические страничные блоки. *Менеджер безопасности* приводит в исполнение сложный механизм безопасности Windows 2000, удовлетворяющий требованиям класса C2 Оранжевой книги Министерства обороны США. *Менеджер кэша* хранит в памяти блоки диска, которые использовались в последнее время, чтобы ускорить доступ к ним в случае, если они понадобятся вновь. Его работа состоит в том, чтобы определить, какие блоки понадобятся снова, а какие нет. *Менеджер plug-and-play* получает все уведомления об установленных новых устройствах. *Менеджер энергопотребления* управляет потреблением электроэнергии. Он выключает монитор и диски, если к ним не было обращений в течение определенного интервала времени. *Менеджер конфигурации* отвечает за состояние реестра. *Менеджер вызова локальной процедуры* обеспечивает высокоэффективное взаимодействие между процессами и их подсистемами.

Исполняющий модуль Win32 GDI обрабатывает определенные системные вызовы (но не все). Изначально он располагался в пространстве пользователя, но в версии NT 4.0 для увеличения производительности был перенесен в пространство ядра. *Интерфейс графических устройств GDI* (Graphic Device Interface) занимается управлением графическими изображениями для монитора и принтеров. Он предоставляет системные вызовы, позволяющие пользовательским программам выводить данные на монитор и принтеры независящим от устройств способом.

Над исполняющей системой размещается тонкий уровень, называемый *системными службами*. Его функции заключаются в предоставлении интерфейса к исполняющей системе. Он принимает настоящие системные вызовы Windows 2000 и вызывает другие части исполняющей системы для их выполнения.

При загрузке операционная система Windows 2000 загружается в память как набор файлов. Основная часть операционной системы, состоящая из ядра и исполняющей системы, хранится в файле ntoskrnl.exe. Уровень HAL представляет собой библиотеку общего доступа, расположенную в отдельном файле hal.dll. Интерфейс Win32 и интерфейс графических устройств хранятся вместе в третьем файле, win32k.sys. Наконец, загружается множество драйверов устройств. У большинства из них расширение .sys.

8.1.2 Реализация интерфейса Win32

Операционной системой Windows 2000 поддерживаются три различных документированных интерфейса прикладного программирования API: Win32, POSIX и OS/2. У каждого из этих интерфейсов есть список библиотечных вызовов, которые могут использовать программисты. Работа библиотек DLL (Dynamic Link Library – динамически подключаемая библиотека) и подсистем окружения заключается в том, чтобы реализовать функциональные возможности опубликованного интерфейса, тем самым скрывая истинный интерфейс системных

вызовов от прикладных программ.

Рассмотрим способ реализации этих интерфейсов на примере Win32. Программа, пользующаяся интерфейсом Win32, как правило, состоит из большого количества обращений к функциям Win32 API. Один из возможных способов реализации заключается в статическом связывании каждой программы, использующей интерфейс Win32, со всеми библиотечными процедурами, которыми она пользуется. При таком подходе каждая двоичная программа будет содержать копию всех используемых ею процедур в своем исполняемом двоичном файле. Недостаток такого подхода заключается в том, что при этом расходуется много памяти, если пользователь одновременно откроет несколько программ, использующих одни и те же библиотечные процедуры.

Чтобы избежать подобной проблемы, все версии Windows поддерживают динамические библиотеки DLL. Каждая динамическая библиотека содержит набор тесно связанных библиотечных процедур и все их структуры данных в одном файле, как правило (но не всегда), с расширением *.dll.

Каждый пользовательский процесс, как правило, связан с несколькими динамическими библиотеками, совместно реализующими интерфейс Win32. Чтобы обратиться к вызову API, вызывается одна из процедур в DLL (шаг 1 на рисунке 64). Дальнейшие действия зависят от вызова Win32 API. Различные вызовы реализованы по-разному.

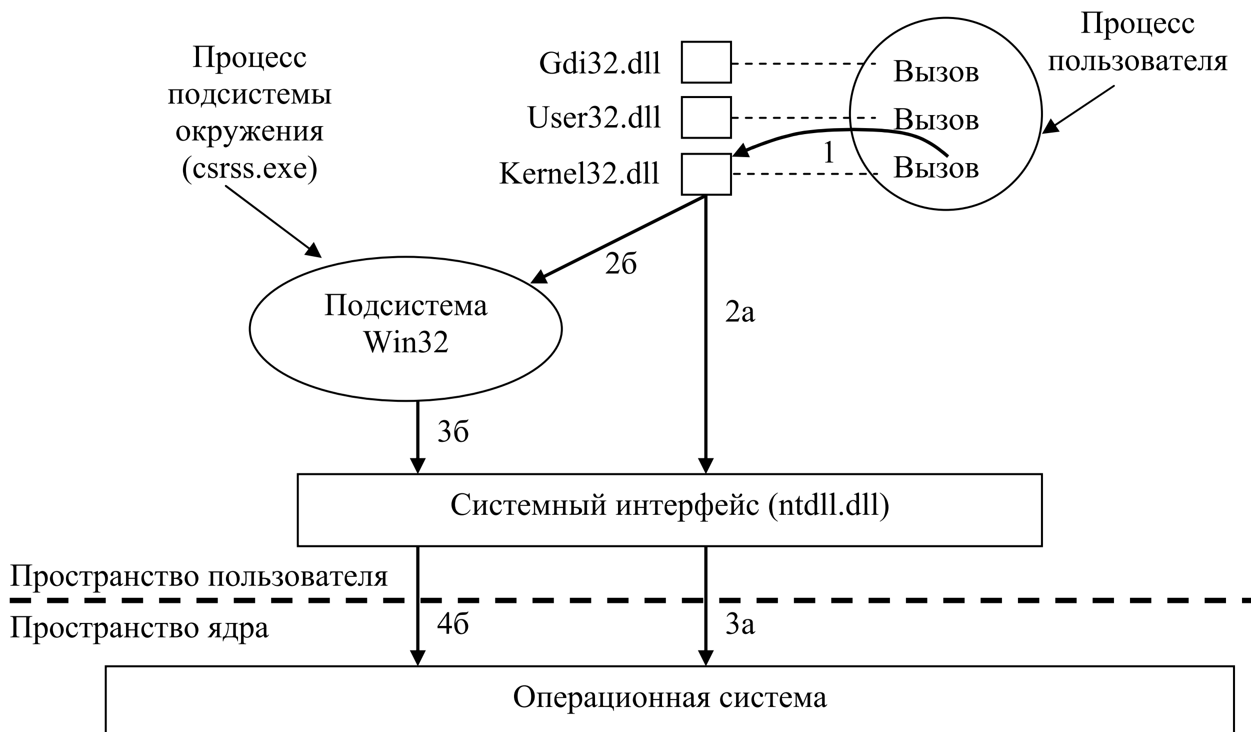

Рисунок 64 – Различные маршруты выполнения вызовов Win32 API

В некоторых случаях динамические библиотеки обращаются к другой динамической библиотеке (ntdll.dll) которая, в свою очередь, обращается к ядру операционной системы. Этот путь показан на рисунке как шаги 2а и 3а. Динамическая библиотека может также выполнить всю работу самостоятельно,

совсем не обращаясь к системным вызовам. Для других вызовов Win32 API выбирается другой маршрут, а именно: сначала процессу подсистемы Win32 (csrss.exe) посылается сообщение, выполняющее некоторую работу, и обращающееся к системному вызову (шаги 2б, 3б и 4б).

В первой версии Windows NT практически все вызовы Win32 API выбирали маршрут 2б, 3б, 4б, так как большая часть операционной системы (например, графика) была помещена в пространство пользователя. Однако, начиная с версии NT 4.0, для увеличения производительности большая часть кода была перенесена в ядро (в драйвер Win32/GDI). В Windows 2000 только небольшое количество вызовов Win32 API, например вызовы для создания процесса или потока, идут по длинному пути. Остальные вызовы выполняются напрямую, минуя подсистему окружения Win32.

Хотя интерфейс процессов Win32 является наиболее важным, в операционной системе Windows 2000 существует еще два интерфейса: POSIX и OS/2. Среда POSIX предоставляет минимальную поддержку для приложений UNIX. Она поддерживает практически только функции, описанные в стандарте P 1003.1. Этим интерфейсом, например, не поддерживаются потоки, работа с окнами или сетью. Перенос любой реальной программы из системы UNIX в Windows 2000 при помощи этой подсистемы практически невозможен. Этот интерфейс был включен только потому, что некоторые министерства правительства США требовали, чтобы операционные системы для правительственных компьютеров были совместимы со стандартом P1003.1. Эта подсистема не является самодостаточной и пользуется вызовами подсистемы Win32 для большей части своей работы, но не предоставляя пользовательским программам полного интерфейса Win32 (если бы это было сделано корпорацией Microsoft, то подсистемой можно было бы пользоваться, причем для этого не потребовалось бы никаких специальных усилий).

Функциональность подсистемы OS/2 ограничена практически в той же степени, что и функциональность подсистемы POSIX. Подсистема OS/2 также не поддерживает графические приложения. На практике она тоже полностью бесполезна. Таким образом, оригинальная идея наличия интерфейсов нескольких операционных систем, реализованных различными процессами в пространстве пользователя, окончилась ничем. Осталась лишь полная реализация интерфейса Win32 в режиме ядра.

8.1.3 Эмуляция MS-DOS

Одна из задач проектирования системы Windows 2000 была унаследована от NT: постараться поддержать по возможности большое (в разумных пределах) количество программ для MS-DOS. Эта цель принципиально отличалась от задачи, поставленной перед создателями системы Windows 98: в ней должны были работать все старые программы для MS-DOS (от себя добавим: неважно, насколько плохо они работали).

Операционная система Windows 2000 запускает старые программы в полностью защищенном окружении. Когда запускается программа, написанная для системы MS-DOS, запускается нормальный процесс Win32, в который загружается

эмулятор MS-DOS ntvdm (NT Virtual DOS Machine - виртуальная машина DOS для NT), сканирующий программу MS-DOS и выполняющий ее системные вызовы (Рисунок 65).

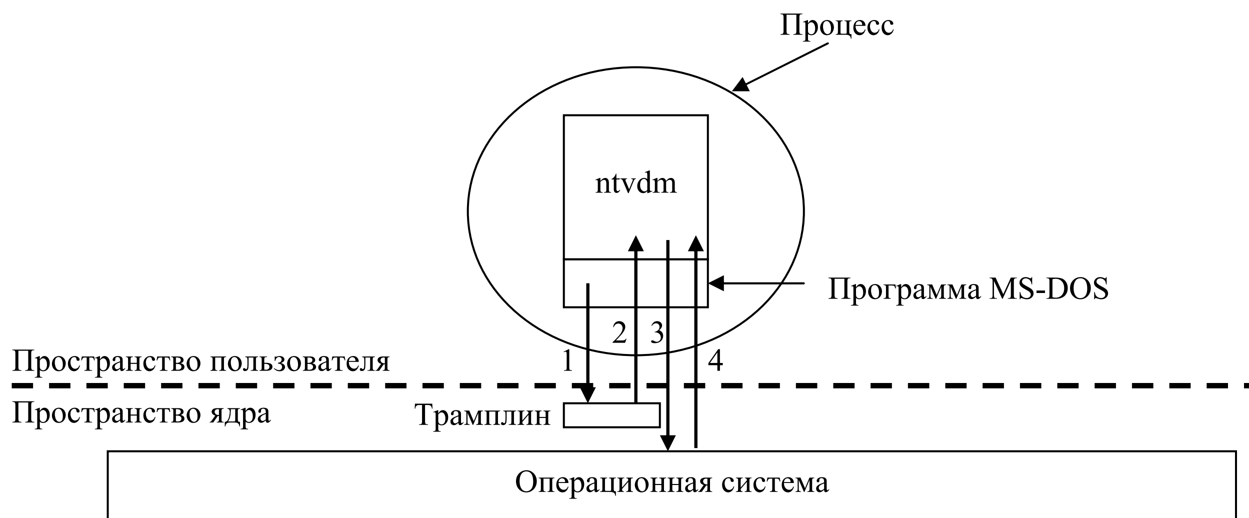

Рисунок 65 – Запуск старых программ для MS-DOS в системе Windows 2000

Когда программа MS-DOS выполняет обычные команды процессора, она может работать на голой аппаратуре, так как процессор Pentium полностью поддерживает наборы команд процессоров Intel 8088 и Intel 80286. Самое интересное начинается, когда программа MS-DOS собирается выполнить операцию ввода-вывода или обратиться к операционной системе. Корректно написанная программа просто выполняет системный вызов. Рассчитывая на корректное поведение, эмулятор ntvdm инструктирует систему Windows 2000 перенаправлять все системные вызовы ему. В результате системный вызов просто «отскакивает» от операционной системы и перехватывается эмулятором (шаги 1 и 2). Такой метод иногда называют использованием трамплина.

Получив управление, эмулятор определяет, что пытается сделать программа, и обращается к вызову Win32 для выполнения требуемой работы (шаги 3 и 4). Пока программа ведет себя корректно и лишь обращается к легальным системным вызовам MS-DOS, этот метод прекрасно работает. Неприятность в том, что некоторые старые программы, написанные для работы в системе MS-DOS, обходят операционную систему и напрямую обращаются к видеопамяти, напрямую читают порты клавиатуры и т. д., то есть выполняют действия, недопустимые в защищенной среде. Если такое некорректное поведение программы приводит к прерываниям, есть надежда, что эмулятор сумеет определить, чего хочет программа, и сможет эмулировать это действие. Если же определить намерения программы не удастся, программа просто уничтожается, так как задача стопроцентной эмуляции старых операционных систем никогда не ставилась при проектировании Windows 2000.

8.2 Архитектура UNIX-образных операционных систем

Операционная система UNIX представляет собой интерактивную систему, разработанную для одновременной поддержки нескольких процессов и нескольких

пользователей. Она была разработана программистами и для программистов, чтобы использовать ее в окружении, в котором большинство пользователей являются относительно опытными и занимаются проектами (часто довольно сложными) разработки программного обеспечения.

Операционную систему UNIX можно рассматривать в виде пирамиды (Рисунок 66). У основания пирамиды располагается аппаратное обеспечение, состоящее из центрального процессора, памяти, дисков, терминалов и других устройств. На голом «железе» работает операционная система UNIX. Ее функция заключается в управлении аппаратным обеспечением и предоставлении всем программам интерфейса системных вызовов. Эти системные вызовы позволяют программам создавать процессы, файлы и прочие ресурсы, а также управлять ими.

Программы обращаются к системным вызовам, помещая аргументы в регистры центрального процессора (или иногда в стек) и выполняя команду эмулированного прерывания для переключения из пользовательского режима в режим ядра и передачи управления операционной системе UNIX. Поскольку на языке С невозможно написать команду эмулированного прерывания, этим занимаются библиотечные функции, по одной на системный вызов. Эти процедуры написаны на ассемблере, но они могут вызываться из программ, написанных на С. Каждая такая процедура помещает аргументы в нужное место и выполняет команду эмулированного прерывания TRAP. Таким образом, чтобы обратиться к системному вызову read, программа на С должна вызвать библиотечную процедуру read. В стандарте POSIX определен именно интерфейс библиотечных функций, а не интерфейс системных вызовов. Другими словами, стандарт POSIX определяет библиотечные процедуры, соответствующие системным вызовам, их параметры, что они должны делать и какой результат возвращать. В стандарте даже не упоминаются фактические системные вызовы.

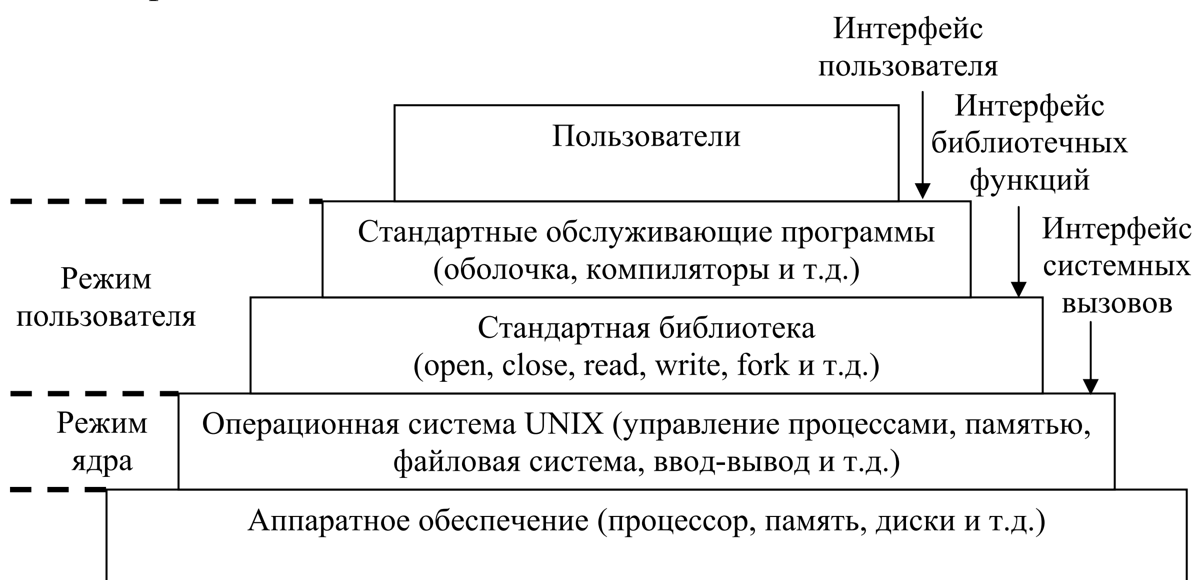

Рисунок 66 – Уровни операционной системы UNIX

Помимо операционной системы и библиотеки системных вызовов, все версии UNIX содержат большое количество стандартных программ, некоторые из них описываются стандартом POSIX 1003.2, тогда как другие могут различаться в

разных версиях системы UNIX. К этим программам относятся командный процессор (оболочка), компиляторы, редакторы, программы обработки текста и утилиты для работы с файлами. Именно эти программы и запускаются пользователем с терминала.

Можно говорить о трех интерфейсах в операционной системе UNIX: интерфейсе системных вызовов, интерфейсе библиотечных функций и интерфейсе, образованным набором стандартных обслуживающих программ. Хотя именно последний интерфейс большинство пользователей считает системой UNIX, в действительности он не имеет практически никакого отношения к самой операционной системе и легко может быть заменен.

Обзор структуры ядра системы UNIX представляет собой довольно непростое дело, так как существует множество различных версий этой системы. Однако, хотя диаграмма на рисунке 67 описывает UNIX 4.4BSD [2], она также применима ко многим другим версиям, возможно, с небольшими изменениями в тех или иных местах.

Нижний уровень ядра состоит из драйверов устройств и процедуры диспетчеризации процессов. Все драйверы системы UNIX делятся на два класса: драйверы символьных устройств и драйверы блочных устройств. Технически сетевые устройства представляют собой символьные устройства, но они обрабатываются по-иному, поэтому их, вероятно, правильнее выделить в отдельных класс, как это и было сделано на схеме. Диспетчеризация процессов производится при возникновении прерывания. При этом низкоуровневая программа останавливает выполнение работающего процесса, сохраняет его состояние в таблице процессов ядра и запускает соответствующий драйвер. Кроме того, диспетчеризация процессов производится также, когда ядро завершает свою работу и пора снова запустить процесс пользователя.

<table>
<tr><td colspan="5">Системные вызовы</td><td colspan="3">Аппаратные и эмулированные прерывания</td></tr>
<tr><td colspan="2">Управление терминалом</td><td>Сокеты</td><td>Именование файлов</td><td>Отображение адресов</td><td>Страничные прерывания</td><td rowspan="2">Обработка сигналов</td><td rowspan="2">Создание и завершение процессов</td></tr>
<tr><td rowspan="2">Необработанный телетайп</td><td>Обработанный телетайп</td><td>Сетевые протоколы</td><td>Файловые системы</td><td colspan="2">Виртуальная память</td></tr>
<tr><td>Дисциплины линии связи</td><td>Маршрутизация</td><td>Буферный кэш</td><td colspan="2">Страничный кэш</td><td colspan="2">Планирование процесса</td></tr>
<tr><td colspan="2">Символьные устройства</td><td>Драйверы сетевых устройств</td><td colspan="3">Драйверы дисковых устройств</td><td colspan="2">Диспетчеризация процессов</td></tr>
<tr><td colspan="8">Аппаратура</td></tr>
</table>

Рисунок 67 – Структура ядра операционной системы UNIX 4.4BSD

В более высоких уровнях программы отличаются в каждом из четырех

«столбцов» диаграммы. Слева располагаются символьные устройства. Они могут использоваться двумя способами. Некоторым программам, таким как текстовые редакторы vi и emacs, требуется каждая нажатая клавиша без какой-либо обработки. Для этого служит ввод-вывод с необработанного терминала (телетайпа). Другое программное обеспечение, например оболочка (sh), принимает на входе уже готовую текстовую строку, позволяя пользователю редактировать ее, пока не будет нажата клавиша ENTER. Такое программное обеспечение пользуется вводом с терминала в обработанном виде и дисциплинами линии связи.

Сетевое программное обеспечение часто бывает модульным, с поддержкой множества различных устройств и протоколов. Уровень выше сетевых драйверов выполняет своего рода функции маршрутизации, гарантируя, что правильный пакет направляется правильному устройству или блоку управления протоколами. Большинство систем UNIX содержат в своем ядре полноценный маршрутизатор Интернета, и хотя его производительность ниже, чем у аппаратного маршрутизатора, эта программа появилась раньше современных аппаратных маршрутизаторов. Над уровнем маршрутизации располагается стек протоколов, обязательно включая протоколы IP и TCP, но также иногда и некоторые дополнительные протоколы. Над сетевыми протоколами располагается интерфейс сокетов, позволяющий программам создавать сокеты для отдельных сетей и протоколов. Для использования сокетов пользовательские программы получают дескрипторы файлов.

Над дисковыми драйверами располагаются буферный кэш и страничный кэш файловой системы. В ранних системах UNIX буферный кэш представлял собой фиксированную область памяти, а остальная память использовалась для страниц пользователя. Во многих современных системах UNIX этой фиксированной границы уже не существует, и любая страница памяти может быть схвачена для выполнения любой задачи, в зависимости от того, что требуется в данный момент.

Над буферным кэшем располагаются файловые системы. Большинством систем UNIX поддерживаются несколько файловых систем, включая быструю файловую систему Беркли, журнальную файловую систему, а также различные виды файловых систем System V. Все эти файловые системы совместно используют общий буферный кэш. Выше файловых систем помещается именование файлов, управление каталогами, управление жесткими и символьными связями, а также другие свойства файловой системы, одинаковые для всех файловых систем.

Над страничным кэшем располагается система виртуальной памяти. В нем вся логика работы со страницами, например алгоритм замещения страниц. Поверх него находится программа отображения файлов на виртуальную память и высокоуровневая программа управления страничными прерываниями. Эта программа решает, что нужно делать при возникновении страничного прерывания. Сначала она проверяет допустимость обращения к памяти и, если все в порядке, определяет местонахождение требуемой страницы и то, как она может быть получена.

Последний столбец имеет отношение к управлению процессами. Над диспетчером располагается планировщик процессов, выбирающий процесс, который должен быть запущен следующим. Если потоками управляет ядро, то управление

потоками также помещается здесь, хотя в некоторых системах UNIX управление потоками вынесено в пространство пользователя. Над планировщиком расположена программа для обработки сигналов и отправки их в требуемом направлении, а также программа, занимающаяся созданием и завершением процессов.

Верхний уровень представляет собой интерфейс системы. Слева располагается интерфейс системных вызовов. Все системные вызовы поступают сюда и направляются одному из модулей низших уровней в зависимости от природы системного вызова. Правая часть верхнего уровня представляет собой вход для аппаратных и эмулированных прерываний, включая сигналы, страничные прерывания, разнообразные исключительные ситуации процессора и прерывания ввода-вывода.

8.3 Мультипроцессоры и мультипроцессорные операционные системы

Мультипроцессор с общей памятью (или просто мультипроцессор) представляет собой компьютерную систему, в которой два или более центральных процессора делят полный доступ к общей оперативной памяти. У всех мультипроцессоров каждый центральный процессор может адресоваться ко всей памяти. По характеру доступа они разделяются на два класса [10]:

- UMA (Uniform Memory Access – однородный доступ к памяти);
- NUMA (Non Uniform Memory Access – неоднородный доступ к памяти).

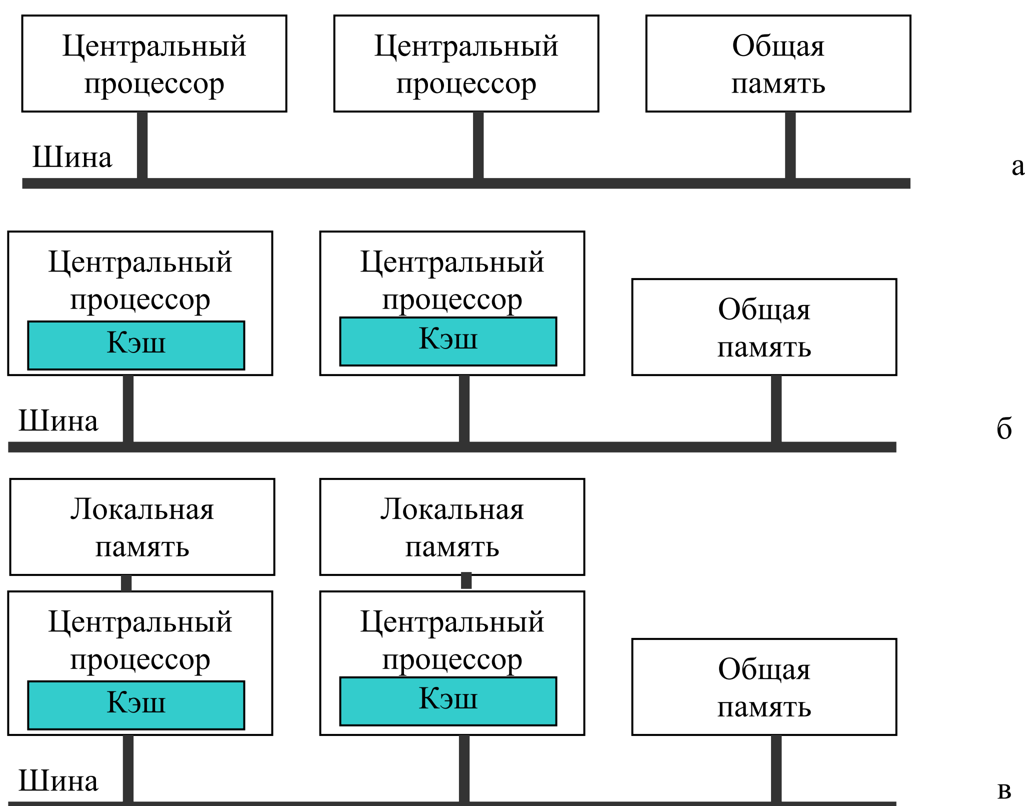

Рисунок 68 – Три варианта архитектуры мультипроцессоров с общей шиной: без кэша (а); с кэшем (б); с кэшем и собственной памятью (в)

UMA-мультипроцессоры обладают свойством считывания каждого слова данных с одинаковой скоростью. NUMA не обладают. Возможные архитектуры симметричных мультипроцессоров UMA с общей шиной представлены на рисунке 68.

Как видно из рисунка 68, а несколько центральных процессоров и несколько модулей памяти одновременно используют одну и ту же шину для общения друг с другом. Когда центральный процессор хочет прочитать слово в памяти, он сначала проверяет, свободна ли шина. Проблема данной архитектуры в том, что при большом количестве центральных процессоров (например, 32) шина будет занята, а производительность системы будет полностью ограничена пропускной способностью шины. Процессоры будут простаивать.

Решение этой проблемы состоит в том, чтобы добавить каждому центральному процессору кэш, как показано на рисунке 68, б. Данная схема частично разгружает шину. Ещё один вариант архитектуры представлен на рисунке 68, в. В этом случае имеется ещё и общая память, с которой процессор соединен по выделенной шине. Однако данную схему должен поддерживать компилятор.

Даже при оптимальном использовании кэша наличие всего одной общей шины ограничивает число UMA-мультипроцессоров. Чтобы преодолеть это ограничение требуется другая схема соединительной сети. Для этого можно использовать координатный коммутатор (Рисунок 69).

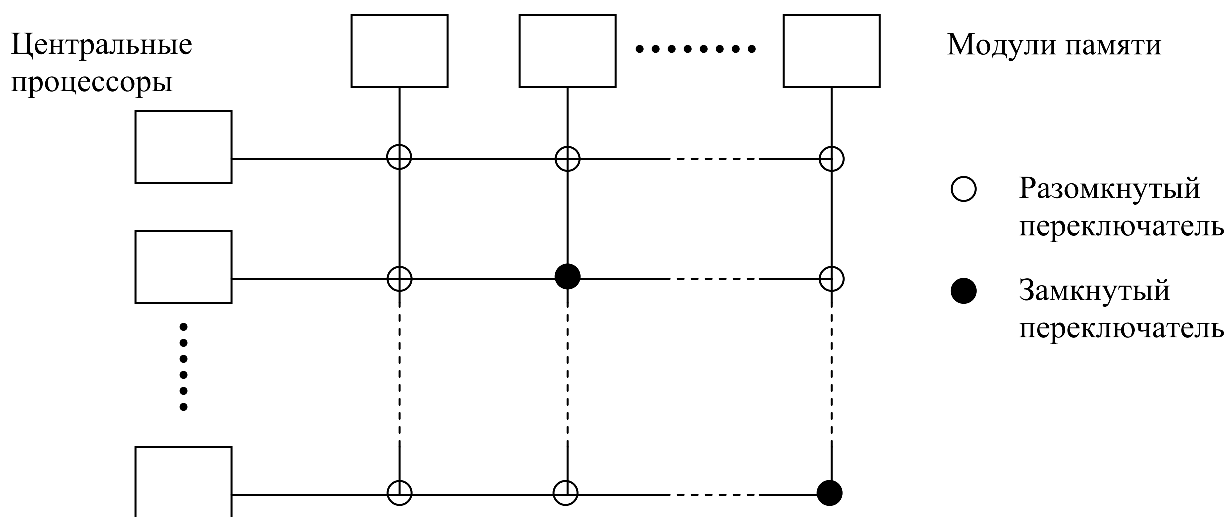

Рисунок 69 – Координатный коммутатор

На каждом пересечении линий располагается координатный переключатель. Достоинство этой схемы заключается в том, что она представляет собой неблокирующую сеть. Т.е. ни один центральный процессор не получает отказа соединения по причине занятости какого-либо переключателя. Недостаток – число переключателей растет пропорционально квадрату от числа процессоров. Поэтому используются многоступенчатые коммутаторные сети.

Для мультипроцессоров UMA с единственной общей шиной пределом является несколько десятков центральных процессоров, в то время как мультипроцессорам с координатным коммутатором или коммутирующей сетью

требуется большое количество дорогого аппаратного обеспечения. Чтобы создать мультипроцессор с числом центральных процессоров, превосходящих 100, обычно жертвуют одинаковым временем доступа ко всем модулям памяти. У машины NUMA есть три ключевые характеристики, которые, взятые вместе, отличают их от других мультипроцессоров:

- для всех центральных процессоров имеется единое адресное пространство;
- доступ к удаленным модулям памяти осуществляется при помощи специальных команд процессора;
- доступ к удаленным модулям памяти медленнее, чем к локальной памяти.

На мультипроцессорах возможны различные варианты организации операционной системы. Рассмотрим 3 из них.

1 *Каждому центральному процессору – свою операционную системы.* Простейший способ организации мультипроцессорных операционных систем состоит в том, чтобы статически разделить оперативную память по числу центральных процессоров и дать каждому центральному процессору свою собственную память с собственной копией системы. Процессоры работают как независимые компьютеры. Процессоры совместно используют код операционной системы, храня только индивидуальные копии данных (Рисунок 70).

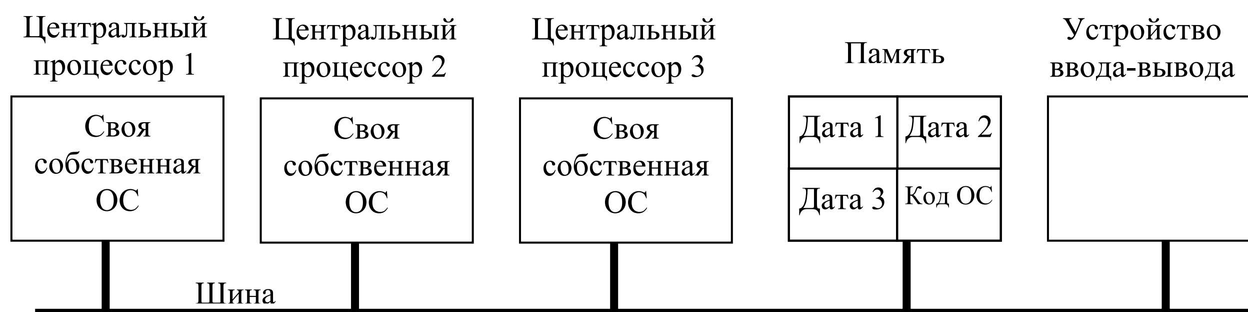

Рисунок 70 – Разделение памяти мультипроцессора между 3 центральными процессорами с общей копией кода операционной системы

Достоинства: простота реализации, совместное использование устройств ввода-вывода, возможное взаимодействие процессов – один процесс пишет информацию в память, а другой считывает оттуда безо всяких ограничений. Системные вызовы обрабатываются каждым процессором.

Недостатки данной схемы. Совместного планирования процессов для всех процессоров нет, поэтому один процессор может быть загружен, а другой – нет. Совместного использования страниц памяти нет, т.е. если у одного процессора свободно много памяти, а у другого нет, то изменить ситуацию невозможно. Проблемы с устройствами ввода-вывода с буферным кэшем (например, жесткий диск) приводят к порче данных из-за невозможности синхронизации со всеми процессорами.

2 *Мультипроцессоры типа «хозяин-подчиненный».* Схема приведена на рисунке 71. Одна копия операционной системы находится на первом центральном процессоре, на него перенаправляются все системные вызовы. Процессы

пользователя работают на остальных процессорах, при этом могут работать и на первом, если у него остается время.

Достоинства: единая структура планирования процессов, страницы памяти управляются динамически, нет проблем с буферным кэшем.

Недостаток: при большом количестве центральных процессоров хозяин может не справляться с нагрузкой по обработке системных вызовов.

Рисунок 71 – Модель мультипроцессора «хозяин-подчиненный»

3 *Симметричные мультипроцессоры.* Схема представлена на рисунке 72.

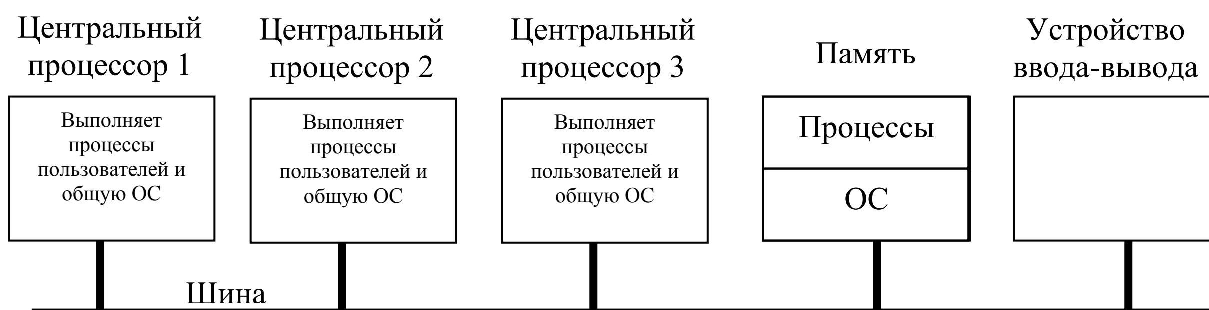

Рисунок 72 – Модель симметричного мультипроцессора

В памяти находится всего одна операционная система, но выполнять ее может любой процессор. При системном вызове на центральном процессоре, обратившемся к системе с системным вызовом, происходит прерывание с переходом в режим ядра и обработкой системного вызова.

Достоинства: Эта модель обеспечивает динамический баланс процессов и памяти, поскольку в ней имеется всего один набор таблиц операционной системы. Она также позволяет избежать простоя системы, связанного с перегрузкой «хозяина», как в предыдущей схеме. Такой подход используется в большинстве современных мультипроцессоров.

В мультипроцессорных системах все центральные процессоры совместно используют общую память. Синхронизация выполняется при помощи мьютексов, семафоров, мониторов и т.п. Однако большие мультипроцессоры сложны в построении и дороги. Поэтому используются многомашинные системы, которые ещё называют кластерными компьютерами. Основными компонентами кластерных компьютеров являются усеченные варианты обычных персональных компьютеров с сетевыми картами. Быстродействие системы кластерных компьютеров зависит от

схемы соединительной сети и интерфейсной карты. Бывают следующие виды соединений: звезда, кольцо, решетка, двойной тор, куб, гиперкуб [10, 14].

8.4 Операционные системы реального времени и мобильные операционные системы

Операционные системы реального времени (ОСРВ) предназначены для обеспечения интерфейса к ресурсам критических по времени систем реального времени [2]. Основной задачей в таких системах является своевременность (timeliness) выполнения обработки данных. В качестве основного требования к ОСРВ выдвигается требование обеспечения предсказуемости или детерминированности поведения системы в наихудших внешних условиях, что резко отличается от требований к производительности и быстродействию универсальных операционных систем. Хорошая ОСРВ имеет предсказуемое поведение при всех сценариях системной загрузки (одновременные прерывания и выполнение потоков).

Существует некое различие между системами реального времени и встроенными системами. От встроенной системы не всегда требуется, чтобы она имела предсказуемое поведение, и в таком случае она не является системой реального времени. Однако даже беглый взгляд на возможные встроенные системы позволяет утверждать, что большинство встроенных систем нуждается в предсказуемом поведении, по крайней мере, для некоторой функциональности, и таким образом, эти системы можно отнести к системам реального времени.

Принято различать системы мягкого (soft) и жесткого (hard) реального времени. В системах жесткого реального времени неспособность обеспечить реакцию на какие-либо события в заданное время ведет к отказам и невозможности выполнения поставленной задачи. Системы мягкого реального времени могут не успевать решать задачу, но это не приводит к отказу системы в целом.

Мартин Тиммерман сформулировал следующие необходимые требования для ОСРВ [18]:

- система должна быть многозадачной и допускающей вытеснение;
- система должна обладать понятием приоритета для потоков;
- система должна поддерживать предсказуемые механизмы синхронизации,
- система должна обеспечивать механизм наследования приоритетов;
- поведение операционной системы должно быть известным и предсказуемым (задержки обработки прерываний, задержки переключения задач, задержки драйверов и т.д.); это значит, что во всех сценариях рабочей нагрузки системы должно быть определено максимальное время отклика.

Как правило, большинство современных ОСРВ построено на основе микроядра (kernel или nucleus), которое обеспечивает планирование и диспетчеризацию задач, а также осуществляет их взаимодействие. Несмотря на сведение к минимуму в ядре абстракций операционной системы, микроядро все же должно иметь представление об абстракции процесса. Все остальные

концептуальные абстракции операционных систем вынесены за пределы ядра, вызываются по запросу и выполняются как приложения.

Большие различия в спецификациях ОСРВ и огромное количество существующих микроконтроллеров выдвигают на передний план проблему стандартизации в области систем реального времени.

Наиболее ранним и распространенным стандартом ОСРВ является стандарт POSIX (IEEE Portable Operating System Interface for Computer Environments, IEEE 1003.1). Первоначальный вариант стандарта POSIX появился в 1990 г. и был предназначен для UNIX-систем, первые версии которых появились в 70-х годах прошлого века. Спецификации POSIX определяют стандартный механизм взаимодействия прикладной программы и операционной системы и в настоящее время включают набор более чем из 30 стандартов. Для ОСРВ наиболее важны семь из них (1003.1a, 1003.1b, 1003.1c, 1003.1d, 1003.1j, 1003.21, 1003.2h), но широкую поддержку в коммерческих ОС получили только три первых.

Несмотря на явно устаревшие положения стандарта POSIX и большую востребованность обновлений стандартизации для ОСРВ, заметного продвижения в этом направлении не наблюдается.

Стандарт DO-178B, создан Радиотехнической комиссией по аэронавтике (RTCA, Radio Technical Commission for Aeronautics) для разработки ПО бортовых авиационных систем. Первая его версия была принята в 1982 г., вторая (DO-178A) - в 1985-м, текущая DO-178B - в 1992 г. Готовится принятие новой версии, DO-178C. Стандартом предусмотрено пять уровней серьезности отказа, и для каждого из них определен набор требований к программному обеспечению, которые должны гарантировать работоспособность всей системы в целом при возникновении отказов данного уровня серьезности.

Стандарт ARINC-653 (Avionics Application Software Standard Interface) разработан компанией ARINC в 1997 г. Этот стандарт определяет универсальный программный интерфейс APEX (Application/Executive) между ОС авиационного компьютера и прикладным ПО. Требования к интерфейсу между прикладным ПО и сервисами операционной системы определяются таким образом, чтобы разрешить прикладному ПО контролировать диспетчеризацию, связь и состояние внутренних обрабатываемых элементов. В 2003 г. принята новая редакция этого стандарта. ARINC-653 в качестве одного из основных требований для ОСРВ в авиации вводит архитектуру изолированных (partitioning) виртуальных машин.

Стандарт OSEK/VDX является комбинацией стандартов, которые изначально разрабатывались в двух отдельных консорциумах, впоследствии слившихся. OSEK берет свое название от немецкого акронима консорциума, в состав которого входили ведущие немецкие производители автомобилей – BMW, Bosch, Daimler Benz (теперь Daimler Chrysler), Opel, Siemens и Volkswagen, а также университет в Карлсруэ (Германия). Проект VDX (Vehicle Distributed eXecutive) развивался совместными усилиями французских компаний PSA и Renault. Команды OSEK и VDX слились в 1994г.

Ниже кратко описаны некоторые современные операционные системы реального времени по источнику [2].

VxWorks

Операционные системы реального времени семейства VxWorks корпорации WindRiver Systems предназначены для разработки программного обеспечения встраиваемых компьютеров, работающих в системах жесткого реального времени. Операционная система VxWorks обладает кросс-средствами разработки программного обеспечения, т.е. разработка ведется на инструментальном компьютере в среде Tornado для дальнейшего ее использования на целевом компьютере под управлением системы VxWorks.

Операционная система VxWorks имеет архитектуру клиент-сервер и построена в соответствии с технологией микроядра, т.е. на самом нижнем непрерываемом уровне ядра (WIND Microkernel) обрабатываются только планирование задач и управление их взаимодействием/синхронизацией. Вся остальная функциональность операционного ядра – управление памятью, вводом/выводом и прочее – обеспечивается на более высоком уровне и реализуется через процессы. Это обеспечивает быстродействие и детерминированность ядра, а также масштабируемость системы.

QNX Neutrino RTOS

Операционная система QNX Neutrino Realtime Operating System (RTOS) корпорации QNX Software Systems является микроядерной операционной системой, которая обеспечивает многозадачный режим с приоритетами. QNX Neutrino RTOS имеет клиент-серверную архитектуру. В среде QNX Neutrino каждый драйвер, приложение, протокол и файловая система выполняются вне ядра, в защищенном адресном пространстве. В случае сбоя любого компонента он может автоматически перезапуститься без влияния на другие компоненты или ядро. Хотя система QNX является конфигурируемой, т.е. отдельные модули можно загружать статически или динамически, нельзя сказать, что она использует подход, основанный на компонентах. Все модули полагаются на базовое ядро и спроектированы таким образом, что не могут использоваться в других средах.

RTEMS

RTEMS (Real-Time Executive for Multiprocessor Systems) – это некоммерческая операционная система реального времени для глубоко встраиваемых систем. Разработчик системы компания OAR (On-Line Applications Research Corporation, США). Система была создана по заказу министерства обороны США для использования в системах управления ракетными комплексами. Система разрабатывается для многопроцессорных систем на основе открытого исходного кода в противовес аналогичным системам с закрытым кодом. Система рассчитана на платформы MS-Windows и Unix (GNU/Linux, FreeBSD, Solaris, MacOS X).

Есть еще множество других ОС реального времени и описать их нет возможности.

8.4.1 Операционная система Windows CE 5.0

Наиболее распространённый класс ОСРВ – это операционные системы для мобильных устройств. На настоящий момент в ходу множество мобильных устройств, на которых установлены собственные операционные системы, которые в основе свой работы содержат описанные принципы работы. Сегодня на рынке

присутствует несколько наиболее известных операционных систем: вездесущая Microsoft с семейством Windows Mobile, Symbian, Palm, а также некоторые малораспространенные ОС, такие как Mac OS X или Linux.

Windows CE – это вариант компактной операционной системы для наладонных компьютеров (Handheld PC) и встраиваемых систем. Эта система очень похожа на настольный вариант Windows, однако не имеет с ним ничего общего. Основной особенностью этой ОС является возможность работы с малым объемом оперативной памяти (от 5 Мегабайт), а также поддержка мобильных процессоров архитектуры ARM, MIPS, Hitachi SuperH (мобильные процессоры) и даже x86 (настольные процессоры).

Последней версией из этого семейства является система Microsoft Windows CE 5.0, в которой объединены возможности систем реального времени и последние технологии Windows. В отличие от других ОСРВ Windows CE проектировалась так, чтобы она была совместимой с универсальными операционными системами.

ОСРВ Windows CE является модульной с небольшим ядром и необязательными модулями, которые выполняются как независимые процессы. Планирование в Windows CE осуществляется на основе приоритетов. Поддерживается защита ядра и процессов друг от друга. Кроме того, возможен режим работы, когда отсутствует защита между процессами и ядром. Следует отметить, что прерывания обрабатываются как потоки и имеют уровни приоритетов потоков. Windows CE поддерживает также нити, являющиеся потоками, которыми ядро не управляет. Каждая нить выполняется в контексте потока, который ее создал; их можно использовать для создания планировщика внутри потока. Такие нити используются в экзотических или унаследованных приложениях, но они непригодны в системах реального времени.

Windows CE имеет ограничение на физическую память – 512MB. RAM в устройстве Windows CE разделяется на две области – хранилище объектов и программная память. Хранилище объектов напоминает постоянный виртуальный диск RAM. Данные в таком хранилище запоминаются во время приостановки или операции частичной переустановки. Когда операция возобновляется, система находит ранее созданное хранилище объектов и использует его. Программная память состоит из оставшейся RAM, она работает как RAM в персональном компьютере – запоминает стеки и области для динамически выделяемой памяти выполняющихся приложений.

Во время старта Windows CE создает единое виртуальное адресное пространство в 4GB, которое затем разделяется на две секции – ядро и пользовательское пространство, как и в универсальной операционной системе Windows. Далее пользовательское пространство делится на 64 слота по 32MB, из которых 32 резервируются для процессов (отсюда ограничение на число процессов в системе). Все процессы разделяют виртуальное адресное пространство, но не имеют доступа друг к другу. В виртуальном адресном пространстве в 32MB находится все, что нужно процессу – программа, данные, область динамической памяти. Если процесс имеет соответствующие права доступа, он может получить память сверх ограничения в 32MB, обратившись к специальному процессу или используя файлы, отображаемые на память.

Windows CE реализует страничное управление виртуальной памятью. Размер страницы зависит от платформы, но, по возможности, используется размер в 4KB. Есть возможность запретить страничную организацию, что важно для систем реального времени. В этом режиме модуль перед выполнением целиком загружается в память. Тогда страничная подкачка не повлияет на выполнение приложения.

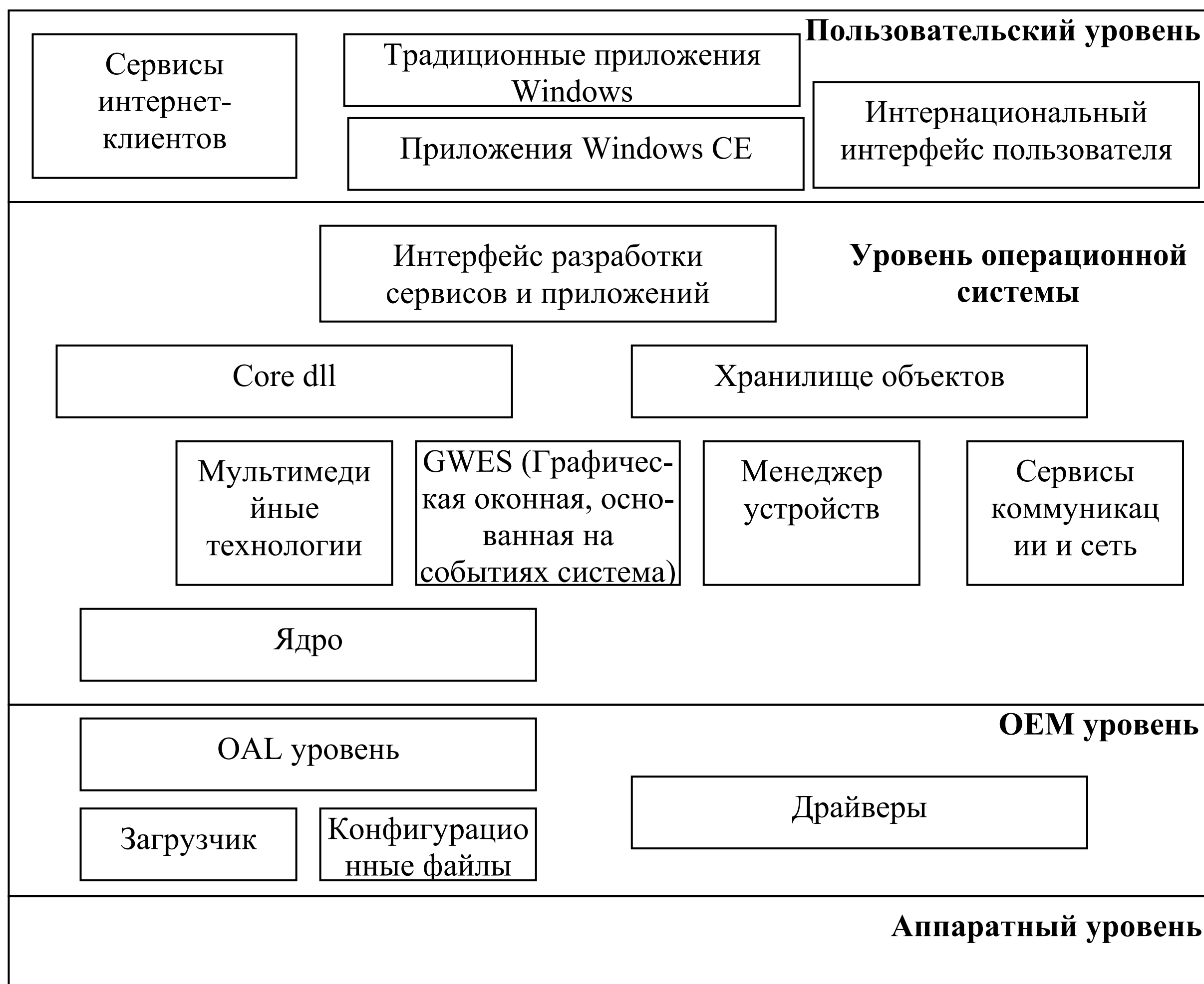

Рисунок 73 – Архитектура Windows CE

Механизмы синхронизации в Windows CE можно разделить на две категории:

- механизмы защиты от одновременного доступа – критические секции, мьютексы и семафоры;
- механизмы взаимодействия – события и очереди сообщений.

На рисунке 73 приведена архитектура операционной системы Windows CE. Здесь OEM – уровень производителей оборудования, OAL – уровень адаптации, Core dll – блок библиотек, отвечающих за взаимодействие с ядром.

В былые времена эта операционная система использовалась в наладонных компьютерах, которые функционально недалеко ушли от записной книжки, а сегодня Windows CE является основной платформой для построения автомобильных навигаторов, что дает возможность создавать многофункциональные устройства с мультимедийным уклоном. Дело в том, что ОС не только поддерживает установку стороннего софта, но и имеет многозадачность, благодаря чему устройства на

Windows CE обладают весьма неплохим набором функций. Естественно, поддерживается и сенсорный экран. Сегодня интерфейсом Windows CE (пятой версии) могут похвастаться автомобильные и карманные навигаторы, особенно известные среди них модели от Nokia, Mitac Mio, LG, Clarion, GlobalSat, Pocket Navigator и других.

Контрольные вопросы по разделу

1 Перечислите ключи реестра Windows 2000. За что отвечает ключ реестра HKEY_PERFORMANCE_DATA?

2 На какие две части делится операционная система Windows 2000?

3 Дайте определение управляющим объектам, находящимся в ядре операционной системы Windows 2000. На какие два класса они подразделяются?

4 Какие интерфейсы прикладного программирования поддерживает Windows 2000? Как реализуется вызовы Win32 API?

5 Объясните, каким образом Windows 2000 поддерживает программы MS-DOS?

6 Перечислите три основных интерфейса операционной системы UNIX.

7 Для чего служат обработанный и необработанный телетайп?

8 Какие два вида мультипроцессоров существуют и чем они отличаются?

9 Кратко опишите три основных вида реализации операционных систем на мультипроцессорах.

10 Сформулируйте требования для операционных систем реального времени. Какие вы знаете операционные системы реального времени?

11 Чем отличается Windows CE 5.0 от других версий операционных систем семейства Windows.

Список использованных источников

1 Аралбаев, Т.З. Контроль и управление доступом в АСУ ТП на основе биометрических характеристик пользователя / Т.З. Аралбаев, А.Г. Африн. – Уфа: Гилем, 2008. – 124 с.
2 Бурдонов, И.Б. Операционные системы реального времени / И.Б. Бурдонов, А.С. Косачев, В.Н. Пономаренко // Препринт Института системного программирования РАН. – 2006. – № 14.
3 Зубков, С.В. Assembler. Для DOS, Windows и Unix / С.В. Зубков. – М.: ДМК, 1999. – 640 с.
4 Казарин, О.В. Безопасность программного обеспечения компьютерных систем / О.В. Казарин. – М.: МГУЛ, 2003. – 212 с.
5 Кэрриэ, Б. Криминалистический анализ файловых систем / Б. Кэрриэ. – СПб.: Питер, 2007. – 480с.
6 Мак-Клар, С. Секреты хакеров. Безопасность сетей – готовые решения / С. Мак-Клар, Дж. Скембрей, Дж. Курц. – М. : Издательский дом «Вильямс», 2002. – 736с.
7 Мак-Клар, С. Секреты хакеров. Проблемы и решения сетевой защиты / С. Мак-Клар, Дж. Скембрей, Дж. Курц. – М.: Лори, 2001. – 464 с.
8 Михайлов, Д. Файловая система NTFS. [Электронный ресурс] / Д. Михайлов. – Режим доступа: http://www.ixbt.com/storage/ntfs.html. – Проверено 17.08.2009.
9 Михайлов, Д. Надежность дисковой системы NT. [Электронный ресурс] / Д. Михайлов. – Режим доступа: http://www.ixbt.com/storage/ntfs2.html. – Проверено 17.08.2009.
10 Немюгин, С.А. Параллельное программирование для многопроцессорных вычислительных систем / С.А. Немюгин, О.Л. Стесик. – СПб.: БХВ-Петербург, 2002. – 400с.
11 Олифер, В.Г. Сетевые операционные системы / В.Г. Олифер, Н.А. Олифер. – СПб.: Питер. – 2002. – 544 с.
12 Саймон, Р. Windows 2000 API. Энциклопедия программиста : пер. с англ. / Р. Саймон. – СПб. : ООО «ДиаСофтЮП», 2002. – 1088 с.
13 Таненбаум, Э. Архитектура компьютера : пер. с англ. / Э. Таненбаум. – 4 изд. – СПб: Питер. – 2003. – 689с.
14 Таненбаум, Э. Современные операционные системы : пер. с англ. / Э. Таненбаум. – 2 изд. – СПб.: Питер. -2002. -1040 с.
15 Coffman, E.G., Elphick, M.J., and Shoshani, A.: «System Deadlocks», Computing Surveys, vol. 3, pp. 67-78, June 1971.
16 Dijkstra, E.W.: «Cooperating Sequential Processes», in Programming Languages, Genuys, F. (Ed.), London: Academic Press, 1965.
17 Peterson, G.L.: «Myths about the Mutual Exclusion Problem», Information Processing Letters, vol. 12, pp. 115-116, June 1981.
18 http://www.dedicated-systems.com.