МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ ФЕДЕРАЦИИ

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ
БЮДЖЕТНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ
ВЫСШЕГО ПРОФЕССИОНАЛЬНОГО ОБРАЗОВАНИЯ
«САМАРСКИЙ ГОСУДАРСТВЕННЫЙ АЭРОКОСМИЧЕСКИЙ
УНИВЕРСИТЕТ имени академика С.П. КОРОЛЕВА
(НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ»

*С.В. ВОСТОКИН*

# ОПЕРАЦИОННЫЕ СИСТЕМЫ

Рекомендовано редакционно-издательским советом
федерального государственного бюджетного образовательного
учреждения высшего профессионального образования
«Самарский государственный аэрокосмический университет
имени академика С.П. Королева (национальный исследовательский
университет)» в качестве учебника для студентов, обучающихся
по образовательным программам высшего профессионального образования
по направлениям подготовки бакалавров «Фундаментальная информатика
и информационные технологии», «Прикладная математика
и информатика», «Прикладная математика и физика»

САМАРА
Издательство СГАУ
2012

УДК СГАУ: 004.451(075)
ББК 32.973я7
В 78

Рецензенты: д-р тех. наук, проф. С. П. О р л о в,
д-р физ.-мат. наук, проф. Д.Л. Г о л о в а ш к и н

***Востокин С.В.***

В78 **Операционные системы [Электронный ресурс]:** учеб. / *С.В. Востокин.* – Самара: Изд-во Самар. гос. аэрокосм. ун-та, 2012. – 120 с.

**ISBN 978-5-7883-0916-3**

Рассмотрены основные вопросы дисциплины «Операционные системы»: обзор операционных систем, проектирование и архитектуры операционных систем, управление процессами и потоками, мультипроцессорная обработка, методы синхронизации процессов, проблема тупиков, методы управления памятью, основы организации ввода-вывода и файловых систем. Рассмотрены практические вопросы программирования на примере операционной системы Windows.Приведены экзаменационные вопросы по разделам дисциплины.

Учебник предназначен для студентов специальностей и направлений «Информатика и вычислительная техника», «Прикладная математика и информатика», «Фундаментальная информатика и информационные технологии», «Прикладная математика и физика».

УДК СГАУ: 004.451(075)
ББК 32.97я7

**ISBN 978-5-7883-0916-3**

## ОГЛАВЛЕНИЕ

## РАЗДЕЛ I. ОСНОВНЫЕ ПОНЯТИЯ

### Лекция 1. Определение операционной системы. Первые операционные системы

Определение операционной системы, операционная система как виртуальная машина и как система управления ресурсами. Поколения компьютеров и операционных систем.

**Операционная система** — это комплекс программ, которые выступают как интерфейс между устройствами вычислительной системы и прикладными программами, предназначены для управления устройствами и вычислительными процессами, а также для эффективного распределения вычислительных ресурсов и организации надёжных вычислений.

Таким образом, операционная система - это программное обеспечение, выполняющее две функции: 1) предоставление пользователю-программисту более удобной в использовании «виртуальной машины», скрывающей реальное оборудование; 2) обеспечение эффективного использования компьютера путем рационального управления его ресурсами.

Рассмотрим, для чего необходим программный интерфейс между оборудованием и программой пользователя на примере. Использование большинства компьютеров на уровне машинного языка затруднительно, особенно при организации ввода-вывода. Например, для чтения блока данных с гибкого диска программист может использовать 16 различных команд контроллера NEC PD765, каждая из которых требует 13 параметров: номера поверхности на диске, сектора на дорожке и других. Когда выполнение операции с диском завершается, контроллер возвращает 23 значения, отражающих, в том числе, наличие и типы ошибок, которые необходимо анализировать. Кроме этого, работа с диском требует специальной организации кода программы, связанной с необходимостью обработки прерываний, отслеживания состояния двигателя привода для уменьшения износа диска с сохранением высокой скорости обращения, организации хранения данных и учета других особенностей.

При работе с диском, используя функции операционной системы, программисту достаточно представлять данные в виде некоторого набора файлов, каждый из которых имеет имя. Работа с файлом заключается в открытии, выполнении чтения или записи, а затем в закрытии файла. Программа, которая скрывает от программиста все реалии аппаратуры и предоставляет возможность простого, удобного чтения или записи файлов – это операционная система. Операционная система ограждает программистов от аппаратуры дискового накопителя и предоставляет ему простой файловый интерфейс, берет на себя обработку прерываний, управление таймерами и оперативной памятью, а также другие низкоуровневые проблемы. В каждом случае абстракция (например, файл), с которой, благодаря операционной системе, может иметь дело пользователь, гораздо проще и удобнее в обращении, чем реальная аппаратура, лежащая в основе этой абстракции. С этой точки зрения функцией операционной системы является предоставление пользователю некоторой расширенной или «виртуальной машины», которую легче программировать и с которой легче работать, чем непосредственно с аппаратурой, составляющей реальную машину.

Программист-пользователь считает, что операционная система, прежде всего, обеспечивает удобный интерфейс. Разработчик аппаратуры имеет представление об операционной системе как о некотором механизме, управляющем всеми частями сложной системы. Современные вычислительные системы состоят из процессоров, памяти, таймеров, дисков, накопителей на магнитных лентах, сетевой коммуникационной аппаратуры, принтеров и других устройств. Функцией операционной системы является распределение процессоров, памяти, устройств и данных между процессами, конкурирующими за эти ресурсы. Операционная система должна управлять всеми ресурсами вычислительной машины таким образом, чтобы обеспечить максимальную эффективность ее функционирования. Критерием эффективности может быть, например, пропускная способность или реактивность системы. Управление ресурсами включает решение двух общих, не зависящих от типов ресурсов задач: планирование ресурсов и мониторинг ресурсов.

Планирование ресурса – это определение кому, когда, а для делимых ресурсов и в каком количестве, необходимо выделить данный ресурс.

Мониторинг (отслеживание состояния ресурса) – поддержание оперативной информации о том, занят или не занят ресурс, а для делимых ресурсов, какое количество ресурса уже распределено, а какое свободно.

Для решения этих общих задач управления ресурсами разные операционные системы используют различные алгоритмы. Это, в конечном счете, определяет их облик в целом, включая характеристики производительности, область применения и даже пользовательский интерфейс.

Рассмотренное функциональное определение операционной системы применимо к большинству современных операционных систем общего назначения.

Для понимания основных механизмов, используемых в современных операционных системах, кратко рассмотрим историю их развития. Применимость той или иной архитектурной идеи в программном обеспечении (software) во многом определяется наличием и доступностью аппаратного обеспечения (hardware).

**Первое поколение компьютеров и операционных систем (1945-1955 гг.), электронные лампы и коммутационные панели.** Изобретателем цифрового компьютера считается английский математик Чарльз Беббидж. В 1833 году им был предложен проект механической универсальной цифровой вычислительной машины — прообраза современной ЭВМ. Первыми действующими компьютерами являлись: компьютер Z3, созданный немецким инженером Конрадом Цузе (1941); компьютер Марк I, созданный американским инженером Говардом Эйкеном (1941); компьютер ЭНИАК, разработанный Джоном Экертом и Джоном Мокли (1945). Первые компьютеры были электромеханическими (реле). В поздних моделях реле были заменены электронными лампами.

Первые компьютеры программировались на абсолютном машинном языке, управление основными функциями машины осуществлялось путем соединения коммутационных панелей проводами. Такая панель являлась носителем программы и подключалась к компьютеру. Также для записи программ использовались перфорированные ленты. В начале 50х панели и ленты были заменены перфокартами.

На компьютерах первого поколения занимались только прямыми числовыми вычислениями, например, расчетами таблиц синусов,

косинусов, логарифмов. Компьютеры имели военные приложения: расчет стреловидных крыльев и управляемых ракет, расчет баллистических таблиц для стрельбы. Компьютеры не имели операционных систем. Программирование осуществлялось в интерактивном режиме: пользователь-программист получал полный контроль над машиной на время отладки программы и выполнения вычислений. В некотором смысле первые компьютеры напоминали современные персональные ЭВМ.

Рис. 1.1. Машинный зал с компьютером IBM 7094, НАСА. На переднем плане, внизу – считыватель перфокарт

**Второе поколение компьютеров и операционных систем (1955-1965 гг.), транзисторы и системы пакетной обработки заданий.** С появлением в середине 50х годов транзисторов компьютеры стали достаточно надежными и могли без сбоев работать длительное время. Такие компьютеры назывались мэйнфреймами. Они располагались в специальных помещениях с кондиционированным воздухом, ими управлял целый штат профессиональных операторов. Цена одного из известных мэйнфреймов IBM 7090/94 рис. 1.1 составляла $2,900,000, а стоимость аренды $63,500 в месяц.

Рассмотрим организацию работы на IBM 7090/94 и подобных машинах. Для программирования использовался язык высокого

уровня Фортран или ассемблер. Программа для компьютера вначале писалась на бумаге, а затем переносилась на перфокарты при помощи перфораторов: электронно-механических устройств, похожих на печатающие машинки (рис. 1.2). Каждая перфокарта представляла одну строчку кода программы. Перфокарты выполняли функции современных текстовых редакторов. Программа – это колода перфокарт, которая вставлялась в устройство для считывания. Результат работы очередной задачи отображался на принтере. Если в процессе расчетов был необходим компилятор языка Фортран, то оператору необходимо было брать его из карточного шкафа и загружать в машину отдельно.

Рис. 1.2. Перфоратор IBM 026

С учетом высокой стоимости оборудования необходимо было повысить эффективность использования машинного времени, сократить простои машины при загрузке программ в память. Данную функцию реализовывали первые операционные системы – системы пакетной обработки.

Рассмотрим автоматизацию расчетов на мэйнфрейме IBM 7094 с использованием пакетной системы IBSYS (рис. 1.3).

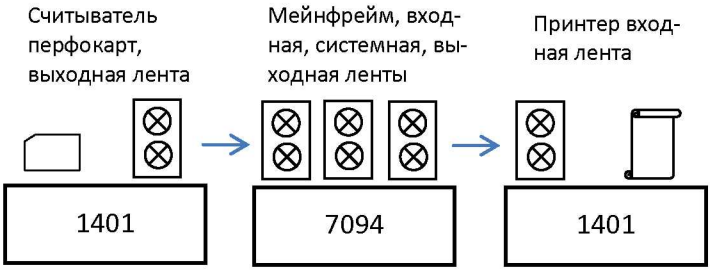



Рис.1.3. Схема работы пакетной системы

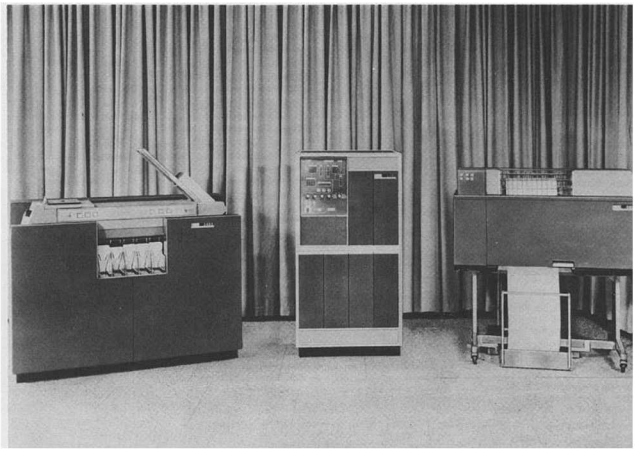


Рис. 1.4. Считыватель-перфоратор 1402, управляющий блок 1401 и принтер 1403 фирмы IBM

Для подготовки пакета заданий использовались относительно недорогие компьютеры IBM 1401 (рис.1.4). К такому компьютеру подключался считыватель перфокарт и устройство записывания магнитных лент. Примерно после часа сбора пакета заданий лента перематывалась, и ее относили в машинную комнату. Там ее устанавливали на лентопротяжное устройство, подключенное к мэйнфрейму. Также к нему подключалась лента с программой операционной системы и выходная лента. По мере накопления выходные данные записывались на ленту вместо того, чтобы идти на печать.

После обработки всего пакета заданий входные и выходные ленты менялись на новые для следующего цикла обработки. Печать результатов на принтере также осуществлялась под управлением маломощного и недорогого компьютера IBM 1401.

Большие компьютеры второго поколения использовались главным образом для научных и технических вычислений, таких как решение дифференциальных уравнений в частных производных, часто встречающихся в физике, и инженерных задачах. Поэтому режим работы с непродолжительными этапами ввода-вывода с использованием магнитных лент значительно оптимизировал загрузку мэйнфрейма. Кроме этого, длительные операции, такие как считывание перфокарт и печать на медленных принтерах, происходили одновременно с расчетами. Такой способ взаимодействия с внешними устройствами получил название автономного (off-line), в отличие от интерактивного способа (on-line), используемого в системах третьего поколения.

Принципы пакетной обработки, несмотря на появление в 60х годах, широко используются в современных системах для аналогичных целей: оптимизации загрузки дорогостоящей системы. Примером современной пакетной системы является система TORQUE (Terascale Open-Source Resource and QUEue Manager), служащая для управления суперкомпьютерами. Также пакетные режимы имеются во всех современных операционных системах разделения времени (Unix, Mac OS, Windows).

## Лекция 2. Современные операционные системы

Поколения компьютеров и операционных систем: интегральные схемы и многозадачность, персональные компьютеры. Классификация и примеры операционных систем.

**Третье поколение компьютеров и операционных систем (1965-1980 гг.), интегральные схемы и многозадачность.** Дальнейшее развитие компьютерная индустрия получила при появлении технологии мелкомасштабных интегральных схем, которая давала преимущество в цене и качестве по сравнению с машинами второго поколения, созданными из отдельных транзисторов.

Основные разработки в области архитектуры современных операционных систем появились именно на компьютерах третьего поколения. Операционные системы, состоящие из миллионов строк, написанных тысячами программистов, на два или три порядка превышали по сложности первые системы типа IBSYS.

Основу к появлению сложного системного программного обеспечения заложила фирма IBM, которая выпустила серию программно-совместимых машин семейства IBM /360. Эти компьютеры различались только ценой и производительностью. Так как все машины семейства имели одинаковую структуру и набор команд, программы, написанные для одного компьютера, могли работать на других. Компьютеры семейства IBM /360 могли использоваться как для сложных научных расчетов, так и для статистической обработки, сортировки и печати. Предполагалось, что одну операционную систему можно будет использовать как с несколькими внешними устройствами, так и с большим их количеством. Одно семейство компьютеров было призвано удовлетворить потребности всех покупателей.

Первой операционной системой для компьютеров IBM /360 была OS /360 (IBM System /360 Operating System). Разработкой системы руководил Фред Брукс, описавший ее в знаменитой книге «Мифический человеко-месяц». В этом проекте впервые столкнулись с проблемой сложности программного обеспечения, с него начала развиваться дисциплина программной инженерии, впервые были применены многие технические приемы, используемые в современных операционных системах.

Самым важным достижением OS /360 являлось использование многозадачности. На компьютере IBM 7094, когда текущая работа приостанавливалась в ожидании данных с магнитной ленты или других устройств, центральный процессор бездействовал до окончания ввода-вывода. Данная ситуация была некритичной при обработке задач численного моделирования. Однако для универсальной системы, на которой обрабатывались задачи, связанные с сортировкой и другой обработкой информации на лентах, время простоя оказывалось существенным и составляло 80-90%. Необходимо было предложить решение, предотвращающее длительный простой дорогостоящих процессоров.

Решение проблемы было найдено. Оно заключалось в разбиении памяти на несколько частей (разделов), каждому из которых давалось отдельное задание (рис.2.1). Пока одно задание ожидало завершение операций ввода-вывода, другое могло использовать центральный процессор. Если в оперативной памяти размещалось достаточное количество заданий, центральный процессор мог быть загружен почти 100% времени. Для реализации многозадачности потребовалось разработать аппаратуру защиты памяти и средства обработки прерываний. Применение многозадачности позволило отказаться от процедуры предварительной подготовки данных на магнитных лентах и выгрузки на ленту для печати. Данные процедуры являлись обычными фоновыми операциями ввода-вывода. Технический прием получил название спулинг (spooling – simultaneous peripheral operation on line).

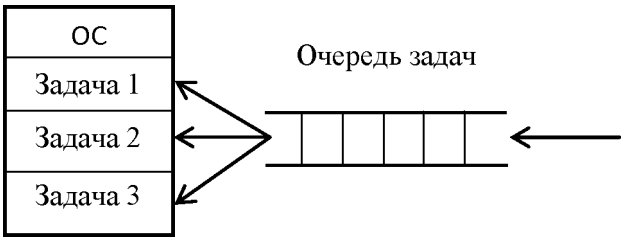



Рис. 2.1. Многозадачная система с тремя заданиями в памяти

Операционная система OS /360 и другие операционные системы третьего поколения являлись, по сути, сложными системами пакетной обработки. Главный недостаток таких систем заключался в том, что, несмотря на оптимизацию работы вычислительной системы, организация работы самих программистов являлась неэффективной. Промежуток времени между передачей задания и получением результата составлял несколько часов. Сбой при компиляции, связанный с незначительной синтаксической ошибкой, приводил к большой потере времени. Желание применить принципы интерактивной разработки, известные по компьютерам первого поколения на новых системах, привело к появлению режима разделения времени.

Режим разделения времени – вариант многозадачности, при котором у каждого пользователя есть свой диалоговый терминал. Так как пользователи чаще выдают короткие команды компиляции при отладке программ, компьютер может обеспечить быстрое интерактивное обслуживание нескольких пользователей.

Первая система разделения времени CTSS (Compatible Time-Sharing System) была разработана Фернандо Корбато и его командой в Массачусетском технологическом институте (M.I.T.) на специально переделанном компьютере IBM 7094. Впоследствии в 1962 году решения, использовавшиеся в демонстрационной системе CTSS, легли в основу проекта первой полноценной системы с разделением времени, названной Multiplexed Information and Computer System (Multics) на компьютере GE-645, позднее Honeywell 6180.

Работа над системой Multics продолжалась вплоть до 1969 года, но коммерческого успеха она не имела. Однако этот проект имел определяющее значение для дальнейшего развития компьютерных технологий. В системе Multics впервые реализованы: сегментно-страничная виртуальная память; отображение файлов на адресное пространство процессов; динамическое связывание исполняемой программы с библиотеками. В ней применялись даже более сложные и гибкие механизмы взаимодействия процессов через разделяемые сегменты, чем в современных системах. Использовалось «горячее» реконфигурирование системы с заменой процессоров, блоков памяти, жестких дисков без остановки обслуживания пользователей. Multics – одна из первых мультипроцессорных систем. Multics была также одной из первых систем, в которой большое внимание уделялось безопасности взаимодействия между программами и пользователями. Более того, она была самой первой операционной системой, задуманной изначально и реализованной как безопасная. Дополнительно к этому, в Multics одной из первых была реализована иерархическая файловая система, имена файлов могли быть практически произвольной длины и содержать любые символы. Файл или директория могли иметь несколько имён (короткое и длинное), были доступны для использования символьные ссылки (symlink) между директориями. В Multics был впервые реализован (теперь уже стандартный) подход использования стеков для каждого вычислительного процесса в ядре системы: с отдельным стеком для каждого уровня безопасности вокруг ядра. Multics явилась одной из первых опера-

ционных систем, написанных на языке высокого уровня PL/I. Можно сказать, что идея «облачных» вычислений, популярная в настоящее время, также берет начало от системы Multics – компьютерного предприятия общественного пользования 60-70 годов прошлого века.

Важным моментом развития компьютерной индустрии во времена третьего поколения был феноменальных рост миникомпьютеров, начиная с выпуска корпорацией DEC в 1961 году компьютера PDP-1 и наиболее коммерчески успешной модели PDP-11. Особенностью этого семейства являлась низкая цена ($120,000, что составляло 5% цены мэйнфрейма IBM) и, сопоставимая с мэйнфреймами, производительность на нечисловых задачах. Кроме этого, PDP-11 отличалась удачной «ортогональной» системой команд: можно было отдельно запоминать команды, и отдельно — методы доступа к операндам. Все регистры, в том числе служебные (указатель стека, счетчик команд), одинаково использовались со всеми командами.

Все эти факторы способствовали очень широкому распространению компьютеров этого семейства, а вместе с ними и операционной системы Unix. Операционная система UNIX была разработана сотрудниками Bell Labs, в первую очередь Кеном Томпсоном, Деннисом Ритчи и Дугласом Макилроем. В 1969 году Кен Томпсон, стремясь реализовать идеи, которые были положены в основу Multics, но на более скромном аппаратном обеспечении (DEC PDP-7), написал первую версию новой операционной системы. Брайан Керниган придумал для неё название – UNICS (UN-multiplexed Information and Computing System – примитивная информационная и вычислительная служба) – в противовес Multics. Позже это название сократилось до UNIX. В ноябре 1971 года вышла версия для PDP-11.Эта версия получила название «первая редакция» (Edition 1) и была первой официальной версией. Системное время все реализации UNIX отсчитывают с 1 января 1970.

UNIX-системы имеют большую историческую важность. UNIX популяризовала предложенную в Multics идею иерархической файловой системы с произвольной глубиной вложенности. Еще одной инновацией Multics, популяризированной UNIX, являлся интерпретатор команд. Так как интерпретатор и команды операционной системы - обычные программы, пользователь мог выбирать их в соответствии со своими предпочтениями, или даже написать собствен-

ную оболочку. Наконец, новые команды можно добавлять к системе без перекомпиляции ядра. Предложенный в командной строке UNIX способ создания цепочек программ, последовательно обрабатывающих данные (конвейер, pipeline), способствовал использованию параллельной обработки данных.

Существенными особенностями UNIX были полная ориентация на текстовый ввод-вывод и предположение, что размер машинного слова кратен восьми битам. Первоначально в UNIX не было даже редакторов двоичных файлов – система полностью конфигурировалась с помощью текстовых команд. Наибольшей и наименьшей единицей ввода-вывода служил текстовый байт, что полностью отличало ввод-вывод UNIX от ввода-вывода других операционных систем. Ориентация на текстовый восьмибитный байт сделала UNIX более масштабируемой и переносимой, чем другие операционные системы. UNIX способствовала широкому распространению регулярных выражений, которые были впервые реализованы в текстовом редакторе ed для UNIX. Возможности, предоставляемые UNIX-программами, стали основой стандартных интерфейсов операционных систем (POSIX).

Среди примеров известных UNIX подобных операционных систем: BSD, Solaris, Linux, Minix, Android, MeeGo, NeXTSTEP, MacOSX.

Отметим, что в нашей стране также были распространены компьютеры, аналогичные семействам IBM /360 и DECPDP. В конце 70х начале 80х годов в СССР и ряде соцстран разрабатывалось семейство управляющих ЭВМ (СМ ЭВМ). Наиболее развитой линией в семействе СМ ЭВМ был ряд машин, совместимых по системе команд и архитектуре с машинами PDP-11фирмы DEC. На них, например, использовались операционные системы РАФОС (аналог RT-11, системы реального времени для компьютера PDP-11), ДЕМОС (советско-российский клон Unix, созданный на основе BSD).

Аналогом серии мэйнфреймов IBM /360 и /370 в СССР являлась «Единая система электронных вычислительных машин» (ЕС ЭВМ). Операционная система ОС ЕС разрабатывалась как клон OS /360 путем дизассемблирования и адаптации исходных кодов. Версия ОС ЕС 7 использовала концепцию гипервизора (виртуальной машины). Эта версия - аналог операционной системы VM, еще одного типа операционных систем компьютеров серии /360 /370.

В целом можно отметить, что третье поколение компьютеров являлось источником основных архитектурных инноваций, используемых в современных операционных системах. Роль четвертого поколения заключается в широком внедрении этих инноваций.

**Четвертое поколение компьютеров и операционных систем (с 1980 года), персональные вычисления.** Следующий этап эволюции операционных систем связан с появлением больших интегральных схем (БИС), по-английски large scale integration (LSI). С точки зрения аппаратной архитектуры персональные компьютеры (микрокомпьютеры) были похожи на PDP-11, но значительно уступали по цене. Теперь каждый желающий получил возможность купить собственный компьютер.

Первой операционной системой для микрокомпьютеров была CP/M (Control Programfor Microcomputers), разработчик Гари Килдэлл, Digital Research. Она работала на первых 8 разрядных системах с процессором Intel 8080, Zilog Z80. С появлением в 1981 году новой серии IBM PC - совместимых компьютеров, на базе 16 разрядного процессора Intel 8088, на рынке программного обеспечения микрокомпьютеров операционную систему CP/M сменила система MS DOS компании Microsoft.

Операционные системы для первых микрокомпьютеров полностью основывались на вводе команд с клавиатуры. Однако еще в 60е годы в научно-исследовательском институте Стэнфорда (Stanford ResearchInstitute) Дугласом Энгельбартом был изобретен графический интерфейс пользователя (GUI, Graphical User Interface), в котором элементы интерфейса (меню, кнопки, значки, списки и т. п.), представленные пользователю на дисплее, исполнены в виде графических изображений. Графический интерфейс был реализован в прототипе современных персональных компьютеров Xerox Alto, разработанном в исследовательском центре XeroxPARC в 1973 году.

На Xerox Alto использовались программы с графическими меню, пиктограммами и другими элементами, ставшие привычными лишь с появлением операционных систем Mac OS и Microsoft Windows. Для Xerox Alto были разработаны текстовые процессоры Bravo и Gypsy, работавшие по принципу WYSIWYG (предшественники MSWord), редакторы графической информации (растровых изображений, печатных плат, интегральных схем и др.), среда Smalltalk (объектно-ориентированное программирование).

Первой успешной коммерческой реализацией GUI для персональных компьютеров была система Apple Macintosh, изготовленная под руководством Стива Джобса в 1984 году. Под влиянием успехов Apple, фирма Microsoft в 1985 году выпускает свою реализацию GUI – систему Windows. На протяжении 10 лет система Windows исполняла роль графической оболочки поверх MSDOS. Были разработаны еще две линейки настольных операционных систем. Семейство Windows 9x включает в себя Windows 95, Windows 98 и Windows ME. Оно являлось промежуточным семейством 32 разрядных систем. Семейство Windows NT – современные версии операционной системы Windows, берущие начало от системы Windows NT 3.1, выпущенной в 1993 году. Наиболее популярные версии операционных систем этого семейства: Windows NT 4.0; Windows 2000 (NT 5.0); Windows XP (NT 5.1); Windows Vista (NT 6.0); Windows 7 (NT 6.1); Windows 8 (NT 6.2).

Главными конкурентами систем Windows в персональных вычислениях становятся различные производные системы Unix: семейство операционных систем OSX Apple (BSD), Android (Linux), Ubuntu (Linux). Для серверных приложений и высокопроизводительных вычислений операционная система Unix (в частности, Linux) является доминирующей операционной системой.

Рассмотрим некоторые классификационные признаки и примеры операционных систем.

*По признаку поддержки многозадачности* различают однозадачные и многозадачные операционные системы. Однозадачные системы в основном выполняют функцию предоставления расширенной машины. Из рассмотренных операционных систем однозадачными являются MSDOS, CP/M, RT-11(версии SJ, SL), ранние версии Unix. Примеры многозадачных систем -Unix, Windows, операционная система персональных компьютеров с графическим интерфейсом IBMOS/2.

*По числу пользователей* операционные системы делятся на однопользовательские и многопользовательские. Главное отличие многопользовательских систем – это наличие средств защиты информации одного пользователя от несанкционированного доступа со стороны других пользователей. Многозадачные системы могут быть однопользовательскими, например Windows 3.1. Теоретически возможны однозадачные многопользовательские системы. Одной из

первых многопользовательских систем с развитыми механизмами защиты была система Multics. Пример другой однопользовательской системы Microsoft – MS DOS. В системах на базе ядра NT реализованы полноценные механизмы защиты многопользовательского режима.

*По аппаратным средствам*, на которых функционируют операционные системы, их можно разделить на системы для мэйнфреймов, миникомпьютеров, микрокомпьютеров. Примеры этих операционных систем были рассмотрены в историческом обзоре. Существуют также встраиваемые системы. Они работают непосредственно в устройствах, которыми управляют: средствах управления техническими процессами; станках с ЧПУ; банкоматах, платежных терминалах; телекоммуникационном оборудовании (Windows CE, QNX, Symbian OS). В отдельную группу можно выделить операционные системы наладонных компьютеров (PalmOS), смартфонов, телефонов и планшетных компьютеров (iOS, Android, Windows Phone). Самые производительные компьютеры нашего времени (суперкомпьютеры и кластерные системы) в основном работают под управлением Linux. В классификации, основанной на аппаратных средствах, разделение идет по возможности работы с устройствами ввода-вывода и периферийным оборудованием. Наименьшие возможности у встраиваемых систем интеллектуальных смарт-карт (MULTOS, JCOP). Наибольшие возможности – у суперкомпьютеров.

*По особенностям работы в сети* операционные системы делятся на сетевые и распределенные. Сетевая операционная система – это обычная система, расширенная поддержкой сетевого взаимодействия. По методам реализации сетевых функций системы различаются по способам реализации справочной информации о сетевых ресурсах и адресации; механизмам обеспечения прозрачность доступа. В линейке Windows первой сетевой системой была WindowsforWorkgroups 3.1. Распределённая операционная система, динамически и автоматически распределяя работы по различным машинам системы для обработки, заставляет набор сетевых машин обрабатывать информацию параллельно. Пользователь распределённой операционной системы не имеет сведений, на какой машине выполняется его работа. Примеры распределенных систем -Amoeba, Plan 9.

Существует несколько вариантов реализации многозадачных операционных систем. В зависимости от *способа переключения с задачи на задачу* различают системы с не вытесняющей многозадачностью и вытесняющей многозадачностью. Вытесняющая многозадачность предполагает использование прерываний от таймера как основной способ переключения задач. Операционные системы семейств Windows 1.0-3.11 использовали не вытесняющую многозадачность, а Windows 9x, WindowsNT – вытесняющую.

Еще одним аспектом реализации многозадачности является *использование концепции нитей (потоков) исполнения* – многозадачности внутри одного процесса. Нити использовались в Windows, начиная с Windows 95.

Многозадачные системы можно разделить на три большие группы по критерию эффективности алгоритма планирования и, что эквивалентно, по областям применения многозадачных операционных систем.

Критерии эффективности *системы пакетной обработки* ориентированы на обеспечение максимальной пропускной способности, максимально полной загрузки вычислительной системы. Критерий характеризуется процентом загрузки центрального процессора; количеством заданий, выполняемых в единицу времени; оборотным временем (среднее время вычисления задачи). В таких системах выбор нового задания для выполнения определяется от текущей внутренней ситуации, складывающийся в системе. Время выполнения задачи зависит от того, какие задачи загружены одновременно с ней и как в них происходит ввод-вывод. Поэтому невозможно гарантировать выполнение некоторого задания в течение определенного интервала времени, что является недостатком.

*Система разделения времени* устраняет данный недостаток пакетных систем. В таких системах распределение времени процессора организовано на основе квантования: периодического переключения управления между всеми одновременно запущенными задачами. С точки зрения задачи это создает иллюзию монопольного использования процессора (концепция терминала). Эффективность систем разделения времени определяется эргономическими показателями, связанными с субъективными ощущениями пользователя, как быстро система обрабатывает команды терминала. Системы разделения времени обладают меньшей пропускной способностью из-за

того, что ресурсы процессора тратятся на постоянное переключение между задачами.

Реальное время в операционных системах — это способность операционной системы обеспечить требуемый уровень сервиса в определённый промежуток времени. *Системы реального времени* применяют для управления техническими объектами. В них существует предельно допустимое время, в течение которого должна быть выполнена программа управления объектом. Это время называется временем реакции, а свойство операционных систем оперативно реагировать на события – реактивностью. Различают системы жёсткого реального времени — с режимом работы системы, при котором нарушение временных ограничений равнозначно отказу системы (RTLinux, AutomationRuntime, ART-Linux); системы мягкого реального времени — с режимом работы системы, при котором нарушения временных ограничений приводят к снижению качества работы (QNX, RT-11, WindowsCE).

Другими важными классификационными признаками являются: возможность исполнения операционной системы на разном типе компьютеров; особенности архитектур и типов ядра; используемые методы управления памятью; организация файловых систем. Они рассматриваются в следующих разделах лекционного курса.

### Экзаменационные вопросы по разделу I

1. Определение операционной системы и ее функции. Понятие виртуальной машины. Управление ресурсами.
2. История разработки операционных систем, поколения ЭВМ и операционных систем. Влияние аппаратуры на развитие операционных систем. Лампы - коммутационные панели. Транзисторы – пакетные системы. Интегральные схемы – многозадачность. СБИС – персональные компьютеры.
3. Классификация и примеры операционных систем. Многозадачность. Вид многозадачности. Многопоточная обработка. Критерии эффективности многозадачных операционных систем.

## РАЗДЕЛ II. ПРОЕКТИРОВАНИЕ ОПЕРАЦИОННЫХ СИСТЕМ

### Лекция 3. Требования к операционным системам и обзор подходов их реализации

Обзор требований, предъявляемых к операционным системам: расширяемость, переносимость, надежность и отказоустойчивость, совместимость, безопасность, производительность операционных систем.

Операционная система выполняет связующую роль между оборудованием и прикладными программами. Она является важнейшим элементом программной инфраструктуры, от свойств которой зависит качество работы прикладного программного обеспечения. Поэтому при проектировании операционных систем уделяют внимание специальным функциональным требованиям, более широким, чем при проектировании прикладных программ.

**Расширяемость.** Код операционной системы должен быть написан таким образом, чтобы при необходимости можно было легко внести дополнения и изменения, не нарушая целостности системы.

Во времена первых компьютеров считалось, что по мере обновления аппаратного обеспечения, код операционных систем будет создаваться заново, с нуля. Однако практика показала, что разработка программного обеспечения значительно более трудоемка, чем разработка аппаратуры. Возникла необходимость сохранения инвестиций, как в разработку операционной системы, так и в прикладные программы, функционирующие под ее управлением.

В то время как аппаратная часть компьютера устаревает за несколько лет, полезная жизнь операционной системы может измеряться десятилетиями (Unix). Аппаратные изменения, к которым должна быть, в первую очередь, адаптирована архитектура операционных систем, связаны с развитием периферийного оборудования. Например, современные модификации в операционных системах связаны с новыми технологиями хранения данных (накопители SSD), сетевого взаимодействия (беспроводные сети, оптические каналы высокой пропускной способности), новыми пользовательскими интерфейсами (жестовый ввод, голосовой ввод, сенсорные панели). Сохранение целостности кода операционной системы, учиты-

вая, что происходит постоянная модернизация аппаратуры, является одной из целей проектирования операционной системы.

Расширяемость может достигаться за счет модульной структуры операционной системы, при которой ее части состоят из сильно связанных внутри и слабо связанных между собой модулей. Взаимодействие между модулями организуется через стандартизированные интерфейсы. Новые компоненты, добавляемые в операционную систему, функционируют, используя интерфейсы, поддерживаемые существующими компонентами. Можно заменить код устаревших модулей, не затрагивая работоспособность системы в целом.

Использование объектов для представления системных ресурсов также улучшает расширяемость системы. Объекты позволяют единообразно управлять системными ресурсами. Добавление новых объектов не разрушает существующие объекты и не требует изменений существующего кода.

Вариантом модульной организации является применение архитектуры клиент-сервер и микроядра. В ней модули изолированы в отдельных процессах и могут замещаться даже без перекомпиляции и остановки вычислений.

Другой возможностью расширить функциональные возможности операционной системы являются средства вызова удаленных процедур (RPC), которые могут добавляться в любую машину сети и немедленно поступать в распоряжение прикладных программ на других машинах сети.

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

**Переносимость.** Код операционной системы должен легко переноситься между процессорами и аппаратными платформами различной архитектуры. Аппаратная платформа включает наряду с типом процессора и способ организации всей аппаратуры компьютера.

Расширяемость позволяет улучшать операционную систему по мере обновления периферийного оборудования. Переносимость дает возможность перемещать всю систему целиком на машину, базирующуюся на обновленном процессоре или аппаратной платформе,

делая при этом по возможности небольшие изменения в коде. Операционные системы разрабатываются либо как переносимые, либо как непереносимые. Ключевым моментом в оценке переносимости является стоимость необходимых изменений.

При написании переносимой операционной системы нужно следовать перечисленным ниже правилам.

Большая часть кода должна быть написана на языке высокого уровня, например, как Unix на языке Си. Код, написанный на ассемблере, не является переносимым, если только он не переносится на машину, обладающую командной совместимостью с исходной машиной.

Необходимо учитывать аппаратную платформу, на которую операционная система должна быть перенесена. Например, система, построенная на 32-битовых адресах, не может быть перенесена на машину с 16-битовыми адресами.

Следует минимизировать или по возможности исключить части кода, которые непосредственно взаимодействуют с аппаратурой. Оставшийся после такой оптимизации аппаратно-зависимый код, который не может быть исключен, локализуется в отдельных модулях (HAL – hardware abstraction layer). Очевидные формы зависимости от аппаратуры включают прямое манипулирование регистрами процессора, аппаратно-зависимыми структурами данных (контекст процесса, дескрипторы страниц и сегментов), обращение к контроллерам периферийного оборудования.

**Надежность и отказоустойчивость.** Система должна быть защищена от отказов оборудования и внутренних ошибок. Действия операционной системы должны быть предсказуемыми. Прикладные процессы не должны иметь возможность наносить вред, как самой операционной системе, так и другим процессам. Надежность обеспечивается применением микроядерной архитектуры.

**Совместимость.** Операционная система должна иметь средства для выполнения прикладных программ, написанных для других операционных систем, или для более ранних версий данной операционной системы. Пользовательский интерфейс должен быть совместим с существующими системами и стандартами. Совместимость операционных систем рассматривается на двух уровнях как двоичная совместимость и совместимость по исходным кодам.

*Двоичная совместимость* достигается совместимостью на уровне команд процессора, системных вызовов и на уровне библиотечных вызовов, если они являются динамически связываемыми.

*Совместимость по исходным кодам* требует наличия соответствующего компилятора в составе программного обеспечения, а также совместимости на уровне библиотек и системных вызовов, при этом необходима перекомпиляция.

Совместимость на уровне исходных текстов важна для разработчиков приложений. Для конечных пользователей практическое значение имеет только двоичная совместимость, благодаря которой один и тот же коммерческий продукт можно использовать в различных операционных средах и на различных машинах. Совместимость зависит от многих факторов, самый важный фактор – архитектура процессора. Если процессор, на который переносится операционная система, использует аналогичный набор команд и тот же диапазон адресов, двоичная совместимость может быть достигнута достаточно просто.

Двоичная совместимость между процессорами, основанными на разных архитектурах, требует написания специальных эмуляторов и использования прикладных сред. Для исполнения кода в гостевой операционной системе требуется: процедура загрузки, адаптированная под формат исполняемого файла; интерпретация команд целевого процессора на гостевом процессоре (если процессорные архитектуры различаются); имитация системных вызовов (вызовов библиотечных функций операционной системы) с использованием заранее написанной библиотеки функций аналогичного назначения.

Примером прикладных сред в Windows NT являются подсистемы для исполнения Win32 приложений, консольных приложений OS/2 и Unix, шестнадцатиразрядных DOS и Win16 приложений. Среда исполнения языка Java, как основной механизм переносимости программ, использует программную эмуляцию архитектуры Java, которая может быть реализована полностью на аппаратном уровне. Для запуска Win32(64) приложений на Linux используется программная среда Wine. Среда Cygwin, наоборот, представляет собой инструмент для переноса программ UNIX в Windows и является библиотекой, которая реализует интерфейс к системным вызовам Win32.

Средствами обеспечения совместимости на уровне исходных кодов являются стандартизированный язык высокого уровня и стандартизированные интерфейсы системных вызовов. Примером стандартизированного процедурного языка программирования является Си (новый стандарт для языка ISO/IEC 9899:2011 вышел 19 декабря 2011 года). Наиболее известным набором стандартов, описывающих интерфейсы между операционной системой и прикладной программой, является POSIX (Portable Operating System Interface for Unix — переносимый интерфейс операционных систем Unix). Стандарт создан для обеспечения совместимости различных UNIX-подобных операционных систем и переносимости прикладных программ на уровне исходного кода. Использование стандарта POSIX (IEEE Std 1003.1-2004, ISO/IEC 9945) позволяет создавать программы в стиле UNIX, которые могут легко переноситься из одной вычислительной системы в другую.

Для семейства операционных систем, основанных на Linux, имеется стандарт совместимости LSB (Linux Standard Base). LSB специфицирует: стандартные библиотеки, несколько команд и утилит в дополнение к стандарту POSIX, структуру иерархии файловой системы, уровни запуска и различные расширения системы X Window System.

**Безопасность.** Операционная система должна обладать средствами защиты. Правила безопасности определяют способы защиты ресурсов пользователя и устанавливают квоты по ресурсам для предотвращения захвата пользователем всех системных ресурсов (например, памяти).

Основы безопасности были заложены стандартом «Критерии оценки надежных компьютерных систем» (Department of DefenseTrusted Computer System Evaluation Criteria, TCSEC, DoD 5200.28-STD, 26 декабря 1985 года). Этот документ часто называют «Оранжевой книгой». Аналогом «Оранжевой книги» является международный стандарт ISO/IEC 15408, опубликованный в 2005 году.

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

Иерархия уровней безопасности, приведенная в «Оранжевой книге», выделяет четыре уровня (D, C, B, A) и 6 классов безопасности внутри уровней (C1, C2, B1, B2, B3, A1). В уровень D попадают системы, оценка которых выявила их несоответствие требованиям безопасности. Уровень C обеспечивает произвольное управление доступом; уровень B – принудительное управление доступом; уровень A – верифицируемую безопасность. В каждом классе от C1 к A1 требования по безопасности расширяются.

Коммерческие системы обычно соответствуют уровню C. Вот некоторые требования безопасности этого уровня: 1) наличие защищенных средств секретного входа, обеспечивающих идентификацию пользователя путем ввода логина и пароля; 2) избирательный (дискреционный) контроль доступа, позволяющий владельцу ресурса определить, кто имеет доступ к ресурсу, и что он может с ним делать, путем предоставляемых прав доступа пользователю или группе пользователей; 3) средства учета и наблюдения, обеспечивающие возможность обнаружить и зафиксировать важные события, связанные с безопасностью: любые попытки создать, получить доступ и удалить системные ресурсы; 4) защита памяти, обеспечивающая инициализацию перед повторным использованием.

Системы уровня B реализуют мандатный (принудительный) контроль доступа. Каждому пользователю присваивается рейтинг защиты, и он может получать доступ к данным только в соответствии с этим рейтингом. Этот уровень, в отличие от уровня C, обеспечивает централизованное управление потоками данных в системе и защищает систему от ошибочного поведения пользователя.

Однако, существует проблема, впервые описанная Батлером Лэмпсоном в 1973 году, известная как «скрытый канал» или проблема ограждения. Лэмпсон показал, что в защищенной системе возможны неконтролируемые принудительной системой безопасности потоки информации. Определяют два вида скрытых каналов.

*Скрытый канал памяти.* Процессы взаимодействуют благодаря тому, что один может прямо или косвенно записывать информацию в некоторую область памяти, а второй считывать. Обычно имеется в виду, что у процессов с разными уровнями безопасности имеется доступ к некоторому ресурсу (например, возможность проверить наличие файла).

*Скрытый канал времени.* Один процесс посылает информацию другому, модулируя своё собственное использование системных ресурсов (например, процессорное время) таким образом, что эта операция воздействует на реальное время отклика, наблюдаемое вторым процессом.

Критерии «Оранжевой книги» требуют, чтобы анализ скрытых каналов памяти был классифицирован как требование для системы класса B2, а анализ скрытых каналов времени как требование для класса B3.

Уровень A является самым высоким уровнем безопасности, он требует в дополнение ко всем требованиям уровня B выполнения формального (математически обоснованного) доказательства соответствия системы требованиям безопасности.

**Производительность.** Система должна обладать настолько хорошим быстродействием и временем реакции, насколько это позволяет аппаратная платформа. На практике удовлетворение рассмотренных выше требований к операционным системам, особенно по надежности и безопасности, приводит к большому потреблению системных ресурсов на функционирование самой операционной системы. Снижение ресурсных требований и повышение производительности является нетривиальной исследовательской задачей при проектировании новых операционных систем.

## Лекция 4. Архитектуры операционных систем

Монолитные системы: модульные и многоуровневые. Клиент-серверные архитектуры: микроядерные и гибридные. Объектно-ориентированные архитектуры. Виртуальные машины: гипервизоры, экзоядра, наноядра.

**Монолитные система** – классическая, наиболее распространённая архитектура ядероперационных систем. Все части монолитного ядра работают в одном адресном пространстве (рис.4.1).

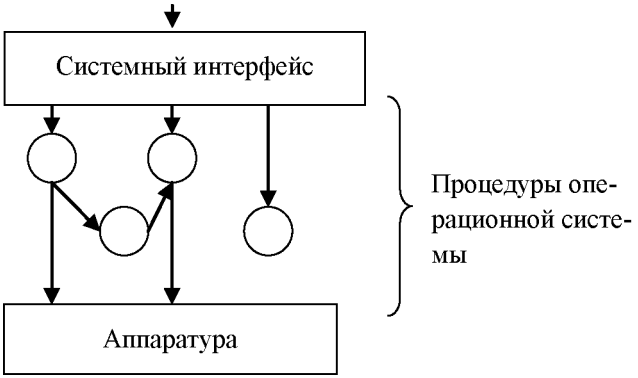



Рис.4.1. Монолитная операционная система

При использовании этой технологии код ядра операционной системы состоит из процедур. Каждая процедура системы имеет определённый интерфейс и может вызывать любую другую процедуру, а также обращаться непосредственно к аппаратуре. Код ядра выполняется в режиме супервизора.

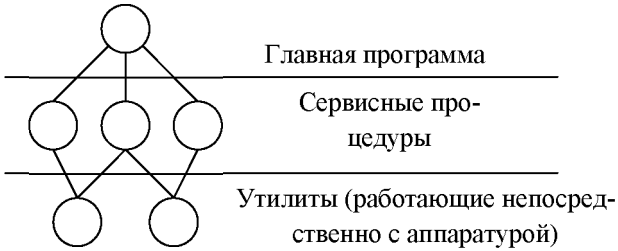



Рис.4.2. Структура ядра монолитной системы

Обычно процедуры операционной системы разделяют на главную программу, сервисные процедуры и утилиты (рис.4.2). Взаимодействие кода пользователя с операционной системой происходит посредством системных вызовов. В отличие от обычных вызовов подпрограмм, в системном вызове происходит передача управления и данных между кодом режима пользователя и кодом режима супер-

визора (ядра). При обращении к системным вызовам главная программа помещает параметры вызова в строго определённое место (регистры, стек) и переключает машину из режима пользователя в режим ядра (например, вызывая специальное программные прерывания).

В режиме ядра по сформированному коду системного вызова вычисляется адрес сервисной процедуры в пространстве ядра, и выполняется вызов. У каждого системного вызова имеется своя сервисная процедура, а утилиты выполняют функции, нужные нескольким сервисным процедурам.

В классической монолитной архитектуре все процедуры операционной системы собраны в один объектный файл. *Модульное ядро* – современная, усовершенствованная модификация архитектуры монолитных ядер, в которой код ядра разделяется на отдельно компилируемые и загружаемые модули.

Модульные ядра предоставляют механизм загрузки модулей ядра, поддерживающих аппаратное обеспечение (например, драйверов). Загрузка модулей может быть как динамической (без перезагрузки операционной системы), так и статической (выполняемой при перезагрузке). Большинство современных ядер, такие как OpenVMS, Linux, FreeBSD, NetBSD и Solaris, позволяют во время работы динамически (по необходимости) подгружать и выгружать модули, выполняющие часть функций ядра.

Модули ядра работают в адресном пространстве ядра и могут пользоваться всеми функциями, предоставляемыми ядром. Поэтому модульные ядра продолжают оставаться монолитными. Модульность ядра осуществляется на уровне бинарного образа, а не на архитектурном уровне ядра, так как динамически подгружаемые модули загружаются в адресное пространство ядра и в дальнейшем работают как его часть.

Имеется вариант монолитной архитектуры, когда код ядра операционной системы строится как иерархия уровней. Такая архитектура ядра называется *многоуровневой*. Уровни образуются группами функций, каждый уровень взаимодействует только со своими непосредственными соседями через строго определенные интерфейсы. Многоуровневая архитектура была предложена ЭдсгеромДейкстра в проекте пакетной системы THE в 1968 году.

В системе THE функции ядра распределялись по уровням следующим образом. Уровень 0 занимался распределением времени процессора, реализуя базовую многозадачность. Уровень 1 осуществлял управление памятью через механизм виртуальной памяти. Уровень 2 отвечал за связь между консолью оператора и процессами. Уровень 3 управлял буферизацией и взаимодействием с устройствами ввода-вывода. На уровне 4 работали программы пользователя. Уровень 5 – это пользователь системы.

Многоуровневая архитектура операционной системы THE, в основном, служила средством разработки. Однако каждый уровень может быть наделён привилегиями и выполнять системный вызов с контролем параметров для обращения к другому уровню. В данной архитектуре уровни называются *кольцами защиты*. Многоуровневая архитектура с кольцами защиты была реализована в операционной системе Multics.

Проблемой многоуровневой архитектуры является множественность и размытость интерфейсов между слоями, так как сложно выполнить однозначную группировку функций по уровням. Основной проблемой монолитной архитектуры является низкая надежность. Она вызвана большим объемом критического к сбою кода, который сложно сопровождать. Также из-за того, что весь код работает в общем адресном пространстве, ошибка в одной из процедур может привести к повреждению разделяемых всеми процедурами данных и выходу системы из строя. Подобная ситуация может быть и при злонамеренном повреждении кода через модифицированные злоумышленниками подгружаемые модули ядра.

Монолитная архитектура, между тем, имеет преимущества с точки зрения удобства организации взаимодействия между частями операционной системы. Оно организуется проще, так как может использовать глобальные структуры данных системы. По этой же причине и по причине отсутствия переключения режима процессора взаимодействие происходит быстрее. Это свойство существенно, например, для эффективной реализации файловых систем.

**Клиент – серверная архитектура.** Эта архитектурная модель предполагает наличие программного компонента потребителя сервиса, который называется *клиентом*, и программного компонента поставщика сервиса (*сервера*). Взаимодействие клиента и сервера

стандартизировано. Инициатором обмена является клиент, который посылает запрос серверу, находящемуся в состоянии ожидания запроса.

Применительно к структурированию операционной системы подход состоит в разделении ее на несколько процессов-серверов, каждый из которых выполняет отдельный набор сервисных функций, например, управление памятью, планирование процессов. Серверные процессы работают в пользовательском режиме. Микроядро работает в привилегированном режиме и выполняет функции доставки сообщений нужному серверу и передачи результатов клиенту (рис. 4.3).

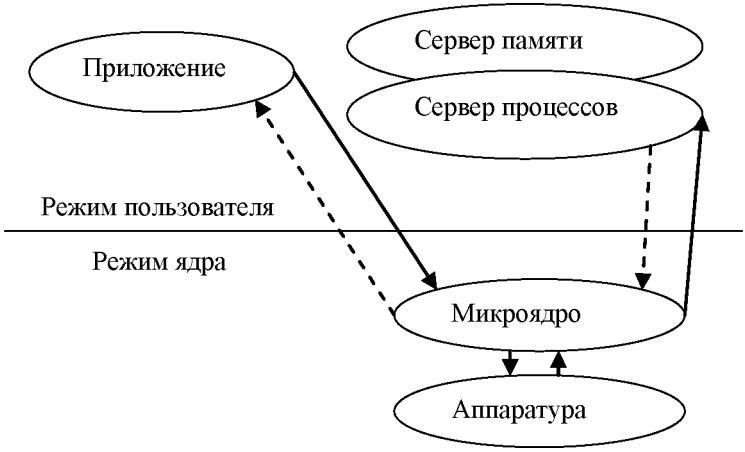



Рис.4.3. Клиент-серверная архитектура

В представленной теоретической модели существует проблема: для реализации своих функций серверы должны иметь доступ к аппаратуре. В зависимости от того как разрешается эта архитектурная проблема различают микроядерную и гибридную реализацию клиент-серверного подхода к построению ядра операционной системы.

Архитектурный подход при проектировании *микроядра* заключается в помещении в ядро только тех функций, которые можно исполнить в режиме супервизора (привилегированном режиме). Все остальные функции распределяются между серверами пользовательского режима. При этом используется парадигма разделения механизма и политики. Ядро отвечает за механизм реализации некоторо-

го решения по управлению ресурсами, а сервер пользовательского режима отвечает за политику, то есть отвечает за принятие решений.

*Пример 1.* Очевидно, что запуск процесса требует привилегированного режима, так как связан с манипулированием аппаратно зависимыми структурами данных и передачей управления между процессами. Сам запуск процесса требует доступа к аппаратуре и выполняется ядром. Решение о приоритетах процессов, дисциплине постановки их в очередь может принимать работающий вне ядра планировщик. Пользователь может использовать тот планировщик, который подходит для его прикладной задачи.

*Пример 2.* Для управления памятью в системе имеется планировщик, который определяет стратегию замещения страниц и работает вне ядра как сервер пользовательского режима (pager). Все остальные аппаратно зависимые функции по управлению памятью реализуются в ядре.

*Пример 3.* У драйверов устройств можно выделить общий интерфейс, работающий в режимах ядра. В его функции входит работа с аппаратными прерываниями и доступ к управляющим регистрам устройств ввода-вывода. Драйвер, работающий в режиме пользователя, например, драйвер СУБД, может включать оптимизацию под конкретный способ доступа к диску, работая на уровне сегментов диска, а не с файлами.

Преимуществом микроядерной архитектуры является высокая надежность и гибкость. Надежность обеспечивается тем, что возможные сбои локализуются в сервере режима пользователя, и не затрагивают другие сервисы и ядро. Восстановление осуществляется перезапуском отказавшего сервиса без необходимости остановки всех процессов и перезапуска операционной системы. Само ядро, в силу малого объема кода, легко проанализировать на наличие ошибок. Например, размер кода ядра L4 составляет 14 килобайт и содержит 7 системных вызовов.

Микроядерная архитектура операционной системы вносит дополнительные накладные расходы, связанные с обменом сообщениями, что отрицательно влияет на производительность. Для того, чтобы микроядерная операционная система по скорости не уступала операционным системам на базе монолитного ядра, требуется обеспечить правильное разбиение системы на компоненты, стараясь ми-

нимизировать взаимодействие между ними. Таким образом, основная сложность при создании микроядерных операционных систем – необходимость очень аккуратного проектирования.

Примерами микроядер являются Mach, L4, Minix, QNX. Наиболее известная операционная система, основанная на микроядре – Symbian OS.

*Гибридное ядро.* Этот архитектурный подход заключается в размещении аппаратно зависимых сервисов в режиме ядра (управление процессами, памятью, вводом-выводом; безопасность). При этом сохраняется ориентированный на сообщения механизм взаимодействия клиент-серверной архитектуры.

Основным преимуществом подхода является значительное повышение быстродействия системы. Это происходит потому, что не требуется частого переключения из режима ядра в режим пользователя при передаче сообщений между серверами, реализующими сложный системный вызов.

Недостатком является некоторая потеря гибкости и надежности. Такие известные семейства операционных систем, как Windows NT и Mac OS X построены на основе гибридных ядер, соответственно, NT kernel и XNU.

Оригинальное решение проблемы производительности с сохранением высокой надежности и гибкости микроядерного подхода предложено Microsoft и развивается в рамках проекта операционной системы Singularity. В данной операционной системе серверные процессы и ядро исполняются на одном уровне привилегий. Защита процессов производится не путем организации аппаратно-защищенных адресных пространств, а путем использования промежуточного языка и его верификации перед компиляцией в код процессора.

**Объектно-ориентированные архитектуры операционных систем.** Преимущества объектной модели (инкапсуляция, наследование, полиморфизм) используются также при написании операционных систем.

Можно выделить три способа применения объектов. Код ядра пишется на процедурном языке, а объекты моделируются средствами языка, например, Си. Такой подход используется в ядре Windows NT. Так пользователю доступны описатели (типа HANDLE) для работы с объектами ядра. Имеются полиморфные функции (например,

CloseHandle, выполняющая очистку произвольного объекта ядра). Сокрытие реализации объектов ядра выполняется аппаратными механизмами защиты.

Код операционной системы может быть реализован непосредственно на объектно-ориентированном языке программирования. Например, язык Objective C используется в операционных системах Mac OS X и iOS.

Шире всего объектно-ориентированные технологии применяются в программном обеспечении промежуточного уровня - программной прослойке между операционными системами и прикладным программным обеспечением. Примером данного вида программного обеспечения Microsoft является архитектура COM (component object model). На основе COM были реализованы технологии: Microsoft OLE Automation, ActiveX, DCOM, COM+, DirectX. В COM используется IDL для описания интерфейсов компонента и язык реализации C++. Кроссплатформенным аналогом технологии COM является архитектура CORBA, позволяющая организовать взаимодействие систем, написанных на разных языках программирования и работающих на разных операционных системах.

**Виртуальная машина.** Концепция виртуальной машины была разработана в конце 60х годов в исследовательском центре IBM (Кембридж, Массачусетс) в процессе работы над операционной системой CP/CMS. Последующая коммерческая реализация VM/CMS используется в компьютерах IBM до настоящего времени.

Проект изначально разрабатывался как альтернатива OS/360. Эта система разделения времени обеспечивает с одной стороны многозадачность, а с другой - расширенную машину. Эти функции можно полностью разделить. Монитор VM работает с оборудованием и обеспечивает многозадачность. Но представленная монитором машина не является расширенной: она полностью идентична оборудованию. На ней может работать любая операционная система, которая работает на аппаратуре, в том числе сама VM. Эта операционная система, в свою очередь, реализует расширенную машину. Такая архитектура оказывается проще и надежнее обычной многозадачной операционной системы OS/360.

На использовании виртуализации основаны интенсивно развивающиеся в настоящее время «облачные» вычисления. Например, одним из основных решений для сглаживания неравномерности

нагрузки «облачного» поставщика услуг является размещение слоя серверной виртуализации между слоем программных услуг и аппаратным обеспечением. Виртуализация позволяет реализовать балансировку нагрузки посредством программного распределения виртуальных серверов по реальным компьютерам без остановки вычислений.

*Экзоядро* развивает идею виртуальной машины. Основная идея операционной системы на основе экзоядра состоит в том, что ядро должно выполнять лишь функции координатора для небольших процессов, связанных только одним ограничением: экзоядро должно иметь возможность гарантировать безопасное выделение и освобождение ресурсов оборудования. В отличие от операционных систем на основе микроядра, экзоядра обеспечивают большую эффективность за счет отсутствия необходимости в переключении между процессами при каждом обращении к оборудованию.

*Наноядро*— архитектура ядра операционной системы компьютеров, в рамках которой крайне упрощённое ядро выполняет лишь одну задачу — обработку аппаратных прерываний, генерируемых устройствами компьютера. Наиболее часто в современных компьютерах наноядра используются для виртуализации аппаратного обеспечения реальных компьютеров или для обеспечения возможности запуска «старой» операционной системы на новом, несовместимом аппаратном обеспечении без её полного переписывания.

## Экзаменационные вопросы по разделу II

1. Функциональные требования, предъявляемые к операционным системам, и способы их реализации. Расширяемость. Переносимость. Надежность. Совместимость. Безопасность. Производительность.
2. Основные архитектуры операционных систем: монолитные, многоуровневые, микроядро, объектно-ориентированные, виртуальные машины.

## РАЗДЕЛ III. УПРАВЛЕНИЕ ПРОЦЕССАМИ

### Лекция 5. Понятие процесса, методы планирования

Определение процесса. Диаграмма состояний процесса. Информационные структуры процесса: контекст и дескриптор. Виды алгоритмов планирования. Виды многозадачности. Потоки исполнения.

**Определение процесса.** Процесс - это выполнение последовательной программы на процессоре компьютера. Компьютерная программа является пассивной совокупностью инструкций, в то время как процесс представляет собой непосредственное выполнение этих инструкций.

С точки зрения операционной системы процесс является, с одной стороны, единицей исполнения (для процесса требуется выделять время процессора). С другой стороны, процесс рассматривается как заявка на потребление системных ресурсов (с ним связана область памяти, открытые файлы и другие ресурсы).

Подсистема управления процессами многозадачной операционной системы планирует выполнение процессов, то есть распределяет процессорное время между несколькими процессами, а также занимается созданием и уничтожением процессов, обеспечивает процессы необходимыми системными ресурсами, поддерживает взаимодействие между процессами.

**Диаграмма состояний процесса.** В многозадачных операционных системах процесс может находиться в одном из трех основных состояний. Это «выполнение», «ожидание», «готовность».

«Выполнение» - активное состояние процесса. В данном состоянии процесс обладает всеми необходимыми ресурсами и непосредственно выполняется процессором.

«Ожидание» - пассивное состояние процесса. Процесс заблокирован и не может выполняться по своим внутренним причинам. Такими причинами могут являться: ожидание завершения операции ввода-вывода; получение сообщения от другого процесса; освобождение необходимого для продолжения вычислений ресурса.

«Готовность» - также пассивное состояние процесса. В этом состоянии процесс заблокирован в связи с внешними причинами, по инициативе операционной системы. Процесс имеет все требуемые

для выполнения ресурсы, однако процессор занят выполнением другого процесса.

В ходе своего выполнения каждый процесс переходит из одного состояния в другое в соответствии с алгоритмом планирования процессов, реализуемым в данной операционной системе (рис.5.1).


Рис. 5.1. Диаграмма состояния процесса

В состоянии «выполнение» в однопроцессорной системе может находиться только один процесс. В каждом из состояний «ожидание» и «готовность» - несколько процессов. Эти процессы образуют очереди. Исполнение процесса начинается с состояния «готовность». В данном состоянии он находится в очереди планировщика процессов операционной системы. При активизации процесс переходит в состояние «выполнение» и находится в нем до тех пор, пока: ему не потребуется ждать некоторого события; он сам не освободит процессор; он будет принудительно вытеснен планировщиком.

Переходя в состояние «ожидания», процесс помещается в очередь, связанную с конкретным событием, которое он ожидает. Например, процесс может попасть в очередь процессов, ожидающих завершения ввода-вывода.

Если процесс вытесняется или добровольно отдает управление планировщику, он попадает в состояние «готовность» и помещается в очередь планировщика. В это же состояние процесс переходит из состояния «ожидание» после того, как произойдет ожидаемое событие.

**Информационные структуры процесса.** Информационные структуры, которые используются для управления исполнением процессов, называются *контекст* и *дескриптор*. Программный код только тогда начнет выполняться, когда для него операционной системой будет создан процесс. Создание процесса состоит из трех этапов: создания дескриптора и контекста процесса; включения де-

скриптора нового процесса в очередь готовых процессов; загрузки кодового сегмента процесса в оперативную память.

На протяжении существования процесса его выполнение может быть многократно прервано и продолжено. Для того, чтобы возобновить выполнение процесса, необходимо восстановить состояние его операционной среды. Состояние операционной среды состоит из значений регистров и программного счетчика, режима работы процессора, указателей на открытые файлы, информации о незавершенных операциях ввода-вывода, кодов ошибок, выполняемых данным процессом системных вызовов. Эта информация называется *контекстом процесса*. Контекст является зависимой от аппаратуры структурой данных.

Операционной системе для реализации планирования процессов требуется дополнительная информация: идентификатор процесса, состояние процесса, данные о степени привилегированности процесса, данные о нахождении процесса в очередях, указатель на контекст процесса. Эта информация называется *дескриптором процесса*. Содержание информационных полей дескриптора определяется разработчиком операционной системы, зависит от особенностей алгоритма планирования и не зависит от аппаратуры.

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

**Виды алгоритмов планирования.** Управление процессами включает в себя решение следующих задач: определение момента времени для смены выполняемого процесса; выбор процесса на выполнение из очереди готовых процессов; переключение контекстов между вновь запускаемым и снимаемым с исполнения процессом. Последняя задача решается аппаратно. То, каким образом решаются первые две задачи, определяется алгоритмом планирования.

Существует множество различных алгоритмов планирования процессов, по-разному решающих вышеперечисленные задачи, преследующих различные цели и обеспечивающих различное качество мультипрограммирования. Однако на планирование можно повлиять двумя способами. Можно управлять длительностью исполнения

процессов, воздействуя на переход между состоянием «выполнение» - «готовность». Алгоритмы, использующие данный подход – это алгоритмы, основанные на квантовании. Можно управлять выбором готового процесса на исполнение, воздействуя на переход «готовность» - «выполнение». Этот подход используют алгоритмы, основанные на приоритетах.

В соответствии с алгоритмами, основанными на квантовании, смена активного процесса происходит, если: процесс завершился и покинул систему; произошла ошибка; процесс перешел в состояние «ожидание»; исчерпан квант процессорного времени для данного процесса. Длительность отводимых на исполнение промежутков времени (квантов времени) определяется системным таймером, который периодически шлет запросы прерывания центральному процессору. Обработчик прерывания передает управление планировщику операционной системы, который и выполняет переключение контекстов между процессами.

Процесс, который исчерпал свой квант, переводится в состояние «готовность» до тех пор, пока ему будет предоставлен новый квант процессорного времени. На выполнение в соответствии с определенным правилом выбирается новый процесс из очереди готовых процессов. Таким образом, ни один процесс не занимает процессор надолго, поэтому квантование широко используется в системах разделения времени.

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

Другая группа алгоритмов использует понятие приоритет. Приоритет – это число, характеризующее степень привилегированности процесса. Приоритет может выражаться натуральными, целыми или вещественными числами. Чем выше привилегии процесса, тем меньше времени он будет проводить в очередях. Приоритет может

назначаться директивно администратором системы в зависимости от важности работы или внесенной платы, либо вычисляться самой операционной системой по определенным правилам. Он может оставаться фиксированным либо изменяться во времени в соответствии с некоторым законом. В последнем случае приоритеты называются динамическими.

Существует две разновидности приоритетных алгоритмов: алгоритмы, использующие относительные приоритеты, и алгоритмы, использующие абсолютные приоритеты. В обоих случаях выбор процесса на выполнение из очереди готовых осуществляется одинаково: выбирается процесс, имеющий наивысший приоритет. Поразному решается проблема определения момента смены активного процесса. В системах с относительными приоритетами активный процесс выполняется до тех пор, пока он сам не покинет процессор, перейдя в состояние «ожидание» (произойдет ошибка или процесс завершится). В системах с абсолютными приоритетами выполнение активного процесса прерывается, если в очереди готовых процессов появился процесс, приоритет которого выше приоритета активного процесса. В этом случае прерванный процесс переходит в состояние готовности.

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

**Виды многозадачности.** В зависимости от того, кто является инициатором перехода из состояния «выполнение» в состояние «готовность» различают три вида многозадачности.

*Вытесняющая многозадачность* (preemptive multitasking) - это такой способ, при котором решение о переключении процессора с выполнения одного процесса на выполнение другого процесса принимается планировщиком операционной системы, а не самой активной задачей.

*Невытесняющая многозадачность* (non-preemptive multitasking) - это способ планирования процессов, при котором операционная система не является инициатором переключения контекста с текущего процесса на новый процесс. Такое переключение может осу-

ществляться по инициативе пользователя с использованием программы-переключателя на фоновый процесс или по инициативе исполняющегося процесса.

Вид многозадачности называется *кооперативной* или *совместной многозадачностью* (cooperative multitasking), когда текущий процесс самостоятельно отдает управление планировщику операционной системы для того, чтобы тот выбрал из очереди другой, готовый к выполнению процесс.

Основным различием между вытесняющей и кооперативной многозадачностью является степень централизации механизма планирования задач. При вытесняющей многозадачности механизм планирования задач целиком сосредоточен в операционной системе. Программист пишет свое приложение, не заботясь о том, что оно будет выполняться параллельно с другими задачами. Операционная система выполняет следующие функции: определяет момент снятия с выполнения активной задачи; запоминает ее контекст; выбирает из очереди готовых задач следующую и запускает ее на выполнение, загружая ее контекст.

При кооперативной многозадачности механизм планирования распределен между системой и прикладными программами. Прикладная программа, получив управление от операционной системы, сама определяет момент завершения своей очередной итерации и передает управление операционной системе с помощью какого-либо системного вызова, а операционная система формирует очереди задач и выбирает следующую задачу на выполнение.

Для пользователей это означает, что управление системой теряется на произвольный период времени. Если приложение тратит слишком много времени на выполнение какой-либо работы, например, на форматирование диска, пользователь не может переключиться с этой задачи на другую задачу. Эта ситуация нежелательна, так как пользователи обычно не хотят долго ждать, когда машина завершит свою задачу. Поэтому разработчики приложений для кооперативной операционной системы, возлагая на себя функции планировщика, должны создавать приложения так, чтобы они выполняли свои задачи небольшими частями. Например, программа форматирования может отформатировать одну дорожку диска и вернуть управление системе. После выполнения других задач система воз-

вратит управление программе форматирования, чтобы та отформатировала следующую дорожку. Подобный метод разделения времени между задачами затрудняет разработку программ и предъявляет повышенные требования к квалификации программиста. Если произойдет зацикливание потока управления внутри задачи, это приведет к общему краху системы. В системах с вытесняющей многозадачностью такие ситуации, как правило, исключены, так как центральный планирующий механизм снимет зависшую задачу с выполнения.

Однако распределение функций планировщика между системой и приложениями не всегда является недостатком, а при определенных условиях может быть и преимуществом. Кооперативная многозадачность дает возможность разработчику приложений самому проектировать алгоритм планирования, наиболее подходящий для решаемой задачи. Так как разработчик сам определяет в программе момент отдачи управления, то исключаются нерациональные прерывания программ. Кроме того, легко разрешаются проблемы совместного использования данных: задача во время каждой итерации использует их монопольно. На протяжении итерации никакой другой процесс не изменит данные. Существенным преимуществом систем с кооперативной многозадачностью является более высокая скорость переключения с задачи на задачу.

Кооперативная многозадачность реализована в Windows до версии 3.11 включительно, в Mac OS до версии Mac OS X, а также во многих ранних UNIX-подобных операционных системах.

**Потоки исполнения.** Важным аспектом реализации современных многозадачных операционных систем является наличие потоков выполнения или нитей (thread). Как отмечалось, процессы - это сущности, являющиеся единицами планирования и единицами выделения ресурсов. Концепция потоков позволяет определить не одну, а несколько единиц планирования внутри пула ресурсов, выделенных процессу.

Каждый поток исполнения имеет собственный программный счетчик, стек, регистры, состояние. Потоки разделяют адресное пространство процесса, в котором они исполняются; глобальные переменные; открытые файлы; объекты, адресуемые через таблицу описателей объектов ядра процесса; статистическую информацию.

Существует несколько моделей реализации многопоточности в зависимости от того, как системные потоки, реализуемые в ядре операционной системы, соотносятся с пользовательскими потоками внутри процесса.

Потоки выполнения, созданные пользователем в *модели потоков* ядра (1:1), *соответствуют* потокам ядра. Это простейший возможный вариант реализации многопоточности.

В *модели пользовательских потоков* (N:1) предполагается, что все потоки выполнения уровня пользователя отображаются на поток уровня ядра, и ядро ничего не знает о составе прикладных потоков выполнения. При таком подходе переключение контекста может быть сделано очень быстро. Однако главным недостатком является отсутствие ускорения на многопроцессорных компьютерах, потому что только один поток выполнения ядра может быть запланирован на процессор.

В *гибридной модели* (M:N) некоторое число M прикладных потоков выполнения отображаются на некоторое число N сущностей ядра или «виртуальных процессоров». Модель является компромиссной между моделью уровня ядра (1:1) и моделью уровня пользователя.

Специальные сущности, используемые для реализации моделей многопоточности (N:1) – это файберы (fiber- волокно). В Windows NT файберы организуются на базе потока. Особенностью является принудительное, выполняемое программистом, переключение контекста между файберами одного потока. Таким образом, на основе файберов могут быть реализованы преимущества кооперативной многозадачности внутри одного процесса.

Есть много различных, несовместимых друг с другом реализаций потоков. К ним относятся как реализации на уровне ядра, так и реализации на пользовательском уровне. Чаще всего они придерживаются более или менее близко стандарта интерфейса POSIX Threads.

## Лекция 6. Многозадачность в операционной системе Windows

Многозадачность на уровне процессов и на уровне потоков: группы процессов, планирование в режиме пользователя, пулы потоков, файберы. Планировщик: классы и уровни приоритета, структуры данных планировщика, переключение контекста, выбор готового потока на исполнение. Инверсия приоритета, способы преодоления. Многопроцессорная обработка.

**Многозадачность.** Рассмотрим некоторые особенности реализации многозадачности на примере операционной системы Windows. В Windows используются все современные технологии управления многозадачностью для однопроцессорных и многопроцессорных систем. Подробную информацию можно найти в библиотеке разработчика Microsoft Developer Network в разделе About Processes and Threads.

В Windows имеются два основных способа использования многозадачности: многозадачность на уровне процессов и многозадачность на уровне потоков.

Многозадачность на уровне процессов используется, если требуется изоляция адресного пространства и ресурсов. Примером является реализация системных служб Windows в виде автономных процессов. Такая реализация предполагает наличие управляющего процесса, который запускает рабочие процессы, выполняет диагностику их состояния и при необходимости перезапускает. Крах системной службы не приводит к краху других процессов за счет изоляции ресурсов.

Для управления приложением, состоящим из нескольких процессов, в Windows предусмотрены специальные объекты – работы (job objects). Работа позволяет рассматривать группу процессов как целое. Операция, применяемая к работе, влияет на все процессы, связанные с ней. Например, можно установить групповой приоритет, размер рабочей области, или одновременно удалить все процессы, ассоциированные с работой.

Многозадачность на уровне потоков исполнения может использоваться при выполнении следующих задач.

1. Управление вводом из нескольких окон. Примером является оконный менеджер, который для каждой папки создаёт отдельный поток.

2. Управление вводом из нескольких коммуникационных устройств. Данный подход применяется в многопоточных серверах в интернете. Каждого подключенного клиента обслуживает отдельный поток. Использование потоков позволяет упростить архитектуру программы сервера за счет применения блокирующих операций ввода-вывода с сохранением эффективности приложений, построенных на асинхронных операциях ввода-вывода.

3. Разделение задач различного приоритета. Может потребоваться для выполнения высокоприоритетным потоком критичных ко времени задач. Например, такой поток может заниматься сохранением телеметрической информации. Основной поток может осуществлять графическое отображение этой информации на терминале.

4. Сохранение интерактивности приложения при выполнении фоновой задачи. При выполнении длительных расчетов возникает необходимость в прерывании вычислений, ввода или корректировки параметров. Для этого вычисления организуют в фоновом низкоприоритетном потоке. Когда пользователь активирует элементы графического интерфейса, более приоритетный поток GUI немедленно прерывает вычисления и обрабатывает команды пользователя.

5. Использование преимуществ многопроцессорных систем для ускорения вычислений. Требуется разделить программу на несколько потоков, если разрабатывается ресурсоемкое приложения, которому необходимо использовать все вычислительные ресурсы современных многоядерных и многопроцессорных систем. Обычная однопоточная программа не сможет выполняться на нескольких ядрах и эффективно использовать возможности современной аппаратуры.

Реализация многозадачности с использованием одного процесса и нескольких потоков, если не требуется изоляции ресурсов, предпочтительна по следующим соображениям: происходит более быстрое переключение контекста между потоками одного процесса, по сравнению с переключением контекстов между потоками разных процессов; потоки одного процесса разделяют глобальные переменные; потоки одного процесса разделяют описатели системных ресурсов.

Операционная система Windows обеспечивает альтернативные потокам методы многозадачности, например, асинхронный ввод-вывод, асинхронные вызовы процедур (APC) и другие.

Как отмечалось ранее, в Windows используется модель потоков ядра (1:1). Поэтому при использовании потоков рекомендуют использовать как можно меньше потоков для одного приложения, так как требуются ресурсы для хранения контекста потока, для отслеживания нескольких активных потоков и, обычно, более сложная синхронизация.

В современных версиях системы Windows NT поддерживается гибридная потоковая модель (M:N). В Windows она называется планированием в режиме пользователя (UMS - user mode scheduling). Модель следует использовать при необходимости создавать большое число потоков одновременно. При этом сочетаются преимущества быстрого переключения между такими потоками и отсутствие блокировки всех UMS-потоков, когда один из них выполняет блокирующую операцию ядра.

Модель пользовательских потоков (N:1) реализуется при помощи файберов (fiber). Данная модель может использоваться, если требуется реализовать самостоятельно планировщик или для переноса в Windows приложений с кооперативной многозадачностью без значительной переделки кода.

В Windows также имеется встроенная поддержка парадигмы параллельного программирования «управляющий - рабочие». Она называется потоковые пулы (thread pools). Потоковый пул – это группа потоков, которые совместно обрабатывают очередь из заданий, сообщений таймера, запланированных асинхронных событий (например, завершение ввода-вывода). При этом достигается уменьшение числа необходимых потоков и не требуется ручное управление очередью заданий.

**Планировщик.** В документации разработчика не описывается конкретный алгоритм планирования, так как он разный в разных версиях операционной системы Windows. Однако имеется возможность адаптировать конкретный планировщик под нужды приложения. Повлиять на алгоритм планирования можно путем управления классом и уровнем приоритета (priority class, priority level).

Класс указывается при создании процесса с использованием функции CreateProcess. Существует 6 классов приоритета:

IDLE_PRIORITY_CLASS – самый низкий класс приоритета, его имеют хранители экрана, средства сбора диагностики, средства индексирования и другие процессы фонового режима;

BELOW_NORMAL_PRIORITY_CLASS – приоритет ниже, чем приоритет по умолчанию;

NORMAL_PRIORITY_CLASS – приоритет по умолчанию;

ABOVE_NORMAL_PRIORITY_CLASS – приоритет выше, чем по умолчанию;

HIGH_PRIORITY_CLASS – приоритет процессов, непосредственно работающих с оборудованием, является приоритетом реального времени;

REALTIME_PRIORITY_CLASS – еще приоритет реального времени, более приоритетен, чем системные потоки, работающие с диском, клавиатурой и мышью.

Класс приоритета определяется с помощью функции GetPriorityClass и задается с помощью функции SetPriorityClass.

Типичная ситуация изменения приоритета возникает, когда процессу нужно гарантировать непрерывное выполнение некоторой операции. Для этого кратковременно приоритет повышается, затем понижается.

Внутри процесса устанавливаются относительные приоритеты для его потоков. Они называются уровнями приоритета или приоритетами потока:

THREAD_PRIORITY_IDLE – минимальный приоритет для фоновых потоков простоя;

THREAD_PRIORITY_LOWEST – более высокий приоритет для потоков простоя;

THREAD_PRIORITY_BELOW_NORMAL – приоритет для менее приоритетных рабочих потоков;

THREAD_PRIORITY_NORMAL – приоритет по умолчанию;

THREAD_PRIORITY_ABOVE _NORMAL – приоритет для более приоритетных рабочих потоков;

THREAD_PRIORITY_HIGHEST – приоритет реального времени;

THREAD_PRIORITY_TIME_CRITICAL – наивысший приоритет реального времени.

Уровень приоритета определяется с помощью функции GetThreadPriority и задается с помощью функции SetThreadPriority. Всем потокам по умолчанию назначается нормальный уровень приоритета. Если возникает необходимость задать уровень приоритета при создании потока, то используется следующая последовательность вызовов. С использованием CreateThread создается поток в приостановленном состоянии с флагом CREATE_SUSPENDED. Далее устанавливается нужный уровень приоритета вызовом SetThreadPriority. Затем поток переводится в готовое состояние вызовом ResumeThread.

Единицами планирования для диспетчера являются именно потоки, а не процессы. Для вычисления приоритета планирования, который называется базовым приоритетом (base priority) по специальным правилам выполняется комбинирование класса и уровня. Операционная система Windows использует 32 базовых приоритета (от 0 до 31). Старшие приоритеты от 16 до 31 относятся к приоритетам реального времени. Младшие приоритеты относятся к приоритетам разделения времени. Приоритет 0 не назначается потокам пользователя, он зарезервирован за потоком обнуления страниц. Этот поток отвечает за очистку страниц, которые переходят от одного процесса к другому, в оперативной памяти. Это обеспечивает защиту объектов, согласно требованию класса безопасности С2.

В планировщике имеется 32 очереди потоков, в которые они попадают согласно своим приоритетам (рис. 6.1). Обслуживание, то есть назначение квантов процессорного времени внутри каждой очереди, осуществляется по принципу карусели (round-robin). Кванты времени получают потоки, находящиеся в непустой очереди наивысшего приоритета. Для переключения контекста между потоками Windows использует следующую последовательность шагов:

- сохранить контекст только что завершившегося потока;
- поместить этот поток в очередь соответствующего приоритета;
- найти очередь наибольшего приоритета, содержащую готовые потоки;
- удалить дескриптор потока из головы этой очереди, загрузить контекст, приступить к исполнению.

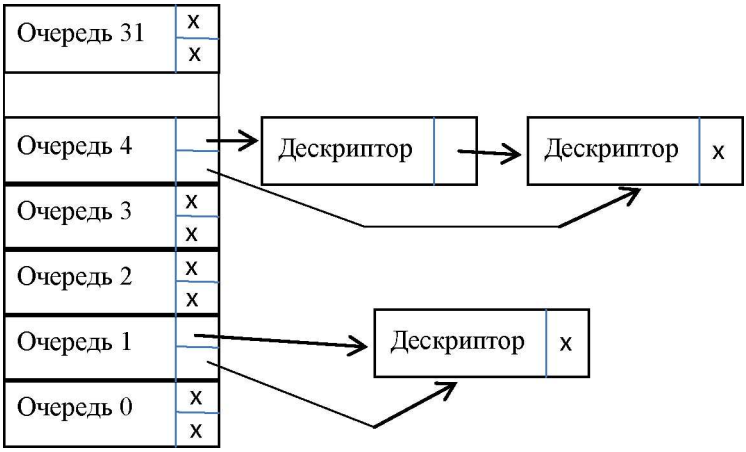



Рис. 6.1. Схема очередей планировщика Windows

Некоторые потоки не находятся в структурах данных планировщика, то есть не являются готовыми. Такими потоками являются потоки, созданные флагом SUSPENDED; остановленные командой SuspendThread; ожидающие события синхронизации (например, в функции WaitForSingleObject) или завершения ввода-вывода.

Причины вытеснения текущего потока в планировщике Windows - истечение кванта времени; появление более приоритетного готового потока; переход исполняющегося потока к ожиданию события или завершения ввода-вывода.

Согласно описанной выше процедуре планирования низкоприоритетные потоки не должны получать обслуживание при наличии более приоритетных потоков. Чтобы предотвратить данную нежелательную ситуацию для потоков разделения времени вводится динамический приоритет. Динамический приоритет используется для продвижения потоков при наличии более приоритетных. Планировщик Windows кратковременно повышает приоритеты простаивающих готовых потоков. Процедура повышения приоритета (priority boost) выполняется в случае, когда: процесс, содержащий поток, переходит на передний план; окно процесса получает событие от мыши и клавиатуры; наступило событие, которое ожидал поток или завершился ввод-вывод. В любом случае динамический приоритет не может быть меньше базового приоритета. По истечению каждого

кванта динамический приоритет уменьшается на 1 до тех пор, пока не достигнет базового приоритета. Повышенный приоритет (priority boost) получают потоки с базовым приоритетом, не превышающим 15. Также в Windows имеется возможность изменения длительности квантов времени, назначаемых потокам.

**Инверсия приоритетов.** Наличие приоритетов может привести к неявной блокировке, когда более высокоприоритетный поток зависит от менее приоритетного потока. Например, ждет освобождения мьютекса, захваченного потоком.

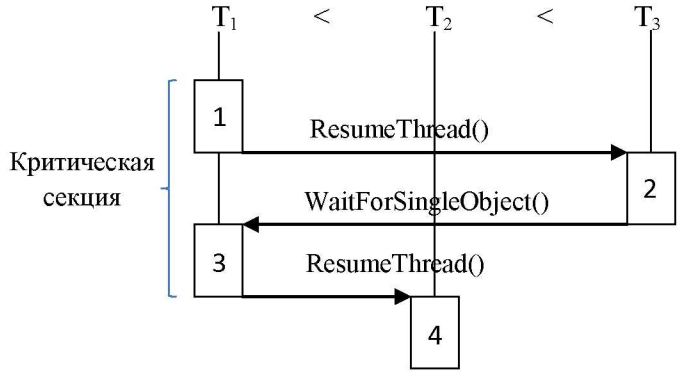



Рис. 6.2. Возникновение инверсии приоритетов

На рис.6.2 показан сценарий возникновения инверсии приоритетов на примере трех потоков. Поток T1 имеет наименьший приоритет, поток T3 наибольший приоритет, поток T2 имеет промежуточный приоритет. Опишем последовательность возникновения инверсии.

Шаг 1. Поток T1 выполняет код в критической секции.

Шаг 2. Появляется поток T3 с наивысшим приоритетом в готовом состоянии. Следовательно, поток T1 вытесняется.

Шаг 3. Поток T3, ожидая событие освобождения мьютекса, отдает управление потоку T1, потому что в данный момент поток T1 наиболее приоритетный поток в готовом состоянии.

Шаг 4. Если в системе появляется готовый поток T2 в то время, как T1 не успел выйти из критической секции и освободить мьютекс, то поток T2 блокирует более приоритетный поток T3. Таким образом, происходит инверсия приоритета.

В Windows 9x диспетчер обнаруживает зависимость более приоритетного потока от менее приоритетного потока, если эта зависимость возникает через объект ядра, и повышает приоритет менее приоритетного потока до уровня приоритета более приоритетного потока. Для предотвращения инверсии в современных версиях Widows планировщик учитывает время простоя готовых потоков и случайным образом повышает их динамический приоритет.

**Многопроцессорная обработка.** В операционной системе Windows имеется возможность управлять назначением потоков на конкретный процессор. Для управления таким назначением используются два атрибута.

Атрибут thread affinity определяет привязку потока к определенной группе процессоров. Этот атрибут представляет собой битовую маску, его указание вынуждает поток исполняться на указанном подмножестве процессоров. Для установки битовых масок привязки к процессорам используются функции SetThreadAffinityMask и SetProcessAffinityMask. Для считывания значения битовых масок привязки к процессорам используются функции GetProcessAffinityMask и GetThreadAffinityMask.

Настройка привязки потока к процессорам может использоваться для отладки в обычных SMP (симметричных мультипроцессорных системах) при наблюдении за активностью потоков средствами диспетчера задач. Основным применением является повышение производительности многопоточных приложений при исполнении в архитектурах с неоднородным доступом к памяти (NUMA).

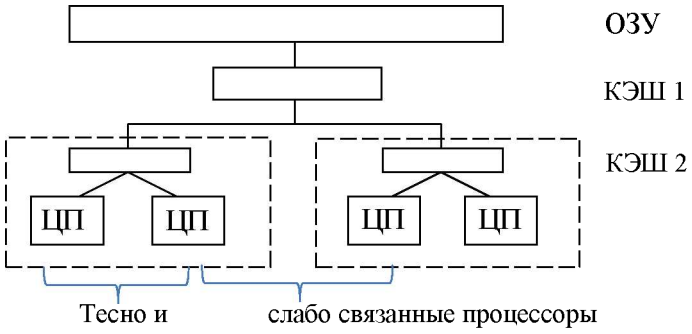



Рис. 6.3. Архитектура с неоднородным доступом к памяти (NUMA)

В таких архитектурах доступ по некоторым адресам памяти из заданного процессора может происходить быстро, а по некоторым медленнее. Точно также выделяются группы более и менее связанных между собой процессоров (рис.6.3). Зная топологию процессоров на компьютере и топологию задач приложения, удается оптимизировать приложение. Для получения данных о топологии имеются специальные функции. Если зависимые потоки поместить на один процессор, то они будут выполняться быстрее, так как будут обрабатываться через один внутренний кэш процессора, а не внешний, как в противном случае (рис.6.3). Однако жесткая привязка потоков к процессорам может снизить производительность. Для рекомендации планировщику назначать, по возможности, поток на процессор имеется еще один атрибут – идеальный процессор (ideal processor). Чтение и установка этого атрибута выполняется функциями GetThreadIdealProcessor и SetThreadIdealProcessor.

## Лекция 7. Синхронизация с использованием разделяемых переменных

Необходимость синхронизации на примере возникновения состояния состязания. Аппаратная реализация синхронизации. Задача о критической секции. Решение задачи для двух процессов в алгоритме Петерсона. Проблемы использования разделяемых переменных: агрессивная оптимизация, голодание, ложное разделение.

**Необходимость синхронизации.** Для того, чтобы параллельно или псевдопараллельно выполняющиеся процессы могли решать общую задачу, их исполнение необходимо синхронизировать. Существуют два типа задач синхронизации: конкурентные и кооперативные.

Синхронизация при совместном использовании разделяемого ресурса. Например, имеется очередь заданий на печать, в которую добавляют свои задания независимо работающие процессы. Такой вид взаимодействия (тип синхронизации) называется *конкурентным*. Его отличительной особенностью является то, что остановка одного из процессов-участников вне протокола взаимодействия не влияет на возможность других процессов продолжать работу.

Если речь идет об уведомлении одного процесса о завершении какой-либо операции другим процессом, выполняется *кооперативное* взаимодействие. Здесь, напротив, остановка любого участника со временем приведет к остановке всей системы процессов. Типичный пример такой синхронизации – буферизация данных, передаваемых по цепочке из процессов.

При отсутствии синхронизации процессов при совместном использовании разделяемого ресурса возникает искажение данных, называемое «состоянием состязания» (race condition). Пусть разделяемый ресурс – это глобальная переменная, используемая как счетчик числа обращений к некоторому ресурсу. При обращении к ресурсу процесс выполняет инкремент счетчика. Рассмотрим пример, иллюстрирующий искажение данных даже для такого простого ресурса.

Разделяемый ресурс – глобальная переменная типа long

```
long  g_x = 0;
```

Процессы представлены идентичными функциями потоков Thr1 и Thr2.

```
DWORD  WINAPI  Thr1(PVOID)    {
      /*обращение к ресурсу*/
      g_x++; /*увеличение счетчика обращений*/
      return 0;
}

DWORD  WINAPI  Thr2(PVOID)    {
      /*обращение к ресурсу*/
      g_x++; /*увеличение счетчика обращений*/
      return 0;
}
```

Для управления потоками объявляются массив указателей на функции потока thr_arr и массив описателей потоков thr_hnd.

```
LPTHREAD_START_ROUTINE thr_arr [2] = {Thr1, Thr2}
HANDLE  thr_hnd [2];
```

В функции main вначале запускаем наши потоки на исполнение вызовом CreateThread, проверяя и обрабатывая ошибки вызовом функции завершения процесса ExitProcess. Затем, при помощи функции WaitForMultipleObjects ожидаем завершения всех запущенных потоков.

```
int main() {
      DWORD id;
      for (inti=0; i<2; i++) {
        thr_hnd [i] = CreateThread(
          NULL,0,thr_arr[i],NULL,0,&id);
        if (thr_hnd [i] = = NULL) ExitProcess(-1);
      }
      WaitForMultipleObjects(
        2, thr_hnd, TRUE, INFINITE);
      return 0;
}
```

Очевидным кажется состояние переменной g_x=2 после завершения запущенных потоков. Однако истинное постусловие рассмотренной программы - g_x=1 $\vee$ g_x=2. Для того, чтобы понять почему значение переменной g_x может также оказаться равным 1, рассмотрим как представляется оператор g_x++ при компиляции программы. Столбцы Thr1 и Thr2 показывают ассемблерные инструкции потоков, а левый столбец – порядок исполнения этих инструкций на однопроцессорной машине. Показанный порядок выполнения инструкций возможен при переключении контекста в момент выполнения инкремента. С учетом того, что каждый из потоков имеет индивидуальную копию регистра EAX, действия потока Thr2 будут потеряны. Итоговое значение g_x оказывается равным 1.

```
      Thr1                      Thr2
/*      g_x++      */     /*      g_x++     */
(1) MOV EAX, [g_x]
(2)                       MOV EAX, [g_x]
(3)                       ADD EAX, 1
(4)                       MOV [g_x], EAX
(5) ADD EAX, 1
(6) MOV [g_x], EAX
```

Из-за «расщепления» команды инкремента возникает состояние состязания (race condition). Для предотвращения этого эффекта необходимо, чтобы эти три ассемблерные команды выполнялись как единое целое.

Для того, чтобы достичь неделимости инкремента и некоторых других арифметических операций в программном интерфейсе Windows имеется группа функций с префиксом interlocked. При использовании функции InterlockedExchangeAdd правильный код подсчета обращений к ресурсу выглядит следующим образом.

```
DWORD WinAPI Thr1(PVOID) {
    /*обращение к ресурсу*/
       InterlockedExchangeAdd(&g_x,1);/*увеличение
        счетчика обращений*/
         return 0;
}
```

**Аппаратная реализация синхронизации.** Для реализации неделимых операций в системе команд компьютеров имеется инструкция «проверить и установить блокировку» (test and set lock). Данная инструкция реализует базовую атомарную операцию, используя которую легко построить более сложные операции. Команда неделимым образом записывает ненулевое значение по адресу в памяти, одновременно сохраняя старое значение по этому адресу в регистре.

```
enter:
   TSL Reg, [Lock]
   CMP Reg, #0;0 значит, что текущий поток
   ; выполнил блокировку
   JNE enter; и ему можно войти в критическую секцию
      ;выполняем неделимую последовательность команд
leave:
   MOV [Lock], #0
```

Такой метод синхронизации называется спин-блокировка потому, что в цикле происходит постоянный опрос значения переменной по адресу Lock. При использовании interlocked-функции InterlockedExchange реализация спин-блокировки выглядит следующим образом.

```
BOOL g_ResInUse = FALSE;
//перед неделимой последовательностью
while(InterlockedExchange(&g_ResInUse,TRUE)==TRUE)
     Sleep(0);
// после неделимой последовательности команд
InterlockedExchange(&g_ResInUse,FALSE);
```

Функция InterlockedExchange присваивает значение, переданное во втором параметре, переменной, адрес которой указан в первом, и возвращает значение до модификации.

**Задача о критической секции.** Помимо аппаратной реализации возможна и программная реализация неделимой последовательности операций с использованием разделяемых переменных. Эдсгер Дейкстра в 1965 году сформулировал постановку задачи. Деккер описал первое корректное решение.

Задача получила название задача о критической секции. Она формулируется следующим образом. Каждый из процессов, участвующих во взаимодействии в цикле, последовательно выполняет четыре секции кода.

```
white (1)       {
           < протокол входа - enter( )>;
           < код в критической секции >;
           < протокол выхода - leave( )>;
           < код вне критической секции >;
}
```

Для решения задачи требуется выполнение четырех условий.

1. Процессы не должны находиться одновременно в критических секциях.

2. Не должно быть предположений о скорости выполнения процессов.

3. Процесс вне критической секции не должен блокировать другие процессы.

4. Если процесс начал выполнять протокол входа, то он рано или поздно должен войти в критическую секцию.

**Решение задачи для двух процессов в алгоритме Петерсона.** С момента постановки задачи было предложено несколько решений. В 1981 году Петерсон предложил наиболее компактный из извест-

ных вариантов решений задачи о критической секции для двух процессов. Он носит название алгоритм разрыва узла. Ниже показана последовательность синтеза этого алгоритма.

Для понимания сложности проблемы рассмотрим «наивный» алгоритм реализации протоколов входа и выхода из критической секции.

```
int lock = 0
void enter( )   {
   while (lock != 0) /*ждем*/;
   lock = 1;
}
void leave( ) { lock = 0; }
```

Очевидно, что такие протоколы не гарантируют выполнения условия взаимного исключения. Дело в том, что, если два процесса одновременно будут выполнять проверку значения флага lock, то они оба увидят разрешенное значение 0 и одновременно войдут в критическую секцию.

Поступим по-другому. Введем переменную turn, которая определяет очередь входа в критическую секцию. Если имеется два процесса, то пусть первый процесс входит в критическую секцию, если turn=0, а второй – если turn =1.

```
int turn = 0;
```

```
while(TRUE){
 while(turn!=0)/*ждем*/;
 < критическая секция >
 turn = 1;
 <вне критической
 секции>;
}
```

```
while(TRUE){
 while(turn!=1)/*ждем*/;
 < критическая секция >
 turn = 0;
 <вне критической
   секции>;
}
```

Этот способ обеспечивает выполнение условия взаимного исключения. Однако не выполняется третье условие: остановившись вне критической секции, первый процесс заблокирует второй и наоборот.

Правильное решение совмещает оба подхода и выглядит следующим образом.

```
int turn = 0;
int interested[2] = { FALSE, FALSE };

void enter( int process){
     int other = 1-process;
     interested[process] = TRUE;
     turn = process;
     while(turn==process&&
      interested[other]==TRUE)/*ждем*/;
}
void leave(int process){interested[process]=FALSE;}
```

Работа алгоритма основана на следующих заключениях. Модификацию условий нужно выполнять до проверки, а не после. Для каждого из процессов нужно завести по флагу (массив interested). Ненулевое значение флага свидетельствует о том, что соответствующий процесс готов войти в критический участок, поэтому каждый из процессов перед входом в критический участок должен выполнять проверку while (interested = = TRUE). Если два процесса одновременно начинают выполнять проверку, то возникает тупик. Для разрешения тупика (разрыва узла) вводится вспомогательная переменная turn, которая в случае конфликта разрешает вход в критическую секцию только одному процессу.

**Агрессивная оптимизация.** Реализация синхронизации с использованием разделяемых переменных и примитива test and set lock имеет максимальное быстродействие. Однако является достаточно сложной в силу ряда эффектов.

Эффект агрессивной оптимизации возникает, когда компилятор на основе предположения о неизменяемости переменной вне текущего потока сохраняет ее значение в регистре, тем самым нарушает логику работы программы. Например, в коде, показанном ниже, используется флаг g_fFinished для ожидания завершения операции в потоке с функцией CalcFunc.

```
volatile BOOL g_fFinished = FALSE;
/*volatile - загружаем значение из памяти при каждом
обращении к переменной (отключает оптимизацию кода
компилятором).*/

int main( )   {
```

```
    CreateThread (…, CalcFunc,…);
    while (g_fFinished = = FALSE);
}
DWORD WINAPI CalcFunc (PVOID)    {
    …
    g_fFinished = TRUE;
}
```

Если не использовать ключевое слово volatile, то возникнет риск того, что значения флага загрузятся в регистр процессора перед вычислением while, и в дальнейшем проверяться будет значение из регистра. То есть, ждущий процесс никогда не увидит окончания вычислений.

**Голодание.** При реализации спин-блокировки возможна ситуация, когда поток длительное время опрашивает условие входа в критическую секцию. Периодически это условие оказывается истинным. Тем не менее, поток не может войти в свою критическую секцию потому, что во время истинного значения условия входа поток не получает время процессора. Проблема решается введением задержек перед повторными попытками проверки условия входа. Например, пусть два потока выполняют идентичный код и в начальный момент времени остановились на строке (1).

```
(1)   while(1){
(2)         while(InterlockedExchange(&x,1));
(3)         /*критическая секция*/
(4)         InterlockedExchange(&x,0)
(5)   }
```

Когда первому потоку будет отведен квант времени, он в цикле выполняет строки (1)-(5). Так как строка (3) выполняется значительно дольше, чем остальные строки, первый поток по истечении кванта остановится на ней. Далее квант времени выделяется второму потоку. Однако он в течение отводимого ему кванта сможет только выполнять цикл (2). Ситуация повторяется при каждом следующем обслуживании: второй поток не сможет войти в свою критическую секцию.

**Ложное разделение.** Еще один эффект связан с понижением быстродействия программы при неправильном объявлении пере-

менных. Например, в объявлении, показанном ниже, две глобальные переменные объявлены вместе. Одна их них используется исключительно в функции потока Thread1, а другая в функции потока Thread2.

```
volatile int x = 0;  // используется в Thread1( )
volatile int y = 0;  // используется в Thread2( )
```

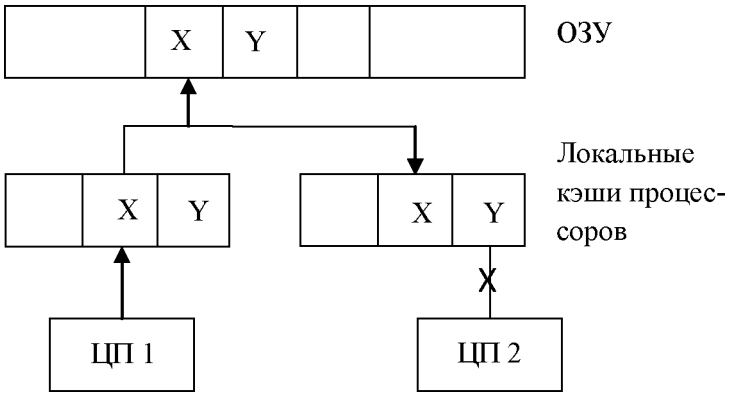



Рис. 7.1. Переменные X и Y находятся в одной линии кэша, к ним невозможен одновременный доступ из разных процессоров

Потоки кажутся независимыми, однако, это не так. Если поток Thread1 выполняет операцию с переменной x, поток Thread2 вынужден приостанавливать операцию с переменной y и наоборот, из-за чего снижается быстродействие программы. Это происходит потому, что для ускорения работы с памятью каждый из процессоров использует локальный кэш, однако загрузка значений в кэш производится не побайтно, а блоком. Обычно это участок памяти, выровненный по 32-байтной границе. Он называется кэш-линия. Если две переменные попадают в одну кэш-линию, то эффект от использования кэш-памяти пропадает. Поэтому следует выравнивать пере-

менные по границам кэш-линий или вводить фиктивные переменные, чтобы гарантировать, что переменные не попадут в одну кэш-линию.

Правильные объявления будут выглядеть следующим образом.

```
#define CACHE_LINE  32
#define CACHE_ALIGN __declspec(align(CACHE_LINE))
// используется в Thread1( )
volatile CACHE_ALIGN int x = 0;
// используется в Thread2( )
volatile CACHE_ALIGN int y = 0;
```

Для того, чтобы избежать проблем, сопутствующих синхронизации с разделяемыми переменными при сохранении высокой скорости, в Windows реализована группа функций для организации критической секции. Функция InitializeCriticalSection используется для настройки структуры данных, служащей для управления критической секцией перед первым входом. Функция DeleteCriticalSection служит для перевода структуры управления критической секцией в исходное состояние. Функции EnterCriticalSection вместе с TryEnterCriticalSection реализуют протокол входа в критическую секцию. Функция LeaveCriticalSection реализует протокол выхода из критической секции. Дополнительно имеются функции InitializeCriticalSectionAndSpinCount и SetCriticalSectionSpinCount для настройки числа попыток входа в критическую секцию перед переходом в режим ядра (или возврата в случае TryEnterCriticalSection).

Достоинством функций является высокое быстродействие. Однако в них нельзя указать таймаут (аварийный выход через определенное время), их нельзя применять для организации взаимодействия между процессами, только для взаимодействия между потоками одного процесса. От этих недостатков свободны процедурные методы синхронизации.

## Лекция 8. Процедурные методы синхронизации в операционной системе Windows

Объекты синхронизации, функции ожидания. Управление объектами ядра. События. Семафоры. Мьютексы. Пример задачи об ограниченном буфере.

**Объекты синхронизации, функции ожидания.** Процедурные методы синхронизации в Windows используются по отношению к объектам, реализованным в ядре операционной системы. Для такой синхронизации применяются только объекты ядра, которые могут находиться в сигнальном (свободном) и несигнальном (занятом) состоянии. К ним относятся рассмотренные ранее объекты: задания, процессы, потоки. Эти объекты переходят в сигнальное состояние при завершении исполнения. Имеется группа объектов, используемых специально для синхронизации потоков и процессов, - это события, мьютексы, семафоры и таймеры. Отдельную группу образуют объекты, предназначенные для синхронизации ввода-вывода.

С точки зрения программиста все перечисленные объекты имеют общее свойство: их описатели можно использовать в функциях ожидания WaitForSingleObject, WaitForMultipleObject и некоторых других. Если объект находится в несигнальном состоянии, то вызов функции ожидания с описателем объекта блокируется. Дескриптор потока, который вызвал функцию, помещается в очередь этого объекта ядра, а сам поток переходит в состояние ожидания. Когда объект ядра переходит в сигнальное состояние, выполняются действия, специфичные для данного типа объекта ядра. Например, может быть разблокирован один или все потоки, ожидающие на данном объекте ядра.

В функцию WaitForSingleObject передаются два параметра: описатель объекта и значение таймаута. Таймаут определяет предельное время нахождения потока в блокированном состоянии. В функцию WaitForMultipleObjects передается массив описателей объектов ядра. Для этого указывается адрес начала массива и количество описателей в нем. Также указывается параметр, определяющий семантику ожидания на группе описателей: можно ждать перехода всех объектов ядра в сигнальное состояние или какого-то одного объекта. Указывается также таймаут.

При возврате из функции ожидания обычно требуется определить причину завершения функции. В случае ошибки функция возвращает значение WAIT_FAILED. Для уточнения состояния ошибки далее используют GetLastError и другие функции. В случае завершения по таймауту возвращается значение WAIT_TIMEOUT. Если причиной возврата является переход в сигнальное состояние, возвращается значение WAIT_OBJECT_0.

Если выполняется ожидание на функции WaitForMultipleObjects, то, чтобы узнать, какой именно объект перешел в сигнальное состояние, нужно проверить, что значение, возвращенное функцией, не равно WAIT_FAILED и WAIT_TIMEOUT и вычесть из него константу WAIT_OBJECT_0. В результате мы получим индекс описателя объекта, перешедшего в сигнальное состояние, в массиве описателей, переданном в функцию WaitForMultipleObject.

**Управление объектами ядра.** Рассмотрим операции, относящиеся как к объектам, использующимся в функциях ожидания, так и к другим объектам ядра в операционной системе Windows.

Для *создания* объектов ядра используются индивидуальные для каждого объекта функции, имеющие префикс Create. Например, мьютекс может быть создан вызовом

HANDLE mutex=CreateMutex(NULL, FALSE, NULL).

Обычно первым параметром всех функций, создающих объекты ядра, является структура атрибутов безопасности, а последним – имя объекта.

*Закрытие* описателя любого объекта ядра выполняется вызовом функции CloseHandle.

Чтобы понять принцип работы функции, рассмотрим более детально, что представляет собой описатель объекта ядра (рис.8.1).

Для каждого процесса в ядре операционной системы создается таблица описателей. Запись в таблице содержит некоторые атрибуты описателя и указатель на объект ядра. На один и тот же объект ядра могут указывать несколько описателей, возможно из таблиц описателей разных процессов. В любом объекте ядра имеется специальный атрибут для подсчета ссылок на объект. Значение описателя, используемое в адресном пространстве процесса, – это смещение на запись в таблице описателей процесса.

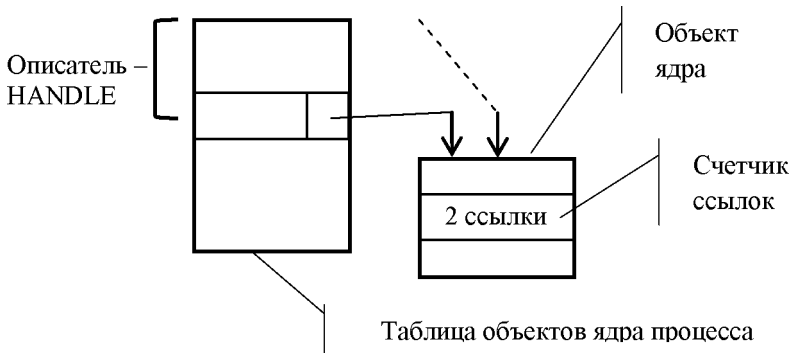



Рис. 8.1.Описатель объекта ядра в Windows

При закрытии описателя происходит модификация атрибута, указывающего на то, что запись теперь недействительна. Далее происходит декремент счетчика ссылок у объекта. Если значение счетчика ссылок достигает нуля, происходит удаление объекта из памяти ядра операционной системы.

Важное преимущество объектов ядра – возможность их использования для синхронизации потоков, принадлежащих разным процессам. Для этой цели нужно уметь передавать описатели объектов между процессами. Их можно передавать путем наследования, именования и дублирования.

При *наследовании* вначале необходимо модифицировать атрибут наследования в таблице описателей процесса. Это можно сделать непосредственно при создании описателя, как показано ниже.

```
SECURITY_ATTRIBUTE sa;
sa.nlength = sizeof(sa);
sa.lpSecurityDescriptor=NULL; /*NULL - защита по умол-
чанию; определяет, кто может пользоваться объектом
(при NULL - создатель процесса и администраторы) */
sa.bInheritHandle=TRUE; /*наследовать описатель*/
HANDLE hMutex = CreateMutex (&sa, FALSE, NULL);
```

Можно воспользоваться функциями GetHandleInformation и SetHandleInformation для изменения атрибутов описателя уже после создания объекта ядра.

Наследование описателей происходит следующим образом (рис.8.2). При создании процесса функцией CreateProcess для него создается новая таблица описателей. В эту таблицу копируются все записи родительского процесса, имеющие атрибут наследования. Записи помещаются по тем же смещениям, по которым размещались исходные записи родительского процесса. Для каждой скопированной записи производится инкремент счетчика ссылок у связанного с ней объекта ядра.

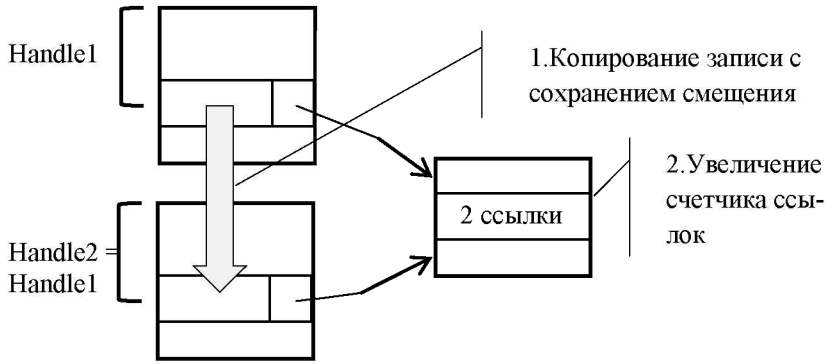



Рис. 8.2. Схема наследования описателя в Windows

Дочерний процесс может использовать ссылки на унаследованные объекты ядра. Для передачи значений унаследованных описателей в дочерний процесс обычно используется командная строка или переменные окружения. Если дочерний процесс уже создан, то нельзя воспользоваться механизмом наследования.

При создании объекта ядра может быть указано его имя:

HANDLE mutex = CreateMutex(NULL, FALSE, “SomeMutex”);.

В данном случае любой процесс может получить *доступ к объекту ядра по имени*, если такой доступ разрешен настройками безопасности. При повторном вызове функции Create проводится проверка: имеется ли объект ядра с данным именем и типом. Если объект с таким именем и типом уже существует, то функция не создает новый объект, а возвращает описатель существующего объекта (при этом остальные параметры в вызове игнорируются). При необходимости можно проверить, создан ли объект ядра или получен описатель ранее созданного объекта:

```
if(GetLastError()==ERROR_ALREADY_EXISTS){
   /*если создан ранее*/
}.
```

Механизм ссылки на объекты ядра подходит для всех случаев взаимодействия, когда можно определить соглашения на имена создаваемых объектов. Этот механизм также часто используется для контроля количества запущенных экземпляров приложения.

Универсальным способом передачи описателей объектов ядра между процессами является *дублирование описателей*. Дублирование выполняется функцией DuplicateHandle:

```
BOOL DuplicateHandle(
HANDLE   hSourceProcessHandle,/*описатель   исходного
процесса*/
HANDLE   hSourceHandle,    /*дублируемый    описатель
процесса-источника*/
HANDLE hTargetProcessHandle, /*процесс-приемник*/
PHANDLE  phTargetHandle,   /*описатель  в  процессе-
приемнике*/
DWORD    dwDesiredAccess,    /*настройка    атрибутов
дублируемого описателя*/
DWORD dwQptions.
```

В дублировании принимают участие (в общем случае) три процесса: процесс, выполняющий дублирование, процесс – источник и процесс – приемник. Если источником или приемником является управляющий процесс, то вместо соответствующего описателя помещается вызов функции GetCurrentProcess. Особенностью дублирования является то, что необходимо использовать механизм межпроцессного взаимодействия для передачи продублированного описателя в процесс – приемник. Функция DuplicateHandle его сформирует, но нужно также поставить в известность сам процесс – приемник о том, что в его таблицу описателей добавлен новый описатель.

Теперь рассмотрим объекты синхронизации и специфические для каждого объекта функции процедурной синхронизации.

**События** используются при реализации кооперативной синхронизации, когда один поток ждет поступления данных от другого. При наступлении события объект переходит в сигнальное состояние.

Если событие не наступило – находится в несигнальном состоянии. В зависимости от того, каким образом осуществляется перевод события в несигнальное состояние, существуют два типа событий: событие со сбросом вручную и событие с автоматическим сбросом. Оба типа объектов создаются функцией CreateEvent:

```
HANDLE CreateEvent(
PSECURITY_ATTRIBUTES sa,/*атрибуты безопасности*/
BOOL fManualReset,/*тип объекта TRUE - со сбросом
вручную*/
BOOL fInitialState,/*начальное состояние события*/
LPCTSTR name);/*имя события*/.
```

Для управления событием используются следующие функции: SetEvent – устанавливает событие в сигнальное состояние; ResetEvent – устанавливает событие в несигнальном состояние; PulseEvent – устанавливает в сигнальное состояние и сбрасывает в несигнальное.

Функция PulseEvent выполняет разблокирование потоков, если таковые имеются в момент ее вызова. События с автоматическим сбросом целесообразно использовать, если происходит периодическое ожидание завершения события. В данном случае вместо комбинации вызовов WaitForSingleObject/ResetEvent достаточно использовать один вызов WaitForSingleObject.

**Семафоры** – универсальные объекты процедурной синхронизации, предложены Э. Дейкстра. Семафоры выполняют роль счетчика числа доступных ресурсов. С семафорами определены две операции. Для возврата ресурсов в пул доступных ресурсов используется операция увеличения счетчика (up-операция или V-операция). Для захвата ресурсов используется операция уменьшения счетчика (down-операция или P-операция). Если счетчик ресурсов принимает значение ноль, то поток блокируется, до тех пор, пока ресурс не будет возвращен в пул другим потоком с использованием up или V-операции. В любой момент счетчик не может быть больше максимального числа ресурсов и меньше нуля.

Семафор создается при помощи функции CreateSemaphore:

```
HANDLE CreateSemaphore(
PSECURITY_ATTRIBUTES sa, /*атрибуты безопасности*/
LONG   lInitialCount,/*начальное   значение   счетчика
(ресурсов)*/
LONG    lMaxCount,/*предельное    значение    счетчика
(ресурсов)*/
LPCTSTR name);/*имя семафора*/.
```

Инкремент счетчика семафора выполняется при помощи вызова функции BOOL ReleaseSemaphore (HANDLE, LONG ReleaseCount). В ней, в отличие от абстрактных операций up или V, можно указать, насколько следует увеличить счетчик (параметр ReleaseCount). Декремент счетчика выполняется при помощи любой функции ожидания, примененной к семафору, например, WaitForSingleObject.

**Мьютекс** (MUTEX –MUTualEXclusion, взаимное исключение) – бинарный семафор, специально предназначенный для решения задачи взаимного исключения, защиты критической секции. Для создания мьютекса используется функция CreateMutex:

```
HANDLE CreateMutex(
PSECURITY_ATTRIBUTES sa,/*атрибуты безопасности*/
BOOL   fInitialOwner,/*признак,   является   ли   поток,
создающий мьютекс его владельцем*/
LPCTSTR name);/*имя мьютекса*/.
```

Операция освобождения мьютекса (up или V операция) выполняется вызовом функции BOOL ReleaseMutex (HANDLE). Операция захвата мьютекса выполняется при помощи любой функции ожидания, например, WaitForSingleObject.

Мьютекс можно рассматривать как бинарный семафор. Также мьютекс похож на критическую секцию. От семафора мьютекс отличается тем, что он имеет атрибут владения (как критическая секция). Отличие от критических секций состоит в том, что при помощи мьютекса можно синхронизировать потоки в разных процессах и указывать таймаут.

Если поток владеет мьютексом, то вызов функции ожидания с этим мьютексом завершится успешно, при этом увеличится внутренний счетчик рекурсий. Функция ReleaseMutex выполняет декре-

мент счетчика рекурсий и освобождает мьютекс, если счетчик достигает значения ноль. Также ведет себя и критическая секция.

Если попытаться освободить мьютекс из протока, который им не владеет, то возникнет ошибка и функция GetLastError вернет значение ERROR_NOT_OWNED.

Если поток, владеющий мьютексом, завершается, то мьютекс переходит в несигнальное состояние и может быть использован другими потоками. Вызов функции ожидания с таким мьютексом будет выполнен, но функция GetLastError вернет ошибку WAIT_ABANDONED.

**Пример задачи об ограниченном буфере.** Проиллюстрируем применение семафоров и мьютексов для решения одной из классических задач синхронизации – задаче об ограниченном буфере, также известной как проблема производителя и потребителя.

Два процесса совместно используют буфер ограниченного размера. Один из них (производитель) помещает данные в этот буфер, а другой (потребитель) – считывает их оттуда. Требуется обеспечить ожидание процесса-производителя, когда буфер заполнен и ожидание процесса-потребителя, когда буфер пуст. В других случаях процессы могут добавлять (поставщик) и извлекать (потребитель) данные из буфера.

Приведем программу на псевдокоде с условными операциями для семафоров, мьютексов и буфера, иллюстрирующую решение проблемы для одного поставщика и одного потребителя.

```
semaphore mutex = 1;// защита критического
                    // ресурса - буфера
semaphore empty = 100; //число пустых сегментов буфера
semaphore full = 0; // число полных сегментов буфера

// код процесса - поставщика
void producer(){
      int item;
      while(1){
            item=produce_item();// создать данные,
            //помещаемые в буфер
            down(&empty); // уменьшить счетчик пустых
            // сегментов буфера
            down(&mutex);// вход в критическую секцию
```

```
            insert_item(item);// поместить в буфер
            // новый элемент
            up(&mutex); //выход из критической секции
            up(&full); //увеличить счетчик
            // полных сегментов буфера
      }
}

// код процесса – поставщика
void consumer(void){
      int item;
      while(1){
            down(&full); // уменьшить счетчик
            // полных сегментов буфера
            down(&mutex);// вход в критическую секцию
            item = remove_item(;// извлечь элемент
            // из буфера
            up(&mutex); //выход из критической секции
            up(&empty); //увеличить счетчик
            // пустых сегментов буфера
            consume_item(item);// обработать элемент
      }
}
```

Заметим, что семафоры в приведенном примере используются по-разному. Семафор mutex служит для реализации взаимного исключения, то есть конкурентной синхронизации. В реальной программе для операционной системы Windows его необходимо реализовывать при помощи мьютекса или критической секции. Семафоры full и empty используются для задания определенной последовательности событий при кооперативной синхронизации. В реализации для Windows уместно использовать семафоры. Таким образом, проблема об ограниченном буфере – пример гибридной проблемы, где встречается как кооперативная, так конкурентная синхронизация.

## Лекция 9. Тупики и защита от них

Определение тупика. Условие возникновения тупика. Моделирование блокировок. Стратегии борьбы с блокировками: игнорирование проблемы, обнаружение и восстановление, динамическое избегание тупиковых ситуаций, предотвращение тупиков при помощи устранения одного из условий их возникновения.

**Определение тупика.** Процедурные методы синхронизации более удобны в применении, чем методы синхронизации на основе разделяемых переменных. Однако, при некорректном их использовании возможно возникновение еще одного типа ошибок синхронизации, известного как проблема тупиков (deadlock).

Группа процессов находится в тупиковой ситуации, если каждый процесс из группы ожидает событие, которое может вызвать только другой процесс из этой же группы.

**Условия возникновения тупика.** В большинстве случаев событием, которого ждет каждый процесс, является освобождение ресурса. В 1971 году Коффман с соавторами доказали, что для возникновения ситуации блокировки в системе процессов и ресурсов должны выполняться четыре условия.

1. Условие взаимного исключения. Каждый ресурс в любой момент отдан ровно одному процессу или доступен.
2. Условие удержания и ожидания. Процессы, удерживающие полученные ранее ресурсы, могут запрашивать новые.
3. Условие отсутствия принудительной выгрузки. У процесса нельзя забрать ранее полученный ресурс. Процесс, владеющий ресурсом, должен сам освободить его.
4. Условие циклического ожидания. Должна существовать круговая последовательность из двух и более процессов, каждый из которых ждет доступа к ресурсу, удерживаемому следующим членом последовательности.

Для возникновения блокировки должны выполняться все четыре условия. Если хотя бы одно условие отсутствует, блокировка невозможна.

**Моделирование блокировок.** В 1972 году Холт предложил метод моделирования условий возникновения тупика, используя ориентированные графы. Графические обозначения диаграмм Холта (рис. 9.1). Графы имеют два типа узлов. Кругом обозначают процесс, квадратом – ресурс. Ребро, направленное от ресурса (квадрат) к процессу (круг) обозначает, что ресурс был ранее запрошен процессом, получен и в данный момент используется. Ребро, направленное от процесса к ресурсу, означает, что процесс в данный момент блокирован и находится в состоянии ожидания доступа к данному ресурсу.

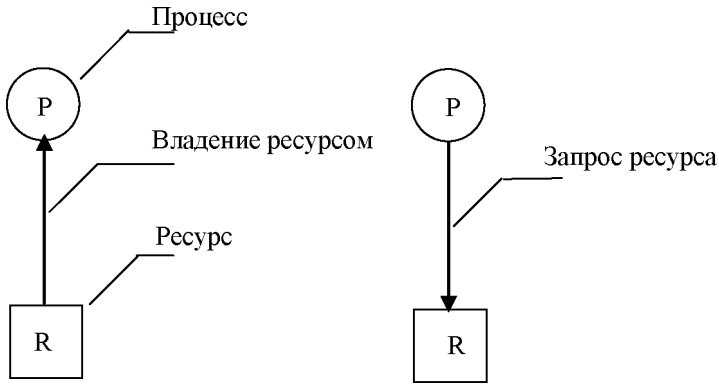



Рис. 9.1. Условные обозначения на диаграммах Холта

Простейший тупик в системе из двух процессов и двух ресурсов представлен графом Холта (рис.9.2).

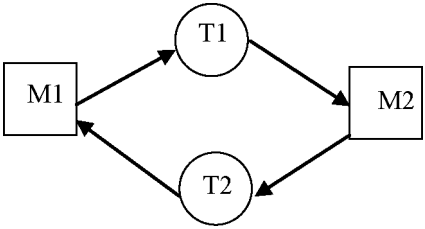



Рис. 9.2. Простейший тупик

Рассмотрим, каким образом может возникнуть показанная на рис.9.2 ситуация. Пусть имеются функции потоков T1 и T2, код которых показан ниже. В качестве ресурсов выступают мьютексы M1 и M2. В начальный момент нет запросов и захваченных ресурсов.

```
DWORD T1(PVOID){          DWORD T2(PVOID){
(1)WaitforSingleObject(M1,-1);
(2)                       WaitforSingleObject(M2,-1);
(3)                       WaitforSingleObject(M1,-1);
(4)WaitforSingleObject(M2,-1);
(5)--------------тупик---------------------------
//критическая секция      //критическая секция
ReleaseMutex(M1);         ReleaseMutex(M1);
ReleaseMutex(M2);         ReleaseMutex(M2);
}                         }
```

В момент времени (1) поток T1 выполняет захват мьютекса M1. Так как мьютекс свободен, происходит успешный захват. Это показано ребром M1→T1. Далее в момент (2) планировщик может передать управление потоку T2, который выполняет захват свободного мьютекса M2. На графе (рис.9.2) появляется ребро M2→T2. Продолжая вычисления, поток T2 выполняет запрос мьютекса M1, но так как мьютекс M1 уже заблокирован, поток переходит в состояние «ожидание» на мьютексе M1. На графе это показано ребром T2→M1. В момент (4) управление передается потоку T1, который выполняет попытку захватить мьютекс M2, уже принадлежащий потоку T2, и тоже переходит в состояние ожидания. Цикл на графе Холта, обозначающий состояние тупика, замыкается ребром T1→M2.

Рассматривая данный простейший сценарий возникновения тупика, можно сделать следующие выводы. Во-первых, тупики возникают не при каждом исполнении. Например, если бы на шаге (2) планировщик не выполнил переключение контекста на поток T2, а дал возможность потоку T1 выполнить захват мьютекса M2, тупик бы не возник. Во-вторых, аккуратным планированием можно избежать блокировок. То есть, планировщик, обладая некоторой информацией о необходимых потокам ресурсах, мог бы исключить переключение контекста, вызвавшее блокировку. В-третьих, блокировки можно обнаружить. При переходе процесса в состояние ожидания при запросе ресурса, можно выполнить анализ графа ресурсов и процессов на наличие цикла, то есть тупика.

**Стратегии борьбы с блокировками.** Рассмотрим стратегии предотвращения блокировок, вытекающие их моделей Холта и Коффмана.

*«Страусовый алгоритм».* Допущение, что блокировки в системе можно игнорировать. Усилия, затраченные на преодоление блокировок, могут себя не оправдать. Например, область памяти, в которой исполняется ядро операционной системы, ограничена. Также имеют предел таблицы процессов, таблицы открытых файлов и таблицы других системных ресурсов. Допустим, в операционной системе можно открыть n файлов, и каждый из N запущенных процессов открыл по n/N файлов. Далее, если каждый из процессов будет периодически повторять попытки открыть еще по одному файлу,

возникнет тупик. Большая часть операционных систем, включая UNIX и Windows, игнорируют эту проблему. Полное ее решение включало бы неприемлемые с инженерной точки зрения ограничения на использование ресурсов процессами.

*Обнаружение и восстановление.* Данная стратегия состоит в том, чтобы дать возможность произойти блокировке, обнаружить ее и предпринять действия для ее исправления. Существует несколько способов восстановления из тупика.

Принудительная выгрузка заключается в том, чтобы взять ресурс у одного процесса, отдать другому процессу и затем возвратить назад. Такая возможность зависит от свойств ресурса. Например, в случае необходимости, можно временно прервать процесс печати и предоставить принтер другому процессу.

При восстановлении через откат процесс периодически сохраняет свое состояние, создавая контрольные точки. Это означает, что его состояние записывается в файл, и впоследствии процесс может быть возобновлен из этого файла. Чтобы выйти из тупика процесс, занимающий необходимый ресурс, откатывает к тому моменту времени, перед которым он получил данный ресурс. Освободившийся ресурс отдается другому процессу, попавшему в тупик.

Простейшим способом выхода из блокировки является уничтожение одного или нескольких процессов, находящихся в цикле взаимоблокировки. В некоторых случаях имеются процессы без побочных эффектов, которые можно перезапустить, не нарушая работу системы. Например, процедуру компиляции всегда можно повторить заново.

Рассмотрим простейший случай обнаружения тупика, когда каждый тип ресурсов системы представлен единственным ресурсом. Например, такая система могла бы иметь один сканер, одно устройство записи дисков, один принтер и так далее.

Обнаружение тупика в случае единственности ресурса каждого типа состоит из двух этапов: в текущем состоянии необходимо построить граф Холта, далее найти на нем цикл по какому-либо алгоритму.

Рассмотрим пример системы из семи процессов, состояние которых описывает таблица.

| Процесс | Захват ресурса | Ждет ресурс |
|---|---|---|
| A | R | S |
| B | - | T |
| C | - | S |
| D | V | S, T |
| E | T | V |
| F | W | S |
| G | V | U |

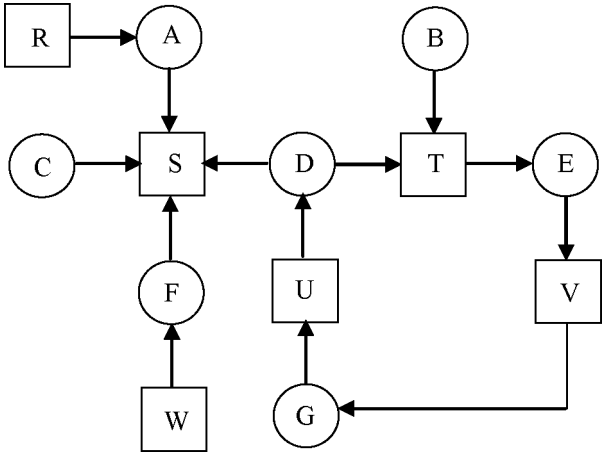



Рис. 9.3. Граф Холта, показывающий состояние тупика процессов D,V и G

Граф Холта, построенный по таблице, показан на рис.9.3. По визуальной модели состояния системы процессов видно, что процессы D, E и G находятся в тупике. Процессы A, C и F могут завершить исполнение и высвободить занимаемые ресурсы, поскольку любому из них можно предоставить ресурс S.

Для анализа графа большей размерности может применяться следующий алгоритм. Используем список L для хранения посещенных узлов и пометки для пройденных ребер. Алгоритм применяется для каждого узла графа.

Шаг 1. Начальные условия: имеется текущий узел, список L пустой, ребра не маркированы.

Шаг 2. Текущий узел добавляем в конец списка L. Если текущий узел присутствует в списке два раза, то граф содержит цикл (записанный в L). Алгоритм завершается.

Шаг 3. Если из текущего узла выходит немаркированное ребро, переходим к шагу 4, иначе переходим к шагу 5.

Шаг 4. Выбираем любое немаркированное ребро, маркируем его, по нему переходим к новому текущему узлу. Переходим к шагу 2.

Шаг 5. Удаляем последний узел из списка. Предпоследний узел объявляем текущим. Возвращаемся к шагу 3. Если перед шагом 5 в списке начальный узел, то граф не содержит циклов. Алгоритм завершается.

Теперь рассмотрим общий случай обнаружения тупика, если имеется один или несколько типов ресурсов представленных не одним, а сразу несколькими ресурсами.

В системе имеется m типов ресурсов. $E = (E_1, E_2, \ldots, E_m)$ – вектор существующих ресурсов, в котором $E_j$- количество ресурсов типа j $(1 \le j \le m)$. $A = (A_1, A_2, \ldots, A_m)$ – вектор доступных ресурсов. В системе имеется n процессов. Матрица текущего распределения ресурсов C имеет вид:

$$\begin{pmatrix} C_{11}C_{12}\ldots\ldots\ldots\ldots C_{1m} \\ C_{21}\ldots\ldots\ldots\ldots\ldots C_{2m} \\ \ldots\ldots\ldots\ldots\ldots\ldots\ldots \\ C_{n1}\ldots\ldots\ldots\ldots\ldots C_{nm} \end{pmatrix}$$

, где $C_{ij}$ – количество ресурсов типа j, полученных процессом i.

Матрица запросов ресурсов R имеет вид:

$$\begin{pmatrix} R_{11}R_{12}\ldots\ldots\ldots\ldots R_{1m} \\ R_{21}\ldots\ldots\ldots\ldots\ldots R_{2m} \\ \ldots\ldots\ldots\ldots\ldots\ldots\ldots \\ R_{n1}\ldots\ldots\ldots\ldots\ldots R_{nm} \end{pmatrix}$$

, где $R_{ij}$- количество ресурсов типа j, запрашиваемых процессом i.

С учетом условия взаимного исключения для ресурсов выполняется соотношение $\sum_i c_{ij} + a_i = e_j$ .

Алгоритм обнаружения тупиков состоит из следующих шагов.

Все процессы не маркированы.

Шаг 1. В матрице R ищем строку, соответствующую немаркированному процессу, которая меньше либо равна A, ($r_j \le a_j$).

Шаг 2. Если такая строка найдена, то маркируем соответствующий процесс, прибавляем строку к вектору A, возвращаемся к шагу 1.

Шаг 3. Если немаркированных процессов нет, то алгоритм завершается.

Если, после завершения алгоритма остаются немаркированные процессы, то они находятся в тупике.

*Пример*. Требуется определить, находится ли система процесс-ресурс в тупике.

E=(4, 2, 3, 1) A=(2, 1, 0, 0)

$$C = \begin{pmatrix} 0010 \\ 2001 \\ 0120 \end{pmatrix} R = \begin{pmatrix} 2001 \\ 1010 \\ 2100 \end{pmatrix}$$

Убедимся в выполнении условия взаимного исключения для ресурсов: сложим строки матрицы C и вектор A, получим вектор E.

$A^1 = (2,2,2,0)$ *маркируем* $3-$*й процесс*

$A^2 = (4,2,2,1)$ *маркируем* $2-$*й процесс*

$A^3 = (4,2,3,1)$ *маркируем* $1-$*й процесс*

Выполняя последовательность шагов алгоритма, маркируем все процессы. Следовательно, нет процессов, находящихся в тупике.

*Избегание взаимных блокировок.* В первом примере показано, что, выбирая определенный порядок планирования среди нескольких возможных, планировщик избегает тупика. Стратегию поведения планировщика в случае двух процессов и произвольного количества ресурсов (в каждом типе имеется 1 ресурс) представляет диаграмма траектории ресурсов (рис. 9.4).

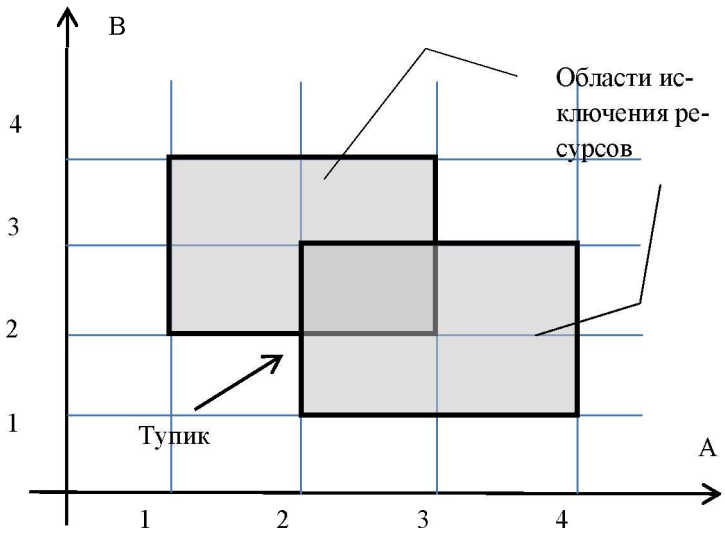



Рис. 9.4. Диаграмма траектории ресурсов

Диаграмма представляет собой первый квадрант координатной плоскости. Оси являются временными, представляя события процесса A и процесса B. Для определенности свяжем отсчеты времени процесса A со следующими событиями: 1 – захват принтера, 2 – захват плоттера, 3 – освобождение принтера, 4 – освобождение плоттера. На временной оси процесса B отмечаются следующие события: 1 – захват плоттера, 2 – захват принтера, 3 – освобождение плоттера, 4 – освобождение принтера.

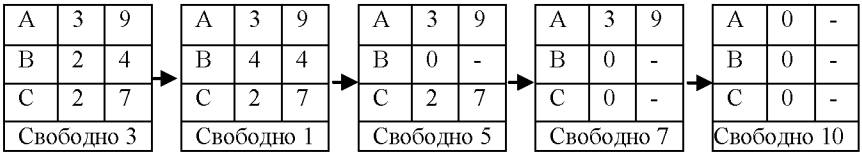
С учетом введенных обозначений, точка на диаграмме траектории ресурсов представляет состояние системы. Она может двигаться только вверх и вправо. В некоторых областях диаграммы система не может находиться по правилу исключения для ресурса. Области называются, соответственно, области исключения ресурса (рис. 9.4).

Если во время планирования точка, обозначающая состояние системы, попадет в область X, то со временем в системе возникнет блокировка. Так как точка не может двигаться вниз и влево. Любое состояние в области серого цвета является безопасным состоянием. В таком состоянии планировщик путем аккуратного планирования ресурсов может избежать возникновения тупика (рис. 9.4).

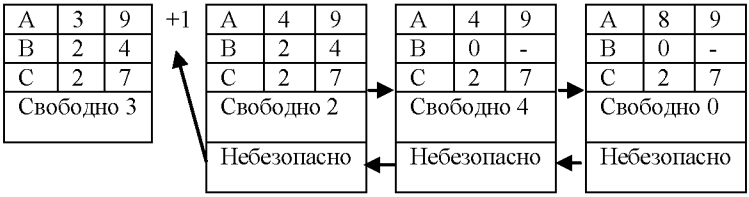
*Алгоритм Банкира для одного типа ресурсов.* Анализ диаграммы траектории ресурсов показал, что цель алгоритма планирования - проверить приводит ли удовлетворение запроса на ресурс к выходу из безопасного состояния системы.

Пусть имеется один ресурс некоторого типа. Воспользуемся аналогией работы планировщика с поведением банкира, кредитующего клиентов. Запишем состояние системы в виде таблицы распределения ресурсов, в которой A, B, C – имена клиентов банка (процессы); второй столбец – выданные клиенту кредиты (ресурсы); третий столбец – число кредитов, необходимых клиентам. Внизу подписано число свободных кредитов у банкира.

Вначале проверим, является ли текущее состояние безопасным. То есть, сможем ли мы предложить стратегию обслуживания клиентов, удовлетворяющую все запросы на кредиты. Для этого будем давать ссуду только тем клиентам, запросы которых в кредитах мы можем покрыть полностью при текущем количестве свободных кредитов у банка. Ниже показана такая стратегия обслуживания.




Теперь допустим, что в рассмотренном безопасном состоянии клиент A выставил запрос на один кредит. Рассмотрим гипотетическое состояние, возникающее при удовлетворении такого запроса.




Мы видим, что возникает состояние, в котором не хватает одного кредита для удовлетворения запроса клиента A (показано выше), так и клиента C (определяется аналогично). Оказавшись в таком состоянии, банкир признает банкротство, так как вынужден отказать

всем не обслуженным клиентам. То есть система оказывается в тупике, следовательно, запрос процессом А одного ресурса приводит к выходу из безопасного состояния и должен быть отклонен.

*Предотвращение при помощи устранения одного из условий возникновения тупика.* Если удастся предложить стратегию работы с ресурсами, в которой исключено хотя бы одно из условий Хоффмана, тупик не возникнет.

1. Атака взаимного исключения. Пример использования очереди печати показывает, что можно избежать захвата принтера.

2. Атака условия удержания и ожидания. Если все ресурсы запрашиваются одновременно, то тупиков не возникнет. Например, можно использовать функцию WaitForMultipleObjects.

3. Атака условия отсутствия принудительной выгрузки. Реализуется при обработке транзакций. Каждый процесс готов освободить захваченные ресурсы по запросу менеджера транзакций.

4. Атака условия циклического ожидания. Если все ресурсы пронумерованы, и каждый процесс имеет право захватывать ресурс только с большим порядковым номером, чем номера уже захваченных ресурсов, то тупика не возникнет.

## Экзаменационные вопросы по разделу III

1. Абстракция процесса, управление процессами в многозадачной операционной системе. Определение процесса. Диаграмма состояния, контекст, дескриптор процесса. Квантование и приоритетное планирование. Нити (потоки исполнения).
2. Функциональные возможности многозадачности в ОС Windows. Способы использования многозадачности в приложениях.
3. Планировщик ОС Windows. Класс и уровень приоритета. Переключение контекста. Потоки, не являющиеся готовыми. Динамический приоритет.
4. Эффект инверсии приоритетов. Пример возникновения инверсии. Способы преодоления.
5. Мультипроцессорная обработка в ОС Windows. Термины, вызовы API, их назначение.
6. Состояние состязания. Пример возникновения и способ преодоления.

7. Средства синхронизации в режиме пользователя в ОС Windows. Функции, реализующие атомарные операции, объект «критическая секция».
8. Задача о критической секции. Алгоритм Питерсона для двух процессов. Условия задачи. Объяснение принципа работы алгоритма.
9. Предотвращение агрессивной оптимизации кода с использованием модификатора volatile. Эффект голодания, пример возникновения.
10. Эффект ложного разделения переменных. Пример влияния кэш-линий на скорость исполнения многопоточных программ.
11. Управление объектами ядра в ОС Windows. Описатель объекта. Таблица описателей объектов процесса. Создание, наследование, именование, дублирование описателей.
12. Средства синхронизации в режиме ядра в ОС Windows. События, семафоры, мьютексы.
13. Эффект взаимоблокировки или возникновения тупика. Определение, условия возникновения, моделирование графами Холта.
14. Стратегия «обнаружение-устранение» для борьбы с взаимоблокировками. Применение графов Холта и матриц распределения ресурсов.
15. Стратегия избегания блокировок. Диаграмма траектории ресурсов. Алгоритм банкира для одного вида ресурсов.
16. Предотвращение блокировок путем исключения условий их возникновения.

## РАЗДЕЛ IV. УПРАВЛЕНИЕ ПАМЯТЬЮ

### Лекция 10. Методы управления памятью

Типы адресов и их преобразований. Классификация методов управления памятью. Методы управления памятью, использующие разделы. Страничный, сегментный и сегментно-страничный методы управления виртуальной памятью. Механизм свопинга. Принцип работы кэш-памяти.

В программе можно выделить три типа адресов: символьные имена, виртуальные адреса и физические адреса (рис.10.1). Символьные имена являются идентификаторами переменных в программе. Виртуальные адреса – это условные адреса, вырабатываемые транслятором исходного кода в объектный код. В простейшем случае транслятор может выполнять преобразование символьных имен непосредственно в физические адреса. Физические адреса – это номера ячеек в памяти. Данные номера выставляются центральным процессором на адресную шину при доступе к оперативной памяти.

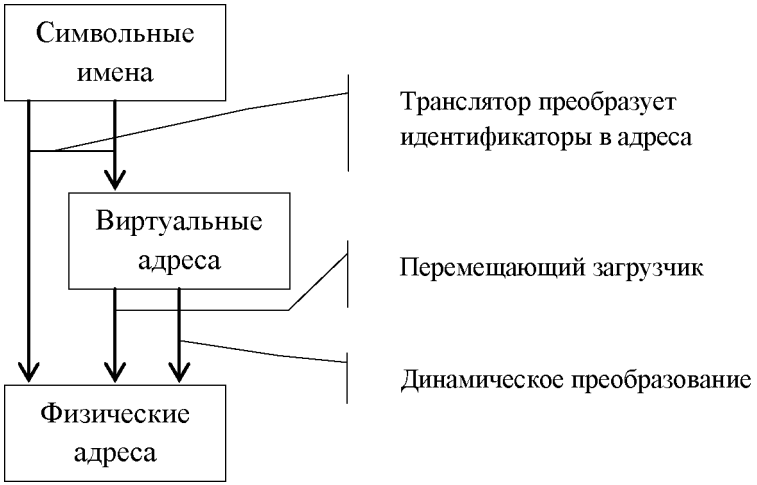



Рис. 10.1. Типы адресов и их преобразований

Преобразование виртуального адреса в физический адрес может выполняться двумя способами. При загрузке программы в память виртуальные адреса отображаются использованием перемещающего загрузчика. Его работа состоит из загрузки программы в последовательные ячейки, начиная с некоторого базового адреса, и настройки смещений внутри программы относительно этого адреса (рис. 10.2).

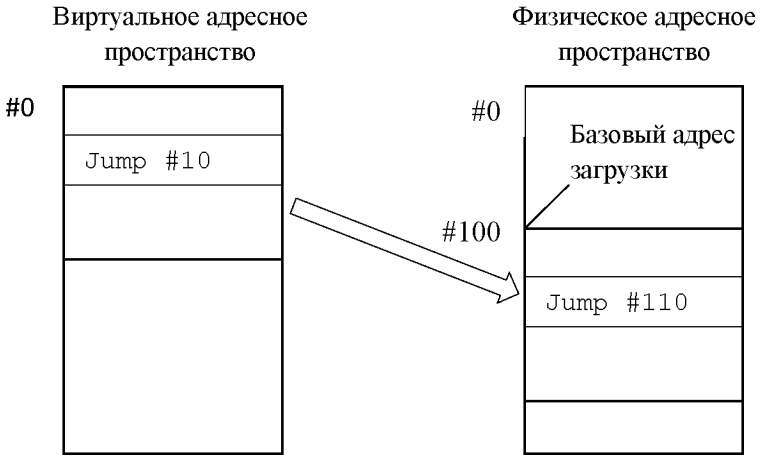



Рис. 10.2. Схема работы перемещающего загрузчика

Динамическое преобразование происходит при каждом обращении по виртуальному адресу в темпе обращений. Для такого преобразования используется аппаратура компьютера и операционная система.

**Методы управления памятью.** Существует два основных подхода к реализации процедур управления памятью: с использованием дискового пространства и без использования дискового пространства. Методы фиксированных разделов, динамических разделов и перемещаемых разделов не используют дисковое пространство. Страничный, сегментный и сегментно-страничный метод используют дисковую память. Специальными приемами управления памятью являются свопинг и кэширование.

Реализацию *метода фиксированных разделов* иллюстрирует рис. 10.3. Перед началом работы оператор разделяет физическую

память на разделы заданного размера. Поступающие в систему задачи либо занимают свободный раздел подходящего размера, либо попадают в очередь. Очередь может быть общей для всех разделов или индивидуальной для каждого раздела.

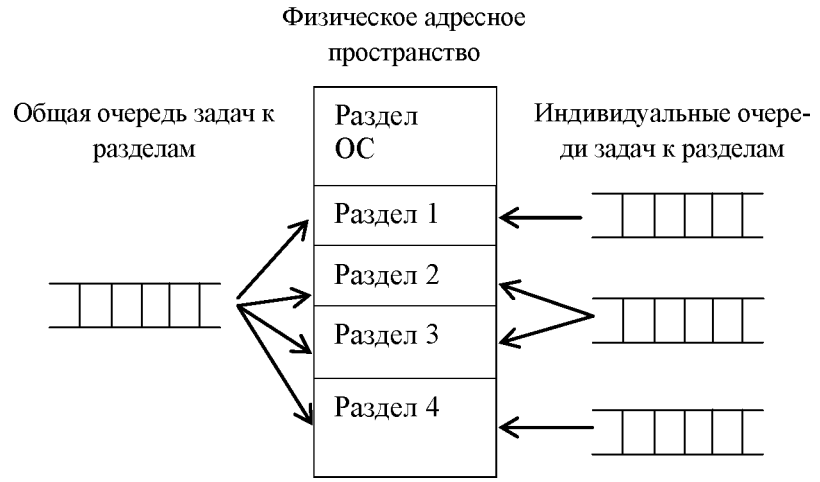



Рис. 10.3. Метод фиксированных разделов

Достоинством данного метода является простота реализации. Вместе с тем, присутствует недостаточная гибкость. Например, уровень мультипрограммирования жестко ограничен числом выделенных разделов. В современных операционных системах пользователь также может вручную управлять разделами внутри виртуального адресного пространства процесса. Пример такого приема описан в лекции 12.

*Динамические разделы.* При использовании динамических разделов распределение памяти по разделам заранее неизвестно. Операционная система ведет таблицы занятых и свободных разделов. При поступлении новой задачи для ее загрузки выбирается свободный раздел подходящего размера. Возможны разные стратегии выбора свободного раздела: первый по порядку подходящий, наименьший по размеру подходящий, наибольший по размеру подходящий раздел.

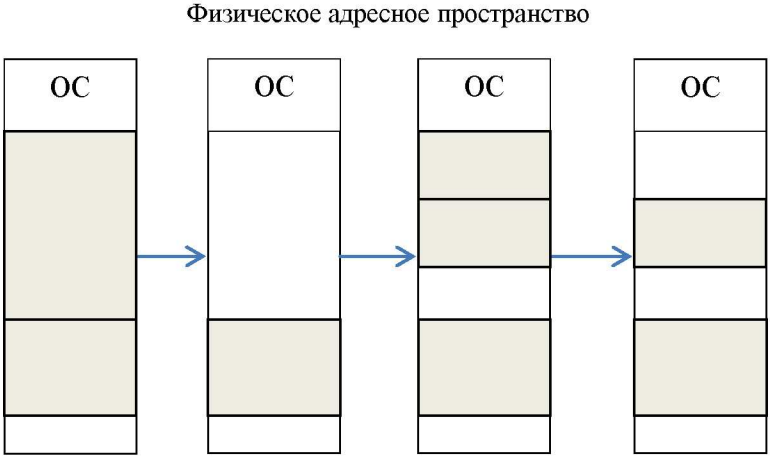



Рис. 10.4. Метод динамических разделов, возникновение фрагментации

Достоинства способа - это большая гибкость по сравнению с методом фиксированных разделов и отсутствие зависимости уровня мультипрограммирования от начального разбиения на разделы. Недостатком является эффект фрагментации памяти (рис. 10.4). Он заключается в появлении с течением времени большого числа небольших несмежных сегментов. Суммарный объем свободной памяти, содержащийся в таких сегментах, может быть большим, однако малый размер каждого отдельного сегмента не позволяет загрузить программу. Варьирование стратегий выбора свободного раздела может уменьшить фрагментацию. В современных операционных системах такой способ используется для управления кучей (динамической памятью) процесса.

*Перемещаемые разделы.* Данный способ расширяет управление динамическими разделами путем добавления процедуры сжатия: перемещения занятых разделов в одну последовательную область старших или младших адресов. В результате свободная память размещается в последовательно расположенных ячейках памяти. Достоинством такой организации является отсутствие фрагментации памяти. Однако в отличие от предыдущих способов нельзя использовать перемещающий загрузчик. Процедура сжатия может быть

затратной по времени. Поэтому обычно сжатие выполняется, когда не удается выполнить загрузку программы или во время простоя системы.

*Страничный способ.* Мотивом использования дискового адресного пространства является предоставление виртуальной памяти. Виртуальная память - это совокупность программно-аппаратных средств, позволяющих исполнять программы, размер которых превосходит доступную оперативную память компьютера. Для прикладного программиста механизм виртуальной памяти является прозрачным. То есть он не требует от него дополнительных усилий по управлению памятью.

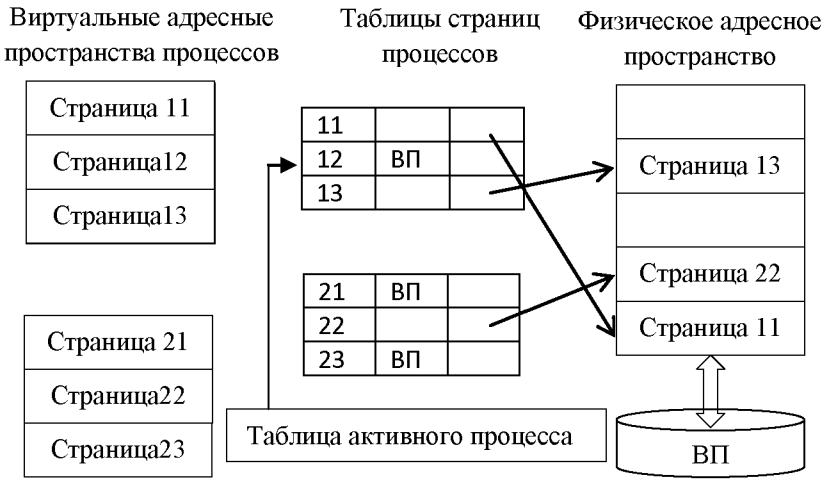



Рис. 10.5. Схема страничной организации памяти

При страничной организации памяти виртуальное пространство процессов делится на страницы равного размера. Размер страницы кратен степени двойки. Физическое адресное пространство делится на страницы такого же размера. Схема организации страничной памяти показана на рис.10.5.

Страницы процесса при таком разбиении размещаются в физическом адресном пространстве произвольным образом, необязательно по непрерывным адресам. Часть адресного пространства процесса может быть выгружена во внешнюю память.


Рис. 10.6. Схема преобразования адреса при страничной организации памяти

Обращение к памяти выполняется по следующей схеме. В случае если виртуальная страница присутствует в физической памяти, старшие биты виртуального адреса (P) содержат номер виртуальной страницы. По таблице страниц устанавливается соответствующий адрес физической страницы. Он конкатенируется с младшими k разрядами виртуального адреса. Получается физический адрес. Схема преобразования виртуального адреса в физический адрес показана на рис.10.6.

В случае если виртуальный адрес отсутствует в физической памяти (клетка ВП, содержимое располагается во внешней памяти) происходит страничное прерывание. Процесс переходит к ожиданию на системном объекте ядра, запускается процедура загрузки страниц, а контекст переключается на следующий готовый процесс.

Для загрузки страниц определяется место в физической памяти. Если оно есть, то выполняется загрузка, коррекция таблицы страниц, перевод процесса в готовое состояние. В противном случае, выбирается страница для вытеснения. Если страница была модифицирована, то предварительно необходимо записать страницу на диск, иначе содержание страницы переписывается загружаемой страницей. Кандидаты на выгрузку определяются управляющей информацией операционной системы.

*Сегментный способ.* При страничном распределении памяти адресное пространство делится механически на страницы равного размера. При сегментном распределении памяти размеры сегментов произвольны, сегменты снабжены атрибутами, определяющими способ их использования. Разделение программы на сегменты опре-

деляется программистом или компилятором. Атрибуты сегмента определяют способ его использования: для исполнения, чтения, записи. Также сегментное распределение позволяет разделять сегменты между процессами.

В таблице сегментов указывается базовый адрес сегмента в физической памяти, а не номер, как в случае страничного способа. Размер сегмента не фиксирован, в отличие от страничного способа адресации. Он хранится в таблице сегментов и используется для контроля выхода за границы сегмента. Из-за разницы в размерах сегментов, как и в случае с динамическими разделами для сегментного способа управления памятью характерно явление фрагментации памяти.

*Сегментно-страничный способ.* Цель сегментно-страничной организации памяти: решить проблему фрагментации физической памяти, сохраняя преимущество сегментного распределения, которое заключается в осмысленном распределении памяти.

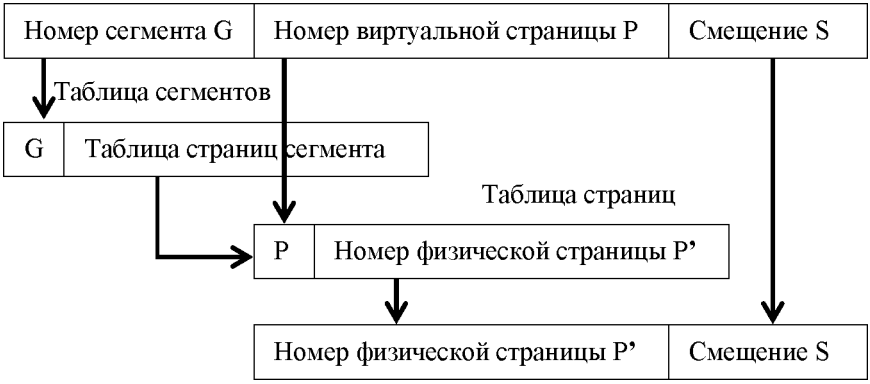



Рис. 10.7. Схема преобразования адреса при сегментно-страничной организации памяти

При сегментно-страничной организации виртуальный адрес состоит из 3 частей: номера сегмента g, номера страницы в сегменте p и смещения внутри страницы S. Номер сегмента преобразуется с использованием таблицы сегментов, а номер страницы – с использо-

ванием таблицы страниц (рис.10.7). В результате линейный физический адрес состоит из 2 частей: номера физической страницы n и смещения S.

**Свопинг.** Виртуализация памяти в ранних операционных системах осуществлялась на основе свопинга. Свопинг (swapping) – способ управления памятью, при котором образы процессов выгружаются на диск и возвращаются в оперативную память целиком. Свопинг представляет собой частный, более простой в реализации, вариант виртуальной памяти, позволяющий совместно использовать оперативную память и диск.

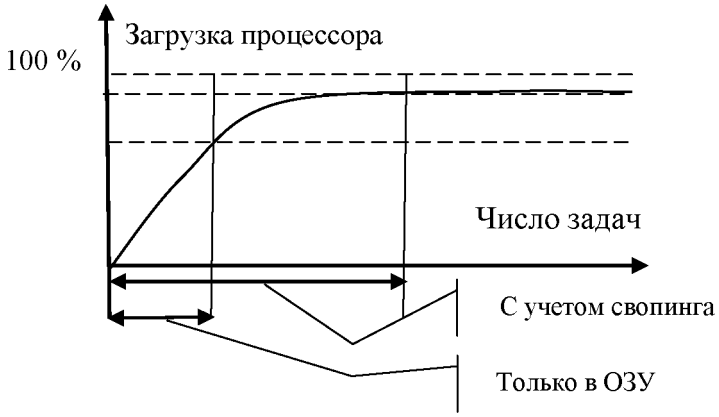



Рис. 10.8. Повышение загрузки процессора при использовании свопинга

Цель применения свопинга в ранних многозадачных операционных системах иллюстрирует рис.10.8. Если на компьютере одновременно выполняется большое число вычислительных задач и задач с интенсивным вводом/выводом, то удается добиться высокой эффективности использования центрального процессора. Это происходит за счет того, что в такой конфигурации на время выполнения ввода/вывода одной задачи процессор можно переключить на другую задачу. Так как положение участка насыщения на графики эффективности рис.10.8 обычно находилось за пределами числа задач, которое можно было разместить в оперативной памяти, ожидающие завершения ввода/вывода задачи было целесообразно сбрасывать на диск.

**Кэш-память** – способ организации совместной работы устройств хранения, различающихся стоимостью хранения единицы информации и скоростью доступа рис.10.9. Цель применения кэш-памяти: увеличить скорость доступа и понизить стоимость хранения за счет размещения часто используемой информации в более быстром запоминающем устройстве. Механизм кэша, аналогично страничному и сегментному способам управления памятью, прозрачен для пользователя.

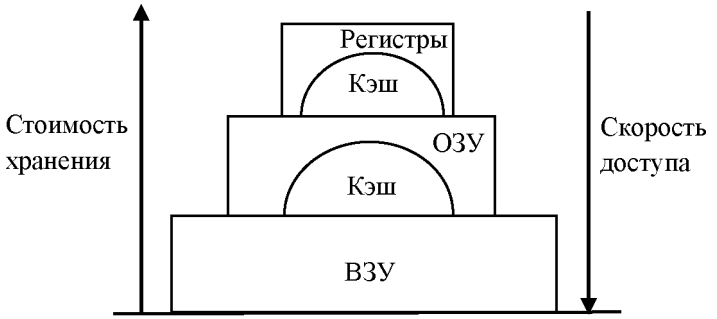



Рис. 10.9. К определению кэш-памяти

Принцип работы кэша оперативной памяти заключается в следующем (рис.10.10). Кэш одновременно с памятью «прослушивает» запрос центрального процессора, но если данные есть в самом кэше ответить успевает быстрее. Хранение информации в кэш-памяти организуется по ассоциативному принципу. Ключом для доступа (тегом) является адрес нескольких последовательно расположенных ячеек. Последствия такой организации памяти для программиста: память разбивается на кэш-линии (несколько последовательных ячеек, адресуемых одним тегом). Это приводит к тому, что в некоторых случаях для оптимизации скорости работы программ необходимо выполнять выравнивание данных по границе кэш-линии. Такая необходимость возникает, например, в случае возникновения эффекта ложного разделения см. лекцию 7.

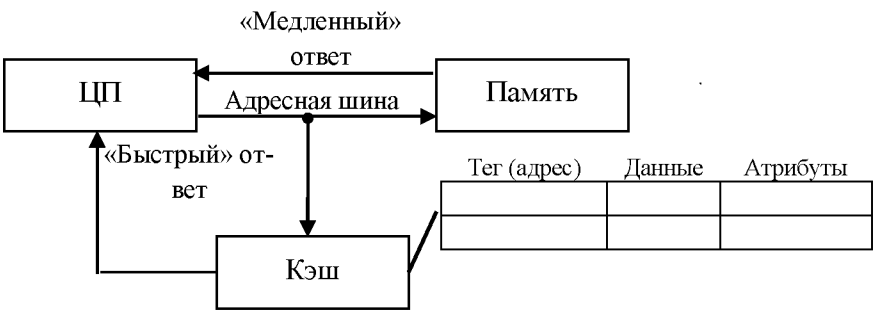



Рис. 10.10. Принцип работы кэш-памяти

Работа кэша основана на принципах пространственной и временной локальности, обеспечивающих высокий процент «попадания» в кэшр.Это приводит к снижению среднего времени доступа к памяти: $t_{доступа}=p \cdot t_{кэш} + (1-p) \cdot t_{памяти}$; p~90%.

Пространственная локальность - если в некоторый момент времени t происходит обращение по адресу A , то с большой долей вероятности в момент времени t+1 происходит обращение по адресу A±1.

Временная локальность – если в некоторый момент времени t произошло обращение по адресу A, то существует большая вероятность того, что в течение некоторого временного интервала (t, t+τ] снова произойдет обращение по адресу A.

## Лекция 11. Средства аппаратной поддержки управления памятью в архитектуре x86_32

Режимы работы микропроцессора x86_32. Сегментный механизм, структуры данных, схема преобразования виртуального адреса. Страничный механизм, структуры данных, схемы преобразования адреса. Средства вызова подпрограмм и задач.

**Режимы работы микропроцессора i386.** Общие принципы управления памятью с использованием внешней памяти представлены в лекции 10. Рассмотрим конкретный пример управления памятью в 32 разрядной архитектуре микропроцессоров семейства Intel

x86. Изложение ведется на примере микропроцессора i386. Старшие модели 32 разрядных процессоров Intel имеют сходную архитектуру.

Микропроцессор i386 имеет два режима работы: реальный (real mode) и защищенный (protected mode). В защищенном режиме доступны сегментный и страничный механизмы виртуальной памяти. В реальном режиме процессор работает как 16 разрядный процессор 8086, но с расширенным набором команд. Размер адресуемого пространства в данном режиме составляет 1 МБ. Для работы в адресном пространстве 1МБ надо сформировать адрес определенным образом, как показано на рис.11.1.

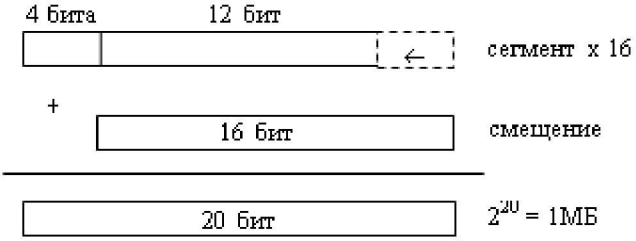



Рис. 11.1. Формирование адреса в реальном режиме

Реальный режим устанавливается по сбросу и используется для начальной инициализации системы. Путем записи управляющих битов в регистры командой MOV процессор переводится в защищенный режим. В защищенном режиме всегда включен сегментный механизм, и может быть установлен режим v86 (virtual 86), при котором процессор работает как несколько процессоров 8086 с общей памятью. Кроме того, в защищенном режиме может быть включен режим страничной адресации.

**Сегментный механизм.** В сегментном механизме управления памятью используются следующие структуры данных. Селектор – это структура данных, которая служит для выбора записи в глобальной или локальной дескрипторной таблице (рис.11.2). Селекторы хранятся в сегментных регистрах и в некоторых типах записей внутри самих дескрипторных таблиц. Дескрипторные таблицы – это аналог таблицы сегментов (см. лекцию 10). Однако на практике, кроме описания сегментов, в них содержится дополнительная информация.

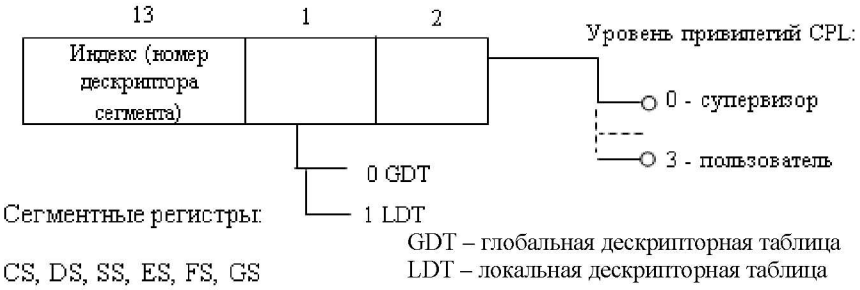



Рис. 11.2. Селектор

Регистр GDTR описывает положение глобальной дескрипторной таблицы в физической памяти (рис.11.3). В нем также указывается размер таблицы в байтах. Поскольку в сегментном регистре на указание индекса дескриптора сегмента отводится 13 бит, всего сегментов 213 = 8192.


Рис. 11.3. Регистр GDTR

Формат дескриптора сегмента данных и кода показан на рис.11.4.Размер дескриптора 8 байт; с учетом 8192 возможных сегментов, размер дескрипторной таблицы составит $8 \cdot 8192 = 2^{16} = 64$ КБ. Бит G определяет способ изменения размера сегмента (в байтах – G=0, или в страницах – G=1). Если G=0, то размер сегмента $2^{20} = 1$ МБ (совместимость с 8086, используется в режиме v86). Если G=1, то при 4КБ страницах, $4КБ \cdot 2^{20} = 4$ ГБ.

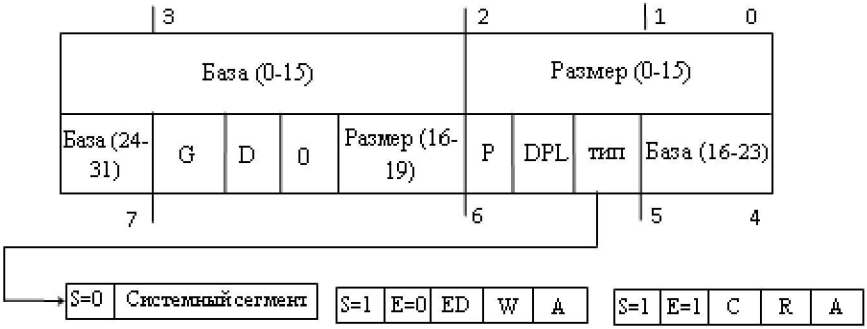



Рис. 11.4. Дескриптор сегмента данных и кода

Бит D определяет выравнивание сегментов по 32-битной (D=1) или 16-битной границе (D=0).

Бит P – бит присутствия сегмента в физической памяти. Если он не установлен, то сегмент выгружен на диск. Обращение к сегменту не возможно и приводит к прерыванию.

Биты DPL определяют уровень привилегий дескриптора, то есть возможность обращения к сегменту с данным уровнем привилегий задачи. Если CPL≤DPL – доступ разрешен, иначе возникает прерывание.

Тип сегментов определяет бит S. Если S=0, то данный дескриптор описывает системный сегмент или имеет специальное назначение. К системным сегментам, например, относятся LDT – локальная дескрипторная таблица, TSS – сегмент состояния задачи, в который отображается содержимое контекста при переключении задач. Дескрипторы специального назначения - ловушки (шлюзы) вызова, предназначенные для межсегментного вызова с повышением уровня привилегий.

Пользовательские сегменты имеют бит S=1. К пользовательским сегментам относятся сегменты данных E=0 и сегменты кода E=1. В пользовательских сегментах имеются следующие управляющие биты. Бит ED – бит распространения сегмента. Если ED=0 сегмент распространяется в сторону старших адресов, если ED=1 – в сторону младших адресов. В сторону младших адресов распространяются сегменты стека. Бит W – бит разрешения записи в сегмент. Бит A – бит доступа к сегменту. Если A=1 произошел доступ к сегменту.

Анализ этого бита позволяет оценить интенсивность обращений к сегменту для выбора наименее используемого сегмента с целью вытеснения его на диск. Бит C – бит подчинения. Если C=1, то проверка CPL≤DPL игнорируется, и можно вызвать более привилегированный сегмент кода, чем тот сегмент кода, из которого выполняется вызов. Наконец, бит R используется для указания на возможность чтения кодового сегмента.

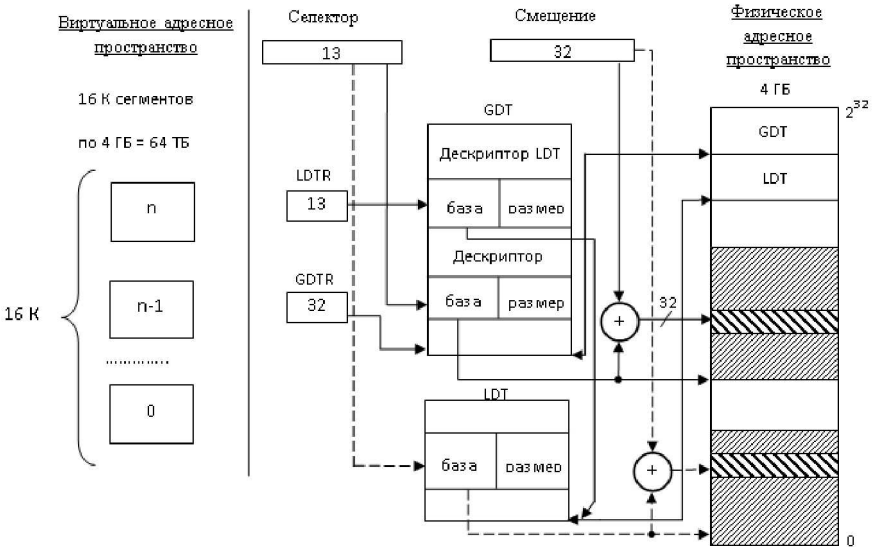



Рис. 11.5. Схема обращения к сегментам кода и данных

Рассмотрим схему обращения к сегментам кода и данных при использовании глобальных и локальных дескрипторных таблиц (рис.11.5). Если используется адресация из таблицы GDT, то индекс сдвигается влево на 3 разряда, так как дескриптор имеет размер 8 Б = $2^3$. Получается смещение в таблице GDT. Далее проверяется, не выходит ли смещение за границы таблицы. Для этого анализируются младшие 16 разрядов GDTR, хранящих ее размер. Далее берется 32-битный адрес GDT и складывается с этим смещением. В итоге получается 32-битный адрес дескриптора в физической памяти. При этом определяется, присутствует ли сегмент в физической памяти и разрешен ли доступ к нему. Если доступ разрешен, то берется базо-

вый адрес сегмента из дескриптора (база) и складывается со смещением из команды. Также на этой стадии производится контроль выхода за границу сегмента. Результатом является 32-битный адрес в физической памяти.

Если обращение идет через локальную дескрипторную таблицу LDT (см. соответствующий признак в селекторе), то используется регистр LDTR, который указывает на дескриптор LDT в глобальной дескрипторной таблице GDT. С использованием этого дескриптора вычисляется базовый адрес LDT в физической памяти. Далее преобразование происходит аналогичным способом.

Для увеличения скорости преобразования адреса каждому из шести сегментных регистров соответствует теневой 8-байтный регистр, хранящий дескриптор, соответствующий значению сегментного регистра.

Таким образом, для инициализации системы в защищенном режиме необходимо как минимум создать дескрипторную таблицу с одним входом и проинициализировать GDTR.

Если включен страничный механизм, то вычисленный 32-битный адрес подвергается дальнейшему преобразованию блоком управления страничной памятью.

**Страничный механизм.**Структура данных, используемая для страничного механизма управления памятью – это дескриптор страниц (рис.11.6).

| 20 | 3 | 2 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
|---|---|---|---|---|---|---|---|---|---|
| Номер физической страницы | AVL | O | D | A | PCD | PWT | U | W | P |

Рис. 11.6. Дескриптор страниц

Общее число страниц, адресуемых через дескриптор страниц, составляет $2^{20}$ = 1 М. Часть линейного адреса, отводимая под смещение внутри страницы, таким образом, составляет 32-20 = 12 бит. Размер страницы составляет $2^{12}$ = 4 К.

Поля в структуре дескриптора станиц имеют следующее назначение: AVL – резерв ОС; D – признак модификации; A – признак того, был ли доступ; PCD, PWT – биты управления кэшированием

страницы; U – пользователь/супервизор; W – разрешение записи в страницу; P – бит присутствия страницы в физической памяти.

Так как дескриптор занимает в памяти 4 байта, а всего дескрипторов 1 М, для хранения таблицы страниц в физической памяти потребуется 4 МБ оперативной памяти. Хранение таблицы страниц целиком привело бы к нерациональному использованию физической памяти. Поэтому в архитектуре i386 используется двухуровневый механизм преобразования линейного виртуального адреса в физический. Схема такого преобразования адреса показана на рис.11.7.

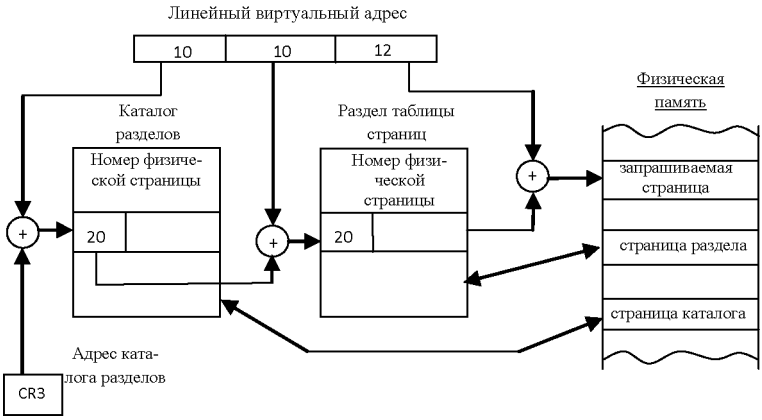



Рис. 11.7. Схема преобразования линейного виртуального адреса в физический

Идея механизма состоит в том, чтобы использовать страничный механизм для управления самой таблицей страниц. То есть, части таблицы страниц могут храниться во внешней памяти и подгружаться в оперативную память по мере необходимости.

Таблица страниц состоит из каталога разделов и разделов. Старшие 10 бит линейного виртуального адреса используются для адресации раздела внутри каталога разделов. Следующие 10 бит используются для адресации страницы внутри раздела. Таким образом, каталог разделов, так же как и раздел, может включать в себя 1024 ($2^{10}$) дескриптора. Размер дескриптора равен 4 Б и каталог разделов целиком умещается в физическую страницу (4 КБ). Каталог разде-

лов всегда присутствует в физической памяти, его адрес содержится в управляющем регистре CR3. Старшие 10 бит умножаются на 4 (т.к. дескриптор имеет размер 4 Б = $2^2$) и складываются с адресом в CR3, получается адрес дескриптора раздела в физической памяти. В нем содержится номер физической страницы раздела. С его использованием вычисляется адрес дескриптора запрашиваемой страницы внутри раздела. Наконец, этот дескриптор содержит адрес запрашиваемой физической страницы.

Для ускорения описанного преобразования линейного адреса в блоке управления страницами кэшируется 20 последних комбинаций номеров виртуальной и физической страницы.

**Средства вызова подпрограмм и задач.** Существуют внутрисегментные и межсегментные вызовы, которые осуществляются командами CALL и JMP. Внутрисегментные вызовы не имеют особенностей. Различаются межсегментные вызовы с использованием дескриптора сегмента кода, шлюза вызова и вызова с переключением задачи через TSS (сегмент состояния задачи). Схема непосредственного вызова через дескриптор кода показана на рис.11.8.

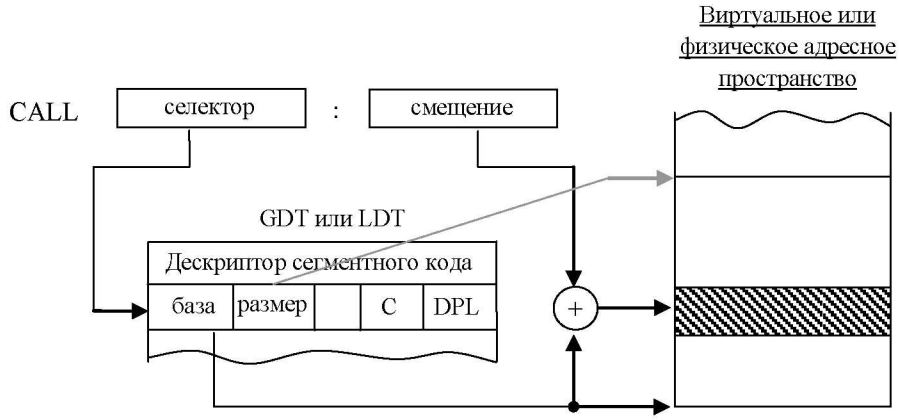



Рис. 11.8. Вызов через дескриптор кода

При вызове может указываться дескриптор сегмента кода. В этом случае возможны подчиненный и неподчиненный вызовы. При подчиненном вызове (C=1) можно вызвать сегмент кода с более высоким уровнем привилегий, чем CPL, при этом текущий уровень привилегий не изменяется. При неподчиненном вызове (C=0) вызов осуществляется только в случае CPL≤DPL. Недостаток этого механизма в том, что он не обеспечивает защиту операционной системы: можно задать произвольную точку входа в кодовый сегмент операционной системы; отсутствует защита ее стека вызовов.

Защищенный вызов с повышением уровня привилегий обеспечивает шлюз вызовов (рис.11.9).


Рис. 11.9. Формат дескриптора шлюза

На рис.11.9 используются следующие обозначения. Селектор – селектор сегмента кода, в котором находится вызываемая процедура или TSS. Смещение – точка входа в процедуру. Счетчик слов показывает, сколько слов требуется скопировать из стека текущего уровня привилегий в стек уровня привилегий вызываемой подпрограммы. Вызов разрешен, если CPL≤DPL, но DPL сегмента, на который указывает селектор, может быть любым. При удачном вызове процедура выполняется именно с указанным в требуемом сегменте уровнем привилегий DPL. Схема вызова через шлюз вызова показана на рис.11.10.

В команде CALL также может быть непосредственно указан селектор сегмента TSS, хранящий полное состояние задачи, на которую выполняется переключение. Состояние текущей задачи запоминается в сегменте TSS, на который указывает регистр TR.

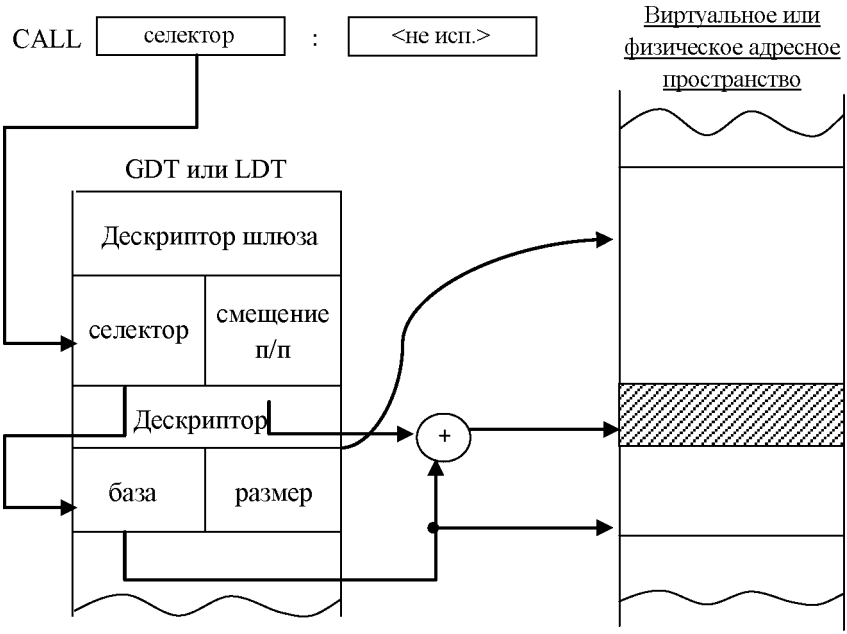



Рис. 11.10. Схема вызова через шлюз

## Лекция 12. Управление виртуальной памятью в ОС Windows

Структура виртуального адресного пространства процесса. Обзор методов управления памятью в ОС Windows. Применение функции VirtualAlloc. Обработка страничных прерываний с использованием структурированной обработки исключений. Файлы, отображаемые в память.

**Структура виртуального адресного пространства процесса.** Виртуальное адресное пространство процесса 32 разрядных версий Windows устроено следующим образом. Существует 3 варианта структур адресного пространства в зависимости от версии операционной системы и системных настроек.

Серверные версии ОС (например, Win2k Advanced Server) имеют следующее распределение памяти. В область старших адресов проецируется код и данные операционной системы. Однако прямой до-

ступ из процесса пользователя к ним невозможен и вызывает прерывание. Младшие 3 ГБ оперативной памяти принадлежат пользователю (рис.12.1).

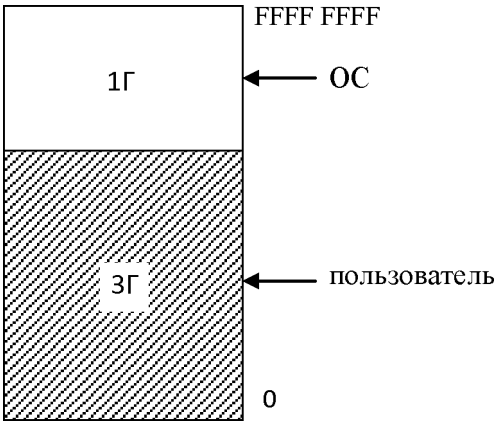



Рис. 12.1. Распределение памяти в системе Win2k Advanced Server

В «десктопных» 32 разрядных версиях (например, Win NT/2k) пользователю выделяется младшие 2 ГБ адресного пространства (рис.12.2).

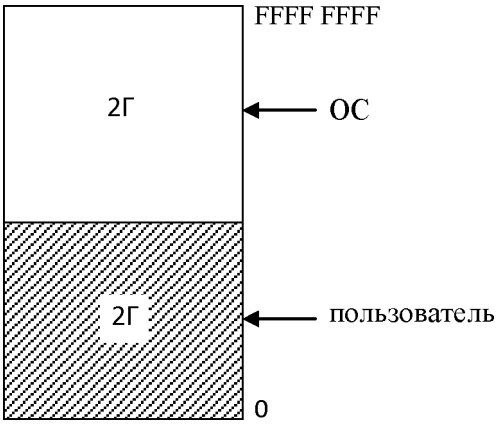



Рис. 12.2. Распределение памяти в системе Win NT/2k

Распределение памяти в ранних 32 разрядных операционных системах Win 95/98/ME показано на рис. 12.3.

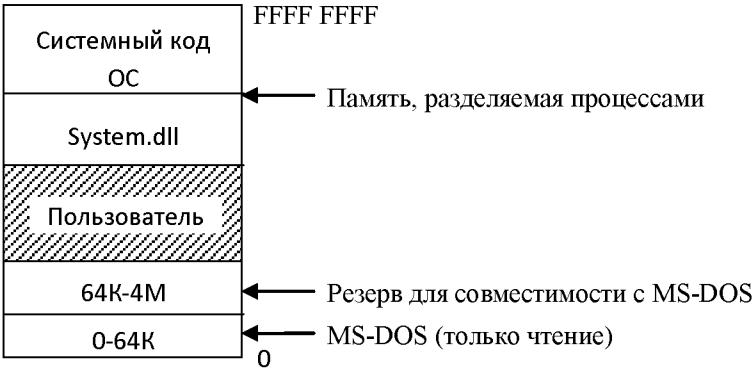



Рис. 12.3. Распределение памяти в системах Win 95/98/ME

Таким образом, в 32 разрядных операционных системах семейства Windows пользователь может выделить блок памяти, располагающийся по непрерывным адресам размером ~2-3 ГБ в зависимости от настройки и версии, так как код и данные операционной системы помещаются (проецируются) в адресное пространство каждого процесса. В Win 95/98/ME не используется защита памяти от ошибочных или злонамеренных действий пользователя. В версиях на базе ядра NT механизм защиты реализован полностью.

Средствами операционной системы поддерживаются несколько механизмов управления памятью. Простейшим является механизм динамической памяти. Процесс может иметь один или несколько разделов динамической памяти внутри пользовательской части виртуального адресного пространства (несколько куч процессов). К кучам может быть организован синхронизированный доступ из нескольких потоков. Кучи можно создавать динамически и удалять целиком. Они предназначены для создания большого количества относительно малых по размеру блоков памяти. Функции для работы с кучами имеют префикс Heap в своих названиях, например HeapAlloc.

Существуют некоторые специфические ситуации в практике программирования, когда использование возможностей библиотеки времени исполнения и куч операционной системы оказывается недостаточным. Например, при обработке массивов данных большого размера, не умещающихся в оперативной памяти целиком, в численном моделировании, при обработке большеформатных изображений. В данном случае необходимо воспользоваться средствами управления виртуальной памятью. В Windows такое управление реализуется функцией VirtualAlloc и другими вспомогательными функциями API. Совместно с функцией VirtualAlloc обычно используется механизм структурированной обработки исключений.

Для управления большими объемами данных, организации межпроцессного взаимодействия через общую (разделяемую) память используется механизм отображения файлов на виртуальную память процессов. На данном механизме основана работа самой операционной системы при загрузке исполняемых файлов и динамической компоновке библиотек.

Дополнительно 32 разрядным приложениям Windows доступен программный интерфейс Address Windowing Extensions (AWE), позволяющий получить доступ ко всему физическому адресному пространству компьютера (если его размер превышает 4 ГБ) через адресное окно в пределах пользовательских 2-3 ГБ в виртуальном адресном пространстве процесса. В современных 64 разрядных версиях операционных систем объем пользовательского адресного пространства, как правило, значительно превышает объем оперативной памяти, и в использовании механизма AWE не возникает необходимости.

В качестве примера работы с виртуальной памятью рассмотрим реализацию поиска простых чисел методом решета Эратосфена. В алгоритме используется массив флагов arr размером MAXNUM, где MAXNUM граница числового диапазона внутри которого ищутся все простые числа. Метод заключается в постепенном вычеркивании всех составных чисел установкой флагов arr[num]=1. Все не помеченные флагами числа будут являться простыми. Для нас в данном алгоритме представляет интерес организация сканирования массива размером около 1 ГБ.

Объявления и инициализация.

```
#include "stdafx.h"

DWORD i,j,num;
DWORD MAXNUM=1024*1024;
LPSTR arr;
DWORD dwPageSize;

VOID ErrorExit(LPTSTR);
INT PageFaultExceptionFilter(DWORD);

int main(intargc, char* argv[])
{
      BOOL bSuccess;
      SYSTEM_INFO sSysInfo;

      int num_MB=1;
      printf("Amount of memory in MB:");
      scanf("%d",&num_MB);
      printf("Getting %d MB...\n",num_MB);

      MAXNUM*=num_MB;

      GetSystemInfo(&sSysInfo);
      dwPageSize = sSysInfo.dwPageSize;

      arr =
(LPSTR)VirtualAlloc(NULL,MAXNUM,MEM_RESERVE,
            PAGE_NOACCESS);
      if (arr == NULL) ErrorExit("VirtualAlloc reserve
failed");
      else printf("VirtualAlloc successful\n");
```

Для работы программы нам потребуется подключить некоторые заголовочные файлы, среди которых <windows.h>. На массив флагов указывает переменная arr, размер массива хранится в переменной MAXNUM. При ее инициализации пользователю программы предлагается ввести размер массива в мегабайтах. Также в начале работы вычисляется размер страницы с использованием функции

GetSystemInfo и выполняется резервирование памяти размером MAXNUM в произвольной области виртуальной памяти процесса под массив флагов.

Резервирование необходимо по следующей причине. Фактически запущенной программе разрешен доступ только к незначительной части адресов из ее пространства в 2 ГБ. Это области, где находится код программы, ее ресурсы, статические переменные, стеки и кучи. Обращения к остальной части адресов пространства пользователя вызовет прерывания. Страницы, соответствующие этим адресам, отсутствуют в оперативной памяти. Более того, операционная система автоматически не поддерживает структуры для управления такими страницами. Управление всем 2 ГБ адресным пространством у каждого запущенного процесса было бы крайне накладно с точки зрения используемых ресурсов.

С точки зрения системного программиста виртуальная память процесса разделена на страницы, которые могут находиться в трех состояниях.

*Свободная страница (free).* Доступ к таким страницам запрещен. Операционная система не поддерживает структуры для управления свободными страницами, при обращении к ним возникает страничное прерывание.

*Зарезервированная страница (reserve).* Операционная система подготавливает необходимые управляющие структуры для этих страниц, но физическая память не выделена. При обращении также возникает страничное прерывание.

*Выделенная страница (committed).* Страница присутствует в физической памяти, при обращении прерывание не происходит.

Последняя часть инициализирующего кода выполняет резервирование блока памяти из последовательно расположенных страниц по произвольному начальному адресу в виртуальном адресном пространстве процесса. Адрес начала блока присваивается переменной arr.

```
__try
    {
            for(i=0;i<MAXNUM;i++){num=i;arr[num]=0;}

            for(i=2;i<MAXNUM;i++)
```

```
            {
                  num=i;if(arr[num])continue;
                  printf("%d\n",i);

      for(j=i*2;j<MAXNUM;j+=i){num=j;arr[num]=1;}
            }
      }
    __except ( PageFaultExceptionFilter(
              GetExcetionCode() ) )
    {
            ExitProcess(GetLastError() );
    }

      bSuccess = VirtualFree(arr,0,MEM_RELEASE);

      printf ("Release was %s.\n", bSuccess ?
         "successful" : "unsuccessful" );

      return 0;
}
```

Далее сам алгоритм реализуется обычным способом, однако он помещается внутри блока структурированной обработки исключений. В блоке выполняется обращение к массиву arr, как будто память действительно выделена. Цель такой организации кода: перехватить прерывания, возникающие при обращении к странице памяти, принадлежащей массиву arr; вручную выполнить загрузку страницы в оперативную память; передать управление в точку возникновения прерывания.

Стандартная конструкция C++ для перехвата исключений catch(ExceptionType C) {...}. В отличие от нее, фильтр исключения __except(...) содержит выражение, вычисляемое в момент исключения. Значение этого выражения управляет передачей управления при возникновении исключения. Особенностью структурированной обработки является наличие семантики продолжения, которую мы используем в примере.

Еще одной особенностью фильтра исключений и тела обработчика исключений является то, что в них используются специальные встроенные псевдо-функции (intrinsic function) обрабатываемые непосредственно компилятором. В примере такой функцией является функция GetExceptionCode(), которая возвращает код исключе-

ния. Ее использование бессмысленно вне контекста исключения и вызывает ошибку компиляции.

Код обработчика исключений выглядит следующим образом.

```
INT PageFaultExceptionFilter(DWORD dwCode)
{
  LPVOID lpvResult;
  DWORD  comSize;

  if (dwCode != EXCEPTION_ACCESS_VIOLATION)
  {
   printf("Exception code = %d\n", dwCode);
   return EXCEPTION_EXECUTE_HANDLER;
  }

  //printf("Exception is a page fault\n");

  if(MAXNUM-num>dwPageSize)comSize=dwPageSize;
  else comSize=MAXNUM-num;

  lpvResult = VirtualAlloc(
   &arr[num],comSize,MEM_COMMIT,PAGE_READWRITE);

  if (lpvResult == NULL )
    {
       printf("VirtualAlloc failed\n");
       return EXCEPTION_EXECUTE_HANDLER;
    } else {
      //printf ("Allocating another page.\n");
    }

  return EXCEPTION_CONTINUE_EXECUTION;
}
```

Обработчик получает код исключения и возвращает значение, определяющее передачу управления после обработки. Мы используем следующие возвращаемые значения: EXCEPTION_EXECUTE_HANDLER для возврата управления в тело обработчика исключений в секции __except и аварийному завершению; EXCEPTION_CONTINUE_EXECUTION для возврата к месту возникновения страничного прерывания и повторной попытки об-

ращения к памяти. Для передачи управления следующему обработчику в иерархии может использоваться значение EXCEPTION_CONTINUE_ SEARCH.

Загрузку страницы в физическую память выполняет вызов VirtualAlloc с параметром MEM_COMMIT. Код загрузки имеет следующие особенности. Страница, в которой произошло исключение, вычисляется с использованием глобальной переменной num, выполняющей роль индекса по массиву arr. Доступ к массиву выполняется только с использованием данной переменной. Способ, не требующей глобальной переменной num, состоит в получении адреса, по которому возникло исключение при помощи еще одной встроенной псевдо-функции GetExceptionInformation().

Другой особенностью является коррекция запрашиваемого к загрузке размера памяти comSize. Она необходима при сканировании последней страницы массива, чтобы не выйти за пределы зарезервированной под массив arr области.

Наконец, после использования, аналогично кучам, виртуальная память требует очистки, которая выполняется вызовом VirtualFree(arr,0,MEM_RELEASE).

Кратко рассмотрим работу с файлами, отображаемыми в память процесса. Работа состоит из следующих этапов. Вначале нужно получить описатель файла, куда при работе механизма отображения будут сбрасываться страницы памяти. Данный этап можно пропустить, если используется файл подкачки.

Далее создается объект отображения при помощи вызова

```
hMapFile = CreateFileMapping(hFile,// описатель файла
NULL, // безопасность
PAGE_READWRITE, // желаемый доступ
0, 0, // размер объекта и файла
"name"); // имя объекта
```

Сконструированный объект ядра с описателем hMapFile может использоваться для организации взаимодействия процессов через разделяемую память. Для этого нужно передать описатель другому процессу рассмотренными ранее способами.

Далее выполняется собственно проецирование при помощи вызова вида

```
lpMapAddress = MapViewOfFile(hMapFile,
FILE_MAP_ALL_ACCESS, 0, 0, 0).
```

В приведенном примере отображается весь файл целиком. Работа с файлами большого размера осуществляется через адресное окно, о чем говорит название функции (map view of file). Функция возвращает адрес начала области памяти, в которую было выполнено отображение файла. Косвенное обращение по этому адресу, например

```
*((LPDWORD) pMapAddress) = 1000;
```

позволяет, как считывать, так и записывать информацию непосредственно в файл без использования функций ввода-вывода. Запись-чтение выполняется механизмом управления страничной памятью (лекции 10, 11).

После окончания работы с отображением выполняется очистка при помощи вызова функции UnMapViewOfFile и закрытия описателей созданных объектов ядра вызовом CloseHandle.

### Экзаменационные вопросы по разделу IV

1. Методы управления памятью без использования внешней памяти. Фиксированные, динамические и перемещаемые разделы.
2. Методы управления памятью с использованием внешней памяти. Сегментный, страничный, сегментно-страничный способ.
3. Назначение, принцип работы механизма свопинга.
4. Назначение, принцип работы механизма кэширования.
5. Реализация сегментного механизма управления памятью в процессорах семейства x86_32.
6. Реализация страничного механизма управления памятью в процессорах семейства x86_32. Размер и основные поля структур данных, особенности реализации.
7. Средства ОС Windows для управления виртуальной памятью процесса. Функция VirtualAlloc. Структурированная обработка исключений. Файлы, отображаемые в память.

## РАЗДЕЛ V. ОРГАНИЗАЦИЯ ВВОДА - ВЫВОДА

### Лекция 13. Введение во взаимодействие с внешними устройствами

Типы внешних устройств. Способы программного взаимодействия с внешними устройствами: циклический опрос, прерывания, прямой доступ к памяти. Архитектура программного обеспечения ввода-вывода. Файловая система. Логическая организация файлов. Физическая организация файлов.

**Типы внешних устройств.** Устройства ввода-вывода можно разделить на два класса: блок-ориентированные, байт-ориентированные. Типичным блок-ориентированным устройством является диск. Обмен информацией с дисками и адресация информации выполняется блоками. Для байт-ориентированных устройств характерна потоковая передача информации и отсутствие адресации. Клавиатура, манипулятор-мышь, сетевой адаптер представляют данную разновидность внешних устройств. Имеются внешние устройства, которые сложно отнести к какому либо из перечисленных классов. Например, программируемый таймер и другие устройства, информирующие компьютер о наступлении внешних событий.

Какой программный механизм используется для передачи информации из внешних устройств в операционную систему? У внешних устройств имеются специальные регистры, в которые можно записывать и считывать значения.

Обращение к регистрам контроллера может выполняться двумя способами. Регистры внешних устройств могут отображаться на память компьютера. Программное взаимодействие при этом происходит с использованием команды MOVE и других аналогичных команд пересылки. Достоинством данного способа является удобство программирования. В языках программирования высокого уровня, в которых предусмотрены механизмы обращение к памяти (например, C, C++, Pascal), не требуется никаких специальных механизмов для взаимодействия внешними устройствами с регистрами, отображенными на память. Также этот способ взаимодействия подразумевает высокую скорость передачи данных. Типичный пример такого взаи-

модействия - видеокарты. Недостатком является необходимость усложнения аппаратуры компьютера по следующим причинам. Требуется согласование скорости с внешними устройствами, так как обычно устройства ввода-вывода обладают значительно меньшей пропускной способностью, чем оперативная память. Необходимо определять границы кэшируемых областей памяти, так как отображенные на внешние устройства участки памяти кэшировать нельзя. Возникающую неоднородность памяти необходимо поддерживать на уровне микросхем вспомогательной логики: контроллеров-концентраторов памяти и контроллеров-концентраторов ввода-вывода (т.н. северный и южный мосты в архитектуре Intel). Кроме этого сужается диапазон адресуемой физической памяти.

Другой возможностью программного взаимодействия является использование специального пространства ввода-вывода, не пересекающегося с оперативной памятью. Доступ к регистрам внешних устройств в нем осуществляется специальными командами чтения IN и записи OUT регистров внешних устройств. В данном случае упрощается аппаратная организация взаимодействия с внешними устройствами и несколько усложняется программное взаимодействие. Необходимы ассемблерные вставки в код на языке высокого уровня, поддержка компилятором или специальные функции в библиотеке времени исполнения.

Процедура организации взаимодействия с внешними устройствами может организовываться тремя способами.

**Циклический опрос.** Команда на выполнение некоторой операции записывается во входной регистр внешнего устройства. Далее в цикле считывается и анализируется значение выходного регистра, до тех пор, пока не будет считан признак завершения операции. Недостатком данного способа является неэффективное использование процессора и шины памяти (данных).

**Взаимодействие по прерыванию.** Команда на выполнение некоторой операции записывается во входной регистр внешнего устройства. Далее процессор может выполнять другие полезные действия одновременно с работой внешнего устройства или остановиться, при этом не блокировать системную шину. Когда операция внешнего устройства завершена, внешнее устройство самостоятельно извещает процессор, инициируя прерывание. В ответ на преры-

вание процессор запоминает адрес текущей команды и переходит к выполнению специальной процедуры обработки прерывания. В ней организуется считывание уже готовой информации из выходных регистров внешнего устройства. Далее происходит возврат управления по сохраненному адресу прерванной команды.

Этот способ является более эффективным с точки зрения использования аппаратных ресурсов. Однако он требует сложной программной реализации: специальной настройки таблицы векторов обработки прерывания; нелинейной организации потока управления в программе; некоторого механизма планирования действий процессора, чтобы исключить простой процессора в течение ввода-вывода.

**Взаимодействие с использованием прямого доступа к памяти.** В данном способе передача информации из внешнего устройства в оперативную память производится не с использованием центрального процессора, а напрямую специальным контроллером прямого доступа (DMA-контроллер). Контроллер самостоятельно реализует описанную во втором способе процедуру взаимодействия с внешним устройством, а по окончании передачи данных прерыванием извещает центральный процессор о завершении взаимодействия. Преимуществом способа является еще большая эффективность, особенно при передаче пакетов данных. Способ позволяет контроллеру обращаться к системной шине, пока сам процессор выполняет команду. Очевидно, что способ требует дополнительных усилий по программированию ввода-вывода.

**Архитектура программного обеспечения ввода-вывода.** Для автоматизации рутинных низкоуровневых процедур в операционных системах реализуется специальная подсистема ввода-вывода, примерная архитектура которой показана на рис.13.1.

Рассмотрим компоненты, показанные на рисунке. Система обработки прерываний находится на нижнем уровне иерархии. Управление прерываниями выполняет сама операционная система. При возникновении прерываний происходит вызов драйверов внешних устройств. Драйверы выполняют обработку запросов с уровня пользователей или прерываний и имеют доступ к регистрам контроллеров внешних устройств. Имеется возможность синхронизации работы драйверов с использованием специальных мьютексов режима ядра.

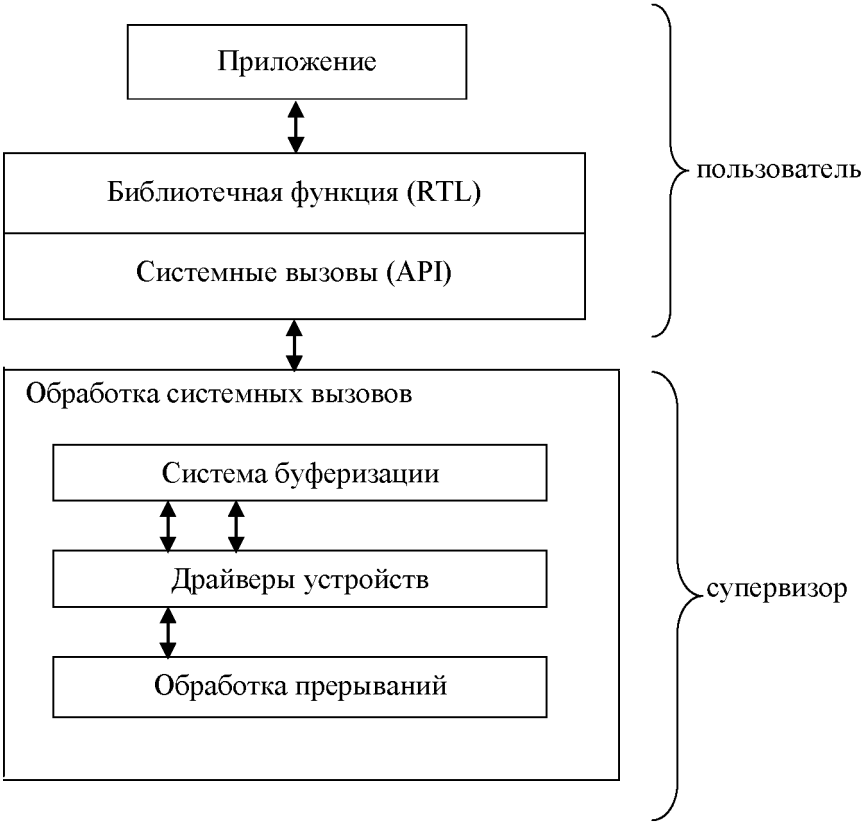



Рис. 13.1. Архитектура подсистемы ввода-вывода

В операционных системах имеется независимый от устройств уровень «системы буферизации», типичными функциями которого являются обеспечение независимого размера блоков данных, защита, общий интерфейс к драйверам устройств и их именование, распределение/освобождение устройств, уведомление об ошибках. Этот уровень единообразно подготавливает блоки для передачи запросов драйверам от вышележащих слоев и получения ответа.

Обращение к функциям ввода-вывода, как и к другим сервисам операционной системы, осуществляется посредством системных вызовов. Иногда они задействуются непосредственно или, как в ОС Windows, через функции интерфейса прикладного программирова-

ния (OpenFile, CreateFile, WriteFile), служащего для унификации различных версий операционной системы.

Выше по иерархии располагается библиотека поддержки времени исполнения (RTL), тоже содержащая функции ввода-вывода (printf, scanf и т.д.). Функции библиотеки RTL используют функции вызовов интерфейсов прикладного программирования API, часто являясь просто их обертками. В C++ функции потокового вывода <<, >> основаны на printf, scanf. Зная о такой организации ввода-вывода в некоторых языках (C, C++), можно сократить размер исполняемого файла, работая напрямую с вызовами API операционной системы. Для этого можно использовать функции ввода-вывода API, не компонуя соответствующие функции из RTL.

**Файловая система.** Важнейшими типами внешних устройств являются разнообразные накопители данных. Необходимые удобные абстракции для работы с ними реализуются файловыми системами.

Файловая система – часть операционной системы, назначение которой – обеспечить удобный интерфейс для работы с данными, хранящимися на дисках. Файлы – именованный набор данных. Файл может идентифицироваться по имени. Обычно файлы могут иметь одинаковые имена. Уникальность файла определяется составным именем, включающим символьные имена каталогов. Файл может идентифицироваться уникальным дескриптором. Если файловая система представлена в виде сети, файл может находиться в нескольких каталогах сразу.

Кроме обычных файлов с данными имеются файлы, ассоциированные с устройствами ввода-вывода. Это позволяет выполнять ввод-вывод посредством программного интерфейса доступа к файлам. Каталоги с файлами также являются специальным видом файлов. Каталоги файлов хранят атрибуты файла. Для организации защиты файлов от несанкционированного доступа, аналогично другим объектам операционной системы, используется избирательный или мандатный механизм безопасности.

Внешние устройства хранения имеют следующий простой интерфейс доступа к данным: по указанному адресу можно записать или считать блок данных фиксированного размера (в несколько килобайт). Например, для доступа к блоку на диске требуется указать его физическое положение на поверхности носителя (номер поверхности, номер цилиндра (дорожки), номер сектора). Поэтому файло-

вая система должна организовывать некоторый произвольный доступ, обеспечивая отличные от блочной модели способы работы с данными. Это логическая организация файлов. Также требуется поддерживать некоторую организацию физических блоков внутри файлов. Это физическая организация файлов.

**Логическая организация файлов.** Приведем некоторые распространенные модели файлов.

*Последовательность байт.* Это типовая модель доступа, используется как произвольный, так и последовательный доступ к хранимым данным.

*Последовательность блоков переменной длины.* Эта модель позволяет по логическому номеру блока адресовать его содержимое. В некоторых файловых система используется гибридная модель, когда в файле содержится несколько независимых последовательностей байт (потоков данных). Это может оказаться удобным, например, для хранения видео, звука и субтитров в отдельных потоках средствами файловой системы (рис.13.2).

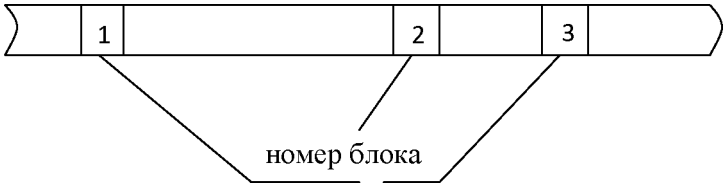



Рис. 13.2. Последовательность блоков переменной длины

*Упорядоченная или неупорядоченная последовательность ключей.* По ключу адресуется ассоциированный с ним блок данных (рис.13.3). Если последовательность ключей упорядочена, то имеется возможность выбрать следующий блок по порядку относительно текущего блока. Данная логическая организация удобна для управления и организации доступа к файлами очень больших размеров.


Рис. 13.3. Последовательность ключей

**Физическая организация файлов.** Рассмотрим некоторые распространенные приемы физической организации файлов.

*Последовательность блоков* (непрерывное размещение). Блоки, принадлежащие файлу, следуют по порядку их физических адресов. В каталоге указывается адрес первого блока, количество блоков или последний блок. Достоинство этого способа – простота, а основная проблема – фрагментация блоков. Этот способ, например, находит применение в файловых системах длительного хранения, в частности ленточных запоминающих устройствах.

*Связанный список блоков.* Простейший способ преодолеть фрагментацию (рис.13.4).

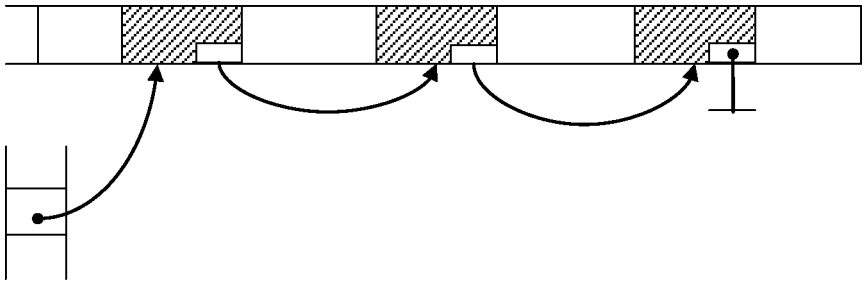


Рис. 13.4. Связанный список блоков

Проблемой является большое время доступа к информации в конце файла. Все блоки нужно последовательно перебрать. Информация о следующем блоке является частью блока, на это уходит некоторое количество памяти.

*Связанный список индексов.* Используемый на практике способ организации файловых систем относительно небольшого размера (рис.13.5). Он основан на применении таблицы размещения файла (File Allocation Table, FAT). Файловые системы, использующие этот способ: FAT12, FAT16, FAT32, VFAT. Метод предложен Microsoft.

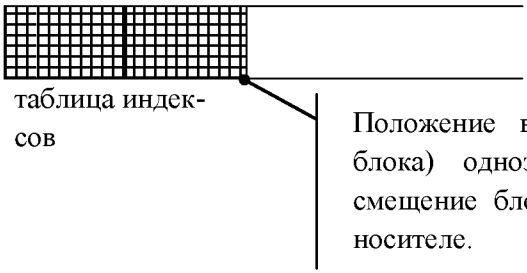



Рис. 13.5. Связанный список индексов

Идея заключается в переносе информации о следующем блоке в отдельную таблицу в начальных блоках физического носителя. Индекс блока в таблице и его физическое положение на диске связаны линейным преобразованием. Блоки таблицы индексов считываются в оперативную память. Благодаря этому можно быстро перебрать цепочку индексов блоков файла в памяти и запросить нужный блок на диске. Поэтому эффективность произвольного доступа оказывается высокой. Преимуществом также является целостность файловой системы, поддерживаемая несколькими копиями таблицы индексов и хранением информации о состоянии каждого физического блока в таблице индексов. Недостатком является рост размера требуемой служебной информации при увеличении размеров файловой системы (фактически предел 8 тебибайт при 2 КБ кластерах).

*Непосредственное перечисление блоков* и способы, использующие сбалансированные деревья (рис.13.6).

Это наиболее распространенная группа способов физической организации файлов. Часть физических блоков в произвольной области дискового пространства отводится под ссылочные структуры для доступа к блокам с данными. Например, на рисунке показана структура таких блоков. В корневом блоке, адрес которого записывается в каталоге, хранится 13 полей, содержащих адреса других блоков. Первые 10 полей непосредственно адресуют блоки файла. Если их недостаточно, то используется следующее поле, адресующее блок, в котором может быть записано 256 адресов следующих блоков файла. Если размер файла превышает 266 блоков, то используется двенадцатая запись корневого блока, с использованием которой физические блоки адресуются уже с двойным уровнем косвен-

ности и так далее. Общее количество физических блоков в файле, таким образом, может достигать 16.843.018.

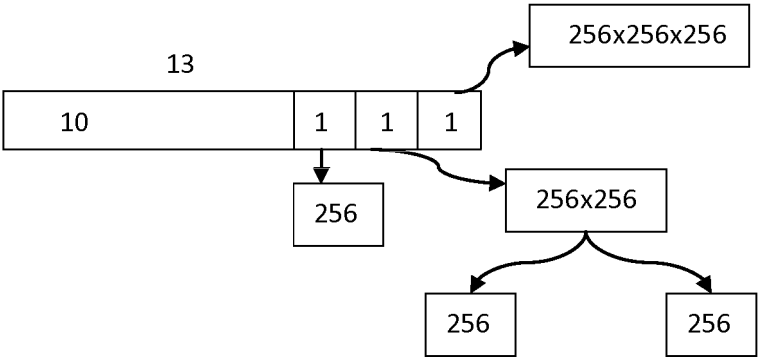



Рис. 13.6. Непосредственное перечисление блоков

В настоящее время разработано большое количество файловых систем различного назначения и устройства. Помимо логической и физической организации при рассмотрении архитектур файловых систем обсуждаются методы обеспечения надежности (например, журналирование), повышения скорости доступа (экстенты), хранения больших объемов данных (распределенные параллельные файловые системы), защиты данных (шифрованные файловые системы) и другие.

Экзаменационные вопросы по разделу V

1. Архитектура программного обеспечения ввода-вывода. Средства программного взаимодействия с внешними устройствами. Архитектура программного стека, функции слоев.
2. Общие принципы организации файловых систем и файлов. Идентификация файлов в файловых системах, логическая и физическая организация файлов.

## Литература

1. Гордеев, Александр Владимирович. Операционные системы [Текст] : учеб. для вузов по направлению подгот. бакалавров и магистров «Информатика и вычисл. Техника» и направлению подгот. дипломир. специалистов «Информатика и вычисл. Техника» / А. В. Гордеев. - 2-е изд. – СПб.: Питер: Питер принт, 2005. – 415 с.

2. Олифер, Виктор Григорьевич. Сетевые операционные системы [Текст] : учеб. пособие для вузов по направлению подгот. дипломир. специалистов «Информатика и вычисл. техника» / В. Г. Олифер, Н. А. Олифер. – СПб.: Питер: Питер Пресс, 2007. – 538 с.

3. Столлингс, Вильям. Операционные системы [Текст] :внутрен. устройство и принципы проектирования / Вильям Столлингс; пер. с англ. – М. [и др.] : Вильямс, 2004. – 843 с.

4. Бэкон, Джин. Операционные системы [Текст] : парал. и распредел. системы / Джин Бэкон, Тим Харрис; пер. с англ. – СПб.: Питер, 2004. – 799 с.

5. Таненбаум, Эндрю. Современные операционные системы [Текст] / Э. Таненбаум. – 2-е изд., перераб. и испр. – СПб.: Питер: Питер Пресс, 2007. – 1037 с. – (Классика computer science).

6. Карпов, Владимир Ефимович. Основы операционных систем [Текст] : курс лекций : учеб. пособие : для вузов по специальности 351400 «Прикладная информатика»/ В. Е. Карпов, К. А. Коньков; под ред. В. П. Иванникова; Интернет- ун-т информ. технологий. – М.: ИНТУИТ. ру, 2004. – 628 с.

7. Дейтел, Харви М. Операционные системы [Текст]: [в 2 т.] / Х.М. Дейтел, П.Д. Дейтел, Д.Р. Чофнес; под ред. С.М. Молявко; пер. с англ. – М.: Бином-Пресс, 2006 - Т. 1: Основы и принципы, – 2006. –1023 с.

8. Дейтел, Харви М. Операционные системы [Текст]: [в 2 т.] / Х. М. Дейтел, П. Д. Дейтел, Д. Р. Чофнес; пер. с англ. ред. С. М. Молявко. – М.: Бином; Королев: Бином-пресс, 2006 – Т. 2: Распределенные системы, сети, безопасность: переводное издание, – 2006. – 704 с.

9. Таненбаум, Эндрю. Операционные системы. Разработка и реализация / Э. С. Таненбаум, А. Вудхалл; пер. с англ. – СПб.; М.; Нижний Новгород: Питер, 2007. – 703 с.

Учебное издание

***Востокин Сергей Владимирович***

# ОПЕРАЦИОННЫЕ СИСТЕМЫ

*Учебник*

Редакторская обработка Ю.Н. Литвинова
Доверстка А.В. Ярославцева

Электронный ресурс.
Арт. – ЭР5(Д2)/2012.

Самарский государственный
аэрокосмический университет.
443086 Самара, Московское шоссе, 34.

---

Изд-во Самарского государственного
аэрокосмического университета.
443086 Самара, Московское шоссе, 34.