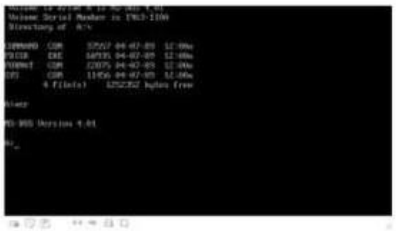
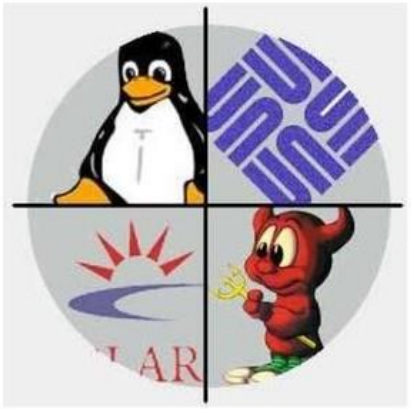
Частное профессиональное образовательное учреждение
**«СЕВЕРО-КАВКАЗСКИЙ КОЛЛЕДЖ ИННОВАЦИОННЫХ ТЕХНОЛОГИЙ
(ЧПОУ «СККИТ»)**

***А.М. Жуков***

# ОПЕРАЦИОННЫЕ СИСТЕМЫ И СРЕДЫ

Учебное пособие



Пятигорск-2022

## СОДЕРЖАНИЕ

**Общие сведения об операционных системах**

План

1 Определение операционной системы (ОС). Место ОС в программном обеспечении вычислительных систем

2. Эволюция операционных систем

3. Назначение, состав и функции ОС

4. Архитектуры операционных систем

5. Классификация операционных систем

6. Эффективность и требования, предъявляемые к ОС

1. Определение операционной системы (ОС). Место ОС в программном обеспечении вычислительных систем

1946 г. – ENIAC (Electronic Numerical Integrator and Computer) – полное отсутствие какого-либо ПО, программирование путем коммутации устройств.

Начало 50-х г. – появление алгоритмических языков и системного ПО.

Усложнение процесса выполнения программ:

1. Загрузка нужного транслятора.

2. Запуск транслятора и получение программы в машинных кодах.

3. Связывание программы с библиотечными подпрограммами.

4. Запуск программы на выполнение.

5. Вывод результатов работы на печатающее или другое устройство.

Для повышения эффективности использования ЭВМ вводятся операторы, затем разрабатываются управляющие программы – мониторы - прообразы операционных систем.

1952 г. – Первая ОС создана исследовательской лабораторией фирмы General Motors для IBM-701.

1955 г. – ОС для IBM-704. Конец 50-х годов: язык управления заданиями и пакетная обработка заданий.

ОПЕРАЦИОННАЯ СИСТЕМА : - это набор программ, контролирующих работу прикладных программ и системных приложений и исполняющих роль интерфейса между пользователями, программистами, приложениями и аппаратным обеспечением компьютера.

ОПЕРАЦИОННАЯ СРЕДА- этопрограммная среда, образуемая операционной системой, определяющая интерфейс прикладного программирования (API) как множество системных функций и сервисов (системных вызовов), предоставляемых прикладным программам.

ОПЕРАЦИОННАЯ ОБОЛОЧКА- часть операционной среды, определяющая интерфейс пользователя, его реализацию (текстовый, графический и т.п.), командные и сервисные возможности пользователя по управлению прикладными программами и компьютером

2 Расположение ОС в иерархической структуре программного и аппаратного обеспечения компьютера


Эволюция ОС

1963 г. – ОС MCP (Главная управляющая программа) для компьютеров B5000 фирмы Burroughs: мультипрограммирование, мультипроцессорная обработка,виртуальная память, возможность отладки программ на языке исходного уровня, сама ОС написана на языке высокого уровня.

1963 г. – ОС CTSS (Compatible Time Sharing System – совместимая система : разделения времени для компьютера IBM 7094 – Массачусетский технологический институт.

1963 г. – ОС MULTICS (Multiplexed Information and Computing Service) – Массачусетский технологический институт.

1974 г. – (UNICS) UNIX (Uniplexed Information and Computing Service) для компьютера PDP-7, публикация статьи Ритчи (C) и Томпсона.

1981 г. – PC (IBM), DOS (Seattle Computer Products) – MS DOS (Б. Гейтс).

1983г. – Apple, Lisa с Apple, Lisa с GUI (Даг Энгельбарт – Стэнфорд).

1985 г. – Windows, X Windows и Motif (для UNIX).

1987 г. – MINIX (Э. Таненбаум) – 11800 стр. C и 800 ассемблер (микроядро – 1600 C и 800 ассемблер)

1991 г. – Linux (Линус Торвальдс).

3. Назначение, состав и функции ОС

Назначение

1. Обеспечение удобного интерфейса между приложениями и пользователями, с одной стороны, и аппаратурой компьютера с другой, за счет предоставляемых сервисов:

1.1. Инструменты для разработки программ

1.2. Автоматизация исполнения программ

1.3. Единообразный интерфейс доступа к устройствам ввода-вывода

1.4. Контролируемый доступ к файлам

1.5. Управление доступом к совместно используемой ЭВМ и ее ресурсам

1.6. Обнаружение ошибок и их обработка

1.7. Учет использования ресурсов

2. Организация эффективного использования ресурсов ЭВМ

2.1. Планирование использования ресурса

2.2. Удовлетворение запросов на ресурсы

2.3. Отслеживание состояния и учет использования ресурса

2.4. Разрешение конфликтов между процессами, претендующими на одни и те же ресурсы

Облегчение процессов эксплуатации аппаратных и программных средств вычислительной системы

3.1. Широкий набор служебных программ (утилит), обеспечивающих резервное копирование, архивацию данных, проверку, очистку, дефрагментацию дисковых устройств и др.

3.2. Средства диагностики и восстановления работоспособности вычислительной системы и операционной системы:

- диагностические программы для выявления ошибок в конфигурации ОС;
- средства восстановления последней работоспособной конфигурации;
- средства восстановления поврежденных и пропавших системных файлов и др.

4. Возможность развития

4.1. Обновление и возникновение новых видов аппаратного обеспечения

4.2. Новые сервисы

4.3. Исправления (обнаружение программных ошибок)

4.4. Новые версии и редакции ОС

Контрольные вопросы

1. Что такое операционная система : (ОС)?

2.Какое место ОС в программном обеспечении вычислительных систем?

3. Опишите состав и функции ОС?

4. Какие модули присутствуют в архитектуры операционных систем?

5. Приведите классификацию операционных систем?

## Архитектура операционных систем

План

1. Концепции построения ОС;
2. Типы ядер ОС;
3. Распределенность ОС.

Особенности архитектуры ОС

При описании операционной системы часто указываются основные концепции, положенные в ее основу.

К базовым концепциям относятся:

- способы построения ядра ОС;
- построение на базе ООПодхода;
- наличие нескольких прикладных сред;
- распределенная организация ОС.

По способам построения ядра ОС подразделяются на

- монолитные (Windows, Linux – можем сами собрать ядро, включив в него модули и драйверы, которые считаем целесообразным включить);
- микроядерные (QNX).

Большинство ОС использует монолитное ядро, которое компонуется как одна программа, работающая в привилегированном режиме и использующая быстрые переходы с одной процедуры на другую, не требующие переключения из привилегированного режима в пользовательский и наоборот.

Альтернативой является построение ОС на базе микроядра, работающего также в привилегированном режиме и выполняющего только минимум функций по управлению аппаратурой, в то время как функции ОС более высокого уровня выполняют специализированные компоненты ОС - серверы, работающие в пользовательском режиме.

Недостаток - ОС работает более медленно, так как часто выполняются переходы между привилегированным режимом и пользовательским.

Достоинство - ОС более гибкая - ее функции можно наращивать, модифицировать или сужать, добавляя, модифицируя или исключая серверы пользовательского режима. Кроме того, серверы хорошо защищены друг от друга, как и любые пользовательские процессы.

Построение ОС на базе объектно-ориентированного подхода дает возможность использовать все его достоинства, хорошо зарекомендовавшие себя на уровне приложений, внутри операционной системы, а именно:

- аккумуляцию удачных решений в форме стандартных объектов, возможность создания новых объектов на базе имеющихся с помощью механизма наследования,
- хорошую защиту данных за счет их инкапсуляции во внутренние структуры объекта, что делает данные недоступными для несанкционированного использования извне,
- структуризованность системы, состоящей из набора хорошо определенных объектов.

Наличие нескольких прикладных сред дает возможность в рамках одной ОС одновременно выполнять приложения, разработанные для нескольких ОС.

Многие современные операционные системы поддерживают одновременно прикладные среды MS-DOS, Windows, UNIX (POSIX), OS/2 или хотя бы некоторого подмножества из этого популярного набора. Концепция множественных прикладных сред наиболее просто реализуется в ОС на базе микроядра.

Распределенная организация операционной системы позволяет упростить работу пользователей и программистов в сетевых средах.

В распределенной ОС реализованы механизмы, которые дают возможность пользователю представлять и воспринимать сеть в виде традиционного однопроцессорного компьютера.

Контрольные вопросы

1. Опишите концепции построения ОС;
2. Назовите и охарактеризуйте типы ядер ОС;
3. Что такое распределенность ОС?

## Микроядерная архитектура (модель «клиент-сервер»)

План

1. Концепция микроядерной архитектуры?
2. Преимущества и недостатки микроядерной архитектуры?
3. Совместимость ОС.

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

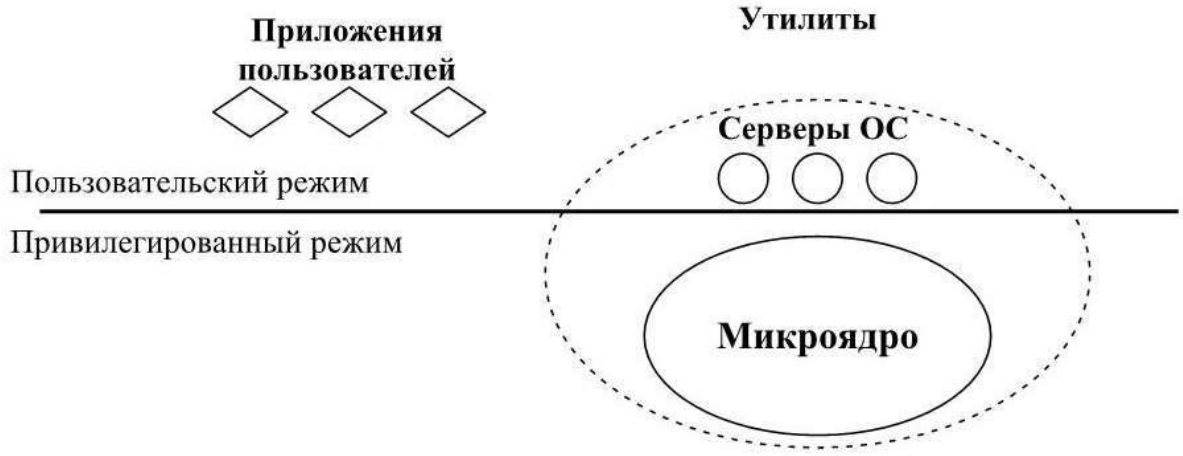



Рисунок 1 – Перенос основного объема функций ядра в пользовательское пространство

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

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

Схематично механизм обращения к функциям операционной системы, оформленным в виде серверов, выглядит следующим образом (Рисунок 8). Клиент, которым может быть либо прикладная программа, либо другой компонент операционной системы, запрашивает выполнение некоторой функции у соответствующего сервера, посылая ему сообщение. Непосредственная передача сообщений между приложениями невозможна, так как их адресные пространства изолированы друг от друга. Микроядро,

выполняющееся в привилегированном режиме, имеет доступ к адресным пространствам каждого из этих приложений и поэтому может работать в качестве посредника. Микроядро сначала передает сообщение, содержащее имя и параметры вызываемой процедуры нужному серверу, затем сервер выполняет запрошенную операцию, после чего ядро возвращает результаты клиенту с помощью другого сообщения. Таким образом, работа микроядерной операционной системы соответствует известной модели клиентсервер, в которой роль транспортных средств выполняет микроядро.

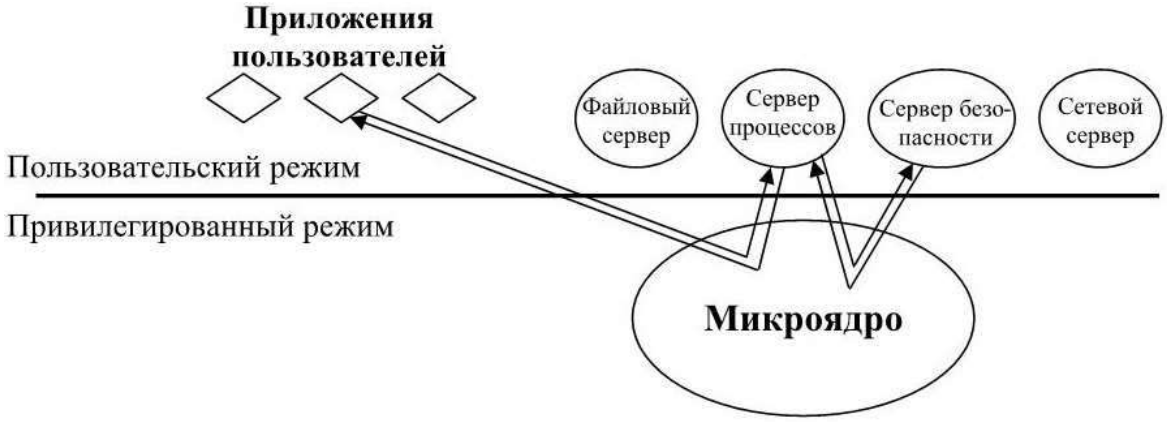



Рисунок 2 – Реализация системного вызова в микроядерной архитектуре

Достоинства микроядерной архитектуры:

1 Переносимость. Высокая степень переносимости обусловлена тем, что весь машинно-зависимый код изолирован в микроядре, поэтому для переноса системы на новый процессор требуется меньше изменений и все они логически сгруппированы вместе.

2Расширяемость присуща микроядерной операционной системе в очень высокой степени.

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

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

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

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


Рисунок 3 – Смена режимов при выполнении системного вызова: в классической архитектуре (а); в микроядерной (б)

Таким образом, операционная система на основе микроядра при прочих равных условиях всегда будет менее производительной, чем система с классическим ядром. Именно по этой причине микроядерный подход не получил такого широкого распространения, которое ему предрекали. Примером микроядерной системы является VM/370, использующаяся в мейнфреймах.

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

Контрольные вопросы

1. Объясните концепцию микроядерной архитектуры?
2. Какие преимущества и недостатки имеет микроядерная архитектура?
3. Как обеспечивается совместимость ОСс разными типами ядер.

## Интерфейс пользователя

План

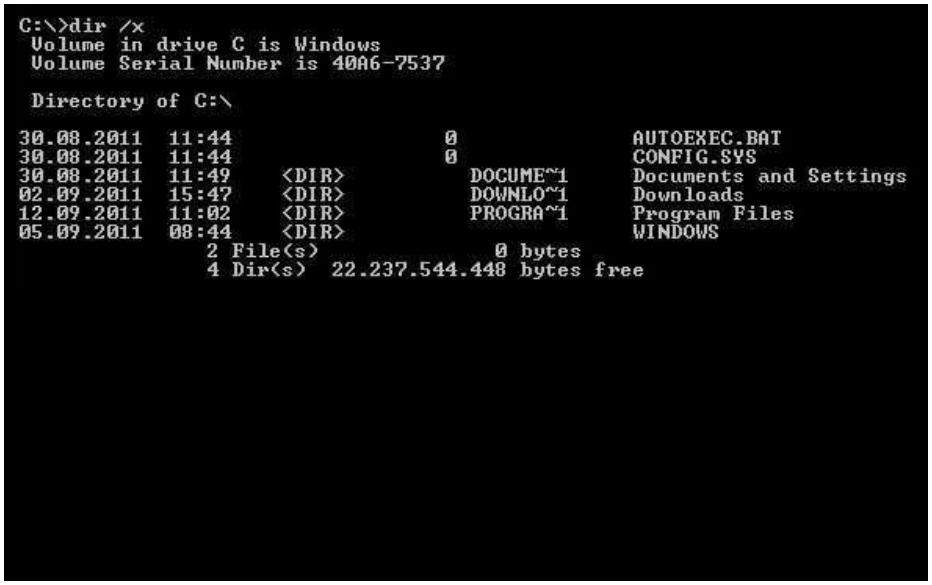
1. Определение интерфейса;
2. Интерфейс командной строки
3. Текстовый интерфейс пользователя;
4. Графический пользовательский интерфейс.

Пользовательский интерфейс

Интерфейс пользователя (user interface или сокращенно UI) – это интерфейс, с помощью которого человек может управлять программным обеспечением или аппаратным оснащением. UI должны быть удобными в использовании, чтобы взаимодействие с ними происходило на максимально интуитивном уровне. Интерфейсы

программного обеспечения также называют графическими пользовательскими интерфейсами (graphical user interface или GUI).

Интерфейс командной строки (Command Line Interface или CLI)

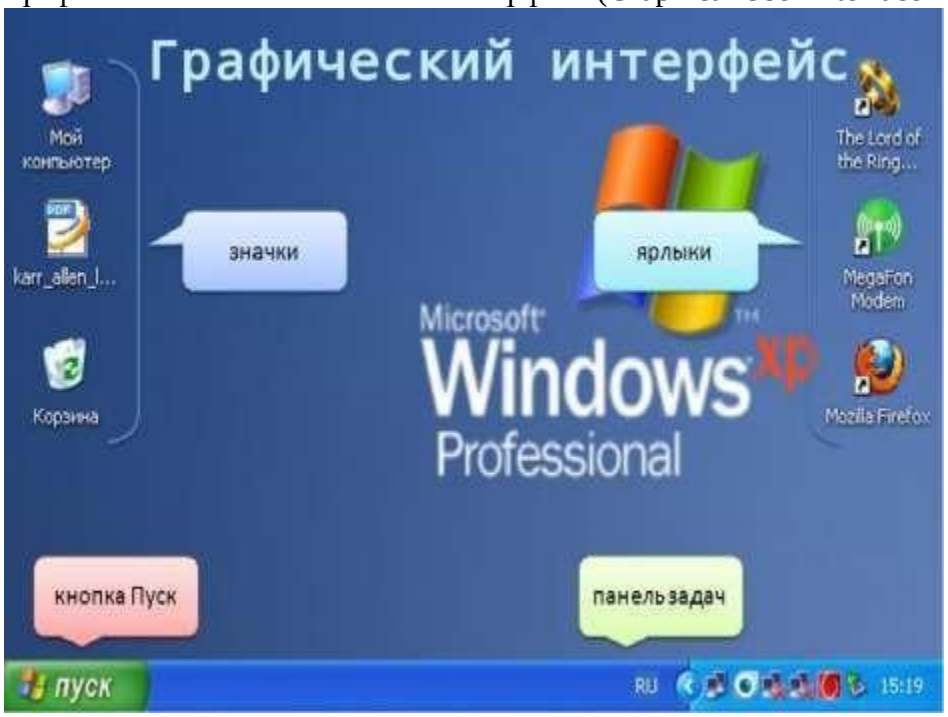



Среди областей применения интерфейса командной строки можно выделить DOS-компьютеры. Взаимодействие происходит с помощью ввода команд. Компьютер обрабатывает эти команды и выводит на экран очередную строку. Данный тип UI давно устарел. Большинство CLI заменены графическими интерфейсами.

Текстовый интерфейс пользователя (Text User Interface или TUI)

Этот тип интерфейса пользователя предназначен для работы с символами. Исполнение происходит в режиме аппаратного текста, однако часто используется и дисплей. В данном случае на каждый источник у программиста имеется 256 символов. Навигация производится клавиатурой, а не мышью. В качестве примера можно привести Norton Commander или Turbo Pascal. Этот интерфейс также используется в загрузчиках ОС и BIOS-программах. Данный тип интерфейса также используется для установки операционных систем.

Графический пользовательский интерфейс (Graphical User Interface или GUI)



Графический пользовательский интерфейс является наиболее популярным UI. Он представляет собой окно, в котором содержатся различные элементы управления. Взаимодействие пользователя с программой при помощи мыши и при помощи клавиатуры.

Также есть возможность использовать кнопки и разделы меню, расположенные внутри самого приложения. Это окно представляет собой нечто вроде шлюза между пользователем и программным обеспечением. В графическом интерфейсе пользователя распространены типичные элементы управления. Они позволяют стандартизировать процесс взаимодействия с различными программами в разных операционных системах.

Реальный мир как модель

При разработке первого графического пользовательского интерфейса за основу были взяты элементы реального мира: мусорная корзина, папка, изображение дискеты в качестве кнопки сохранения. Сегодня многие иконки считаются устаревшими, но все равно используются.

Даже при использовании современных изображений и иконок дизайнеры стараются хотя бы минимально отразить их предназначение. Это позволяет облегчить интуитивное взаимодействие с интерфейсом. Цель GUI заключается в том, что люди могли легко определить предназначение каждой кнопки. Благодаря этому нам не приходится запоминать все команды, как это было в случае с командной строкой.

Контрольные вопросы

1. Что такое пользовательский интерфейс?
2. Опишите особенности интерфейса командной строки
3. Опишите особенности текстового интерфейс пользователя;
4. Опишите особенности графического пользовательский интерфейс.

## Обработка прерываний

План

1. Типы прерываний
2. Принцип действия «регистра прерываний»
4. Принцип действия «вектора прерываний»
4. Принцип действия регистра «слово состояние процессора».

Прерывание - событие в компьютере, при возникновении которого в процессоре происходит предопределенная последовательность действий. Прерывания возникают в нестандартных ситуациях (например, в регистре команд декодируется операция с неизвестным кодом).мы заранее оговариваем какого рода нестандартные ситуации могут произойти. Прерывание – программно-аппаратное средство.

Прерывания бывают:

внутренние - инициируются схемами контроля работы процессора

внешние - события, возникающие в компьютере в результате взаимодействия центрального процессора с внешними устройствами.

Обработка прерываний проходит в два этапа. Первый этап – аппаратный, он выполняется ЦП. Второй этап – программный, он выполняется ОС.

Аппаратный этап обработки прерываний

На аппаратном этапе обработки прерываний процессором производятся следующие действия:

Включение режима блокировки прерываний. В этом режиме все поступающие прерывания либо игнорируются, либо становятся в очередь (это зависит от конкретной архитектуры).

Завершение выполнения текущей команды.

Сохранение актуального состояния (некоторого подмножества регистров) процессора в аппаратный буфер ("малое упрятывание").

Присвоение регистру адреса некоторого заранее определенного значения (адреса обработчика), соответствующего программному этапу обработки прерываний. В зависимости от модели организации прерываний это может быть один и тот же адрес для всех прерываний или свой адрес для каждого типа прерываний.

Программный этап обработки прерываний

На программном этапе сначала определяется тип прерывания.

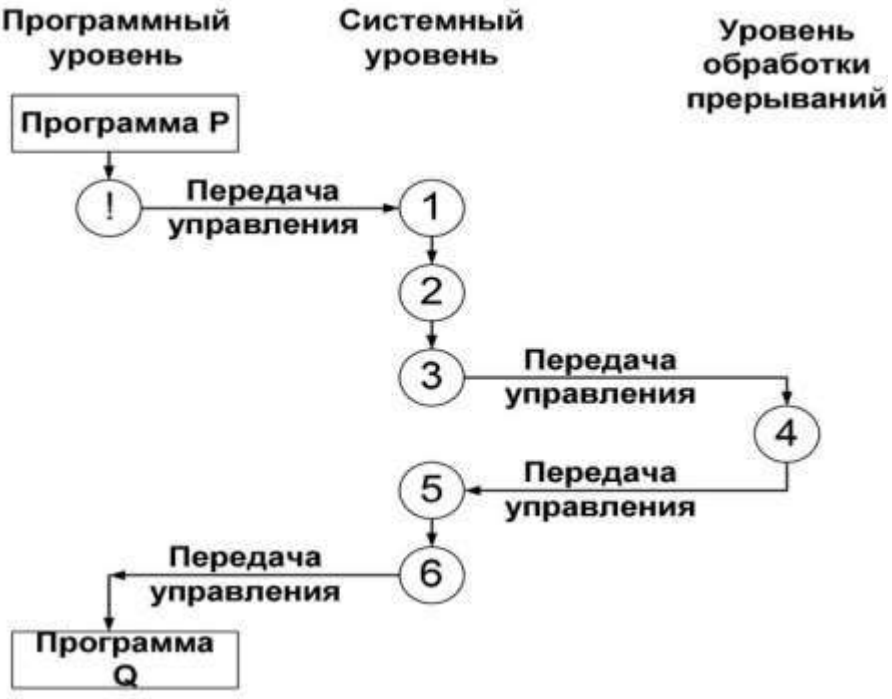
Прерывание может быть «коротким», т.е не требующим больших ресурсов и значительного времени обработки. Пример: прерывание, связанное с таймером. В этом случае происходит обработка и осуществляется выход из прерывания (т.е. восстановление состояние процессора в точке прерывания (за счет аппаратного буфера), возврат в точку прерывания и одновременное снятие блокировки прерывания).

Если прерывание не «короткое», то возможны две ситуации:

Прерывание «фатальное», т.е. такое, после которого продолжение выполнения прерванной программы невозможно. В этом случае происходит снятие блокировки прерываний и ОС завершает выполнение программы, т.е. выполняет те действия, которые освобождают ресурсы.

Прерывание не «фатальное», т.е. прерванная программа еще будет выполняться после его обработки. В этом случае ОС сохраняет все регистры, соответствующие контексту процесса (“полное упрятывание”) в программный буфер, затем снимает блокировку прерываний (с этого места обработка прерывания уже может быть прервана другим прерыванием) и завершает обработку прерывания.

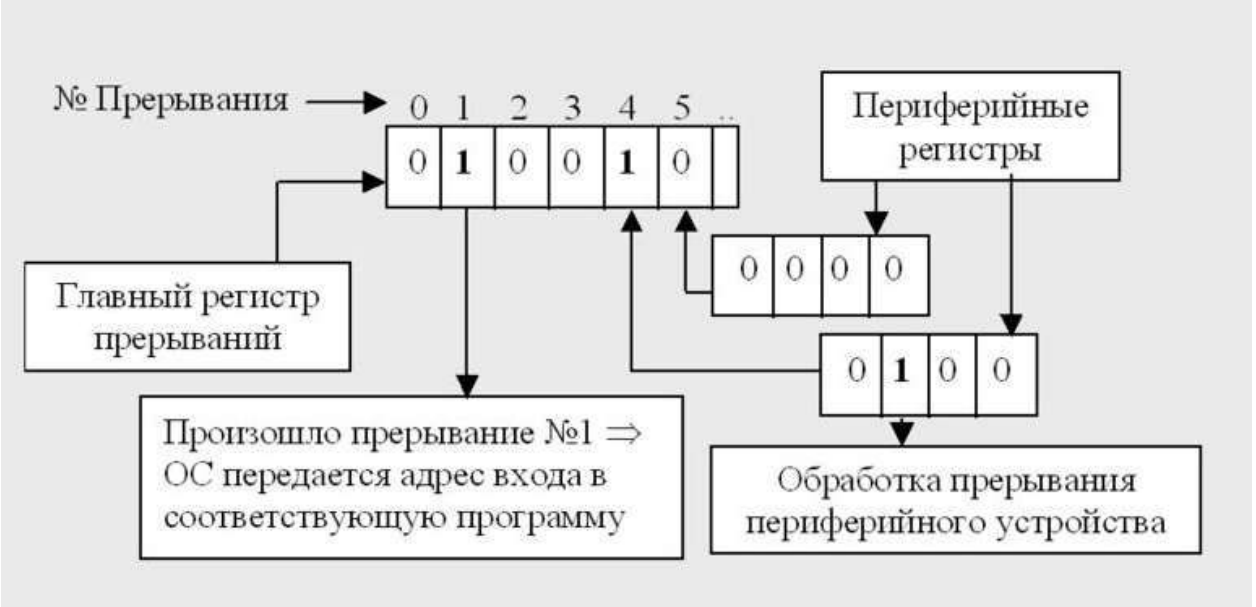
Схема программного этапа обработки прерываний:



Модели организации прерываний

Рассмотрим три модели организации прерываний:

Использование «регистра прерываний»




Каждый разряд этого регистра отвечает за появление того или иного прерывания или группы прерываний (т.е. каким-то регистрам может соответствовать прерывание определенного типа, а в каком-то регистре возможна индикация о том, что есть еще один периферийный регистр прерываний, в котором появилось прерывание). Когда ОС получает управление, то специальными командами, которые доступны только ОС, она может прочесть регистр прерываний и определить причину прерывания, а после этого, в зависимости от причины, передать управление на ту или иную программу обработки прерывания.

Использование вектора прерываний

Здесь процессор предполагает, что в определенном фрагменте ОП размещается вектор прерываний. Это таблица, каждая строка которой соответствует определенному прерыванию, соответственно содержимое строки есть адрес программы-обработчика соответствующего прерывания, также в этой строке может находиться дополнительная информация, например, о том, в какой режим нужно перевести процессор при переходе, приоритет операций и т.д. и т.п. Соответственно, при возникновении прерывания аппаратно управление передается не на одну точку, а уже на точку, которая соответствует конкретному прерыванию, т.е. уже сразу идет попадание на обработчик прерываний.


Использование регистра «слово состояние процессора»


Код прерывания аппаратно помещается в регистр «слово состояние процессора», после этого программа-обработчик прерывания, выбрав этот код, принимает решение о дальнейшей последовательности действий, которые необходимо осуществить для обработки прерывания, стоящего под этим кодом.

Контрольные вопросы

1. Какие бывают прерывания и зачем они используются?
2. Объясните принцип действия «регистра прерываний»
3. Объясните принцип действия «вектора прерываний»
4. Объясните принцип действия «регистра прерываний»
5. Объясните принцип действия регистра «слово состояние процессора».

**Планирование и диспетчеризация процессов**

План

1. Диспетчеризация процессов в ОС;
2. Дисциплина обслуживания *FCFS;*
3. Дисциплина обслуживания *SJR;*
4. Дисциплина обслуживания *RR:*
5. Критерии алгоритмов диспетчеризации.

**Планирование процессов и диспетчеризация задач**

Стратегия планирования определяет, какие процессы планируются на выполнение, чтобы достичь поставленной цели. Существует много стратегий, можно назвать некоторые из них:

a) по возможности заканчивать вычислительные процессы в том же самом порядке, в котором они были начаты;

b) отдавать предпочтение более коротким процессам;

c) предоставлять всем процессам пользователей одинаковые услуги, в том числе и одинаковое время ожидания.

Когда говорят о стратегии обслуживания, всегда имеют в виду понятие процесса, поскольку процесс может состоять из нескольких задач.

Известно большое количество правил (дисциплин) диспетчеризации, в соответствии с которыми формируется список (очередь) готовых к выполнению задач. Различают два больших класса таких дисциплин – *бесприоритетные* и *приоритетные*. При реализации приоритетных дисциплин отдельным задачам предоставляется преимущественное право попасть в состояние исполнения.

Рассмотрим кратко некоторые наиболее часто используемые дисциплины диспетчеризации.

Самой простой в реализации является *дисциплина FCFS* (first come – first served), согласно которой задачи обслуживаются в порядке очереди, т.е. в порядке их появления. Те задачи, которые были заблокированы в процессе работы, после перехода в состояние готовности ставятся в эту очередь перед теми задачами, которые еще не выполнялись, т.е. образуется две очереди – одна из новых задач, а вторая очередь – из ранее выполнявшихся, но попавших в состояние ожидания. Такой подход позволяет реализовать стратегию обслуживания «по возможности заканчивать вычислительные процессы в порядке их появления».

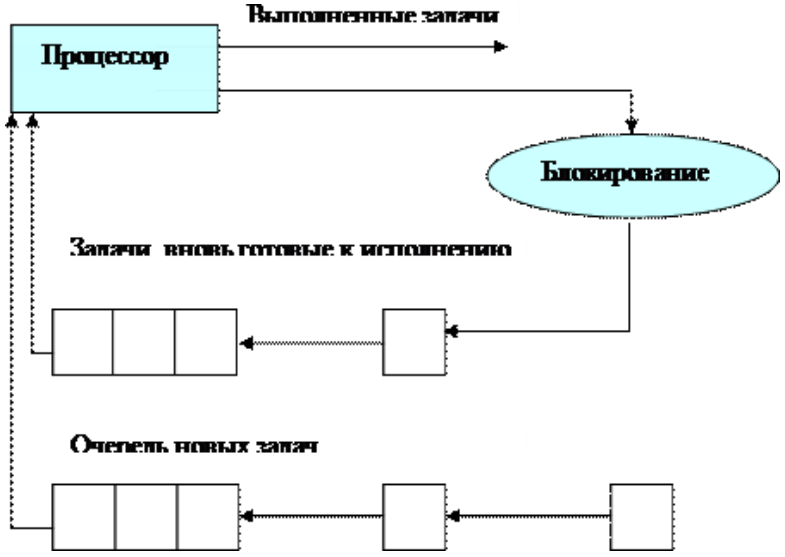



Рис.1. Дисциплина FCFS

К достоинствам этой дисциплины можно отнести простоту реализации и малые расходы системных ресурсов на формирование очереди задач.

Но она приводит к тому, что при увеличении загрузки вычислительной системы растет и среднее время ожидания обслуживания, причем короткие задания вынуждены ожидать столько же, сколько и трудоемкие задания. Избежать этого недостатка позволяют дисциплины SJN и SRT.

Дисциплина *обслуживания SJN* (shortest job next – следующим будет выполняться кратчайшее задание) требует, чтобы для каждого задания была известна оценка в потребностях машинного времени. Для этого были разработаны специальные языковые средства, например, язык JCL (job control language – язык управления заданиями). Пользователи должны были указывать предполагаемое время выполнения, а чтобы они не ловчили, ввели подсчет реальных потребностей. Если обнаруживался обман, то задание ставилось в конец очереди или оплата шла по более высоким тарифам.

Дисциплина обслуживания SJN предполагает, что имеется только одна очередь заданий, готовых к выполнению. И задания, которые в процессе своего выполнения были временно заблокированы, вновь попадают в конец очереди. Это приводит к тому, что задания, которым требуется очень немного времени для завершения, ожидают процессор наравне с длительными работами.

Для устранения этого недостатка была предложена *дисциплина SRT* (shortest remaining time) – следующее задание требует меньше всего времени для своего завершения.

Все эти три дисциплины могут использоваться для пакетных режимов обработки, когда пользователь сдает свое задание, не ожидает реакции системы, а ему нужен только результат вычислений. Для интерактивных вычислений желательно обеспечить приемлемое время реакции системы и равенство в обслуживании, если система является мультитерминальной. Для однопользовательских систем с возможностью мультипрограммной обработки желательно, чтобы программы, с которыми работают непосредственно, имели лучшее время реакции, чем фоновые задания. Для решения подобных проблем используется дисциплина обслуживания RR(round robin – круговая, карусельная) и приоритетные методы обслуживания.

*Дисциплина RR* предполагает, что каждая задача получает процессорное время порциями (квантами времени q). После окончания кванта времени q задача снимается с процессора, и он передается следующей задаче. Снятая задача ставится в конец очереди задач, готовых к исполнению. Для оптимальной работы системы необходимо правильно выбрать закон, по которому кванты времени выделяются задачам.

Величина кванта времени q выбирается как компромисс между приемлемым временем реакции системы на запросы пользователей и накладными расходами на частую смену контекста задачи (надо запоминать много информации для прерываний).

Дисциплина RR – это одна из самых распространенных дисциплин диспетчеризации. В своей простейшей реализации на предполагает, что все задачи имеют одинаковый приоритет. Для введения приоритетного обслуживания вводят несколько очередей. Процессорное время будет предоставляться в первую очередь тем задачам, которые стоят в самой привилегированной очереди. Если она пустая, то диспетчер задач начнет просматривать остальные очереди. По такому алгоритму действует диспетчер задач в Windows NT.

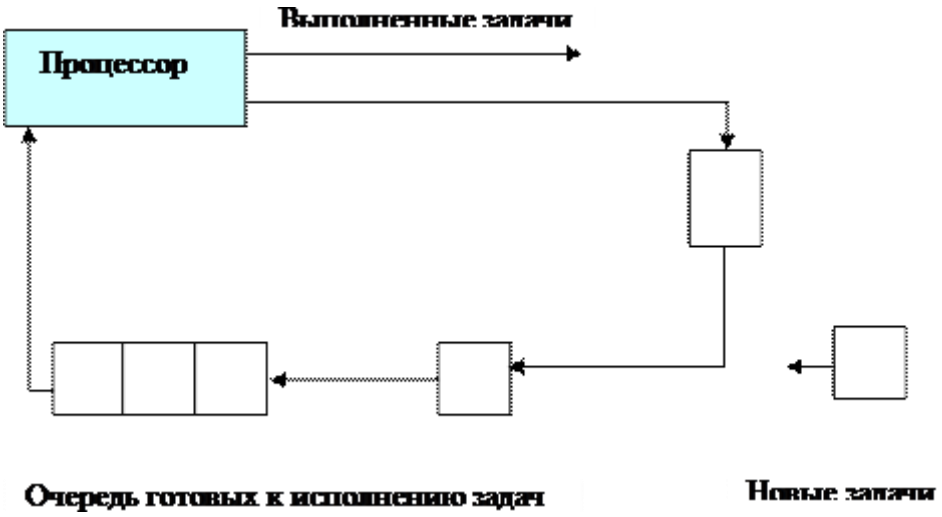



Рис.2. Дисциплина RR

Диспетчеризация без перераспределения процессорного времени, т.е. *не вытесняющая многозадачность* – это такой способ диспетчеризации процессов, при котором активный процесс выполняется до тех пор, пока он сам не отдаст управление диспетчеру задач. Дисциплины FCFS, SJN, SRT относятся к не вытесняющим.

Диспетчеризация с перераспределением процессорного времени между задачами, т.е. *вытесняющая многозадачность* – это такой способ, при котором решение о переключении с одного процесса на другой принимается диспетчером задач, а не самой активной задачей. Дисциплина RR относится к вытесняющим.

Для сравнения алгоритмов диспетчеризации обычно используют следующие критерии:

a) использование (загрузка) CPU – центрального процессора;

b) пропускная способность CPU – количество процессов, выполняющихся в единицу времени;

c) время оборота – интервал от момента появления процесса во входной очереди до момента его завершения;

d) время ожидания – суммарное время нахождения в очереди готовых процессов;

e) время отклика – время, прошедшее от момента попадания во входнуюъ очередь до момента первого обращения к терминалу. Является важным для интерактивных программ.

Правильное планирование сильно влияет на производительность всей системы.

Контрольные вопросы

1. Что такое диспетчеризация процессов в ОС?
2. Опишите дисциплину обслуживания *FCFS;*
3. Опишите дисциплину обслуживания *SJR;*
4. Опишите дисциплину обслуживания *RR:*
5. Какие критерии алгоритмов диспетчеризации существуют.

**Типы загрузки процессов**

План

1. Алгоритмы планирования процессов;
2. Принцип FIFO и LIFO;
3. Алгоритмы, основанные на квантовании времени
4. Алгоритмы, основанные на приоритетах.

**Создание процесса**:

- ОС создает контекст и дескриптор процесса.
- Загрузка кодового сегмента в оперативную память.
- Дескриптор помещается в очередь процессов, находящихся в готовности.

С этого момента можно считать, что процесс стартовал.

Разберём подробно вопрос **создания процесса**. Когда запускается приложение, Операционная Система создаёт две информационные структуры: контекст и дескриптор (идентификатор и т.д.). Кодовый элемент грузится в оперативную память (информацию о нём заносится в контекст). Затем дескриптор процесса включается в очередь процессов; далее планировщик решает, что процесс должен быть выполнен, а дескриптор процесса находится в очереди готовых процессов. Прежде прервать выполнение процесса, ОС вначале сохраняет его контекст, чтобы в последствие использовать эту инструкцию для последовательного возобновления. Затем контекст обновляется (переключение контекста) и активный контекст получает информацию о новом процессе. Далее процессу выделяются ресурсы, процессорное время и т.д. При удалении процесса: уничтожается дескриптор и удаляется контекст. Новый процесс может получить дескриптор с тем же номером.

**Алгоритмы планирования процессов**

В зависимости от того, какие критерии накладываются, алгоритмы планирования могут основываться на:

- Квантовании времени.
- Приоритетах.

**Алгоритмы, основанные на квантовании времени**

Алгоритмы, основанные на квантовании времени – любому процессу на выполнение отводится определенный квант времени (несколько милисекунд). Переключение активного процесса происходит если:

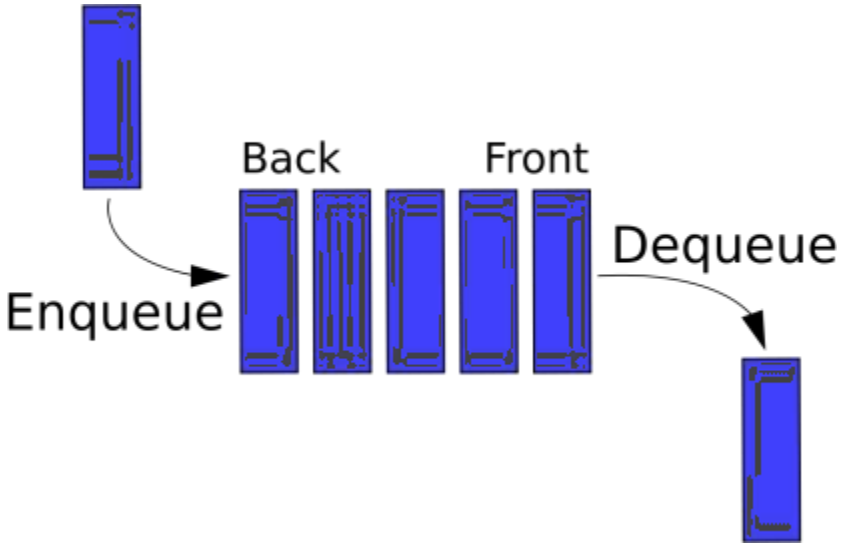
- Истек срок времени, выделенного на выполнение процесса.
- Процесс завершился.
- Процесс перешел в состояние ожидания.

По истечении выделенного времени планировщик ставит другой процесс. Если до истечения времени процесс находится в режиме ожидания, запускается другой процесс. Кванты времени выделенные процессами, могут быть для разных процессов одинаковыми или различными. Кванты, выделяемые одному процессу могут быть фиксированной величины, а могут и изменяться в разные периоды жизни процесса. Некоторые из процессов используют полученные кванты времени не полностью из-за необходимости выполнить операции ввода-вывода. Тогда возникает ситуация, когда процессы с интенсивными обращениями к вводу-выводу используют только небольшую часть выделенного им процессорного времени. В качестве компенсации за не полностью

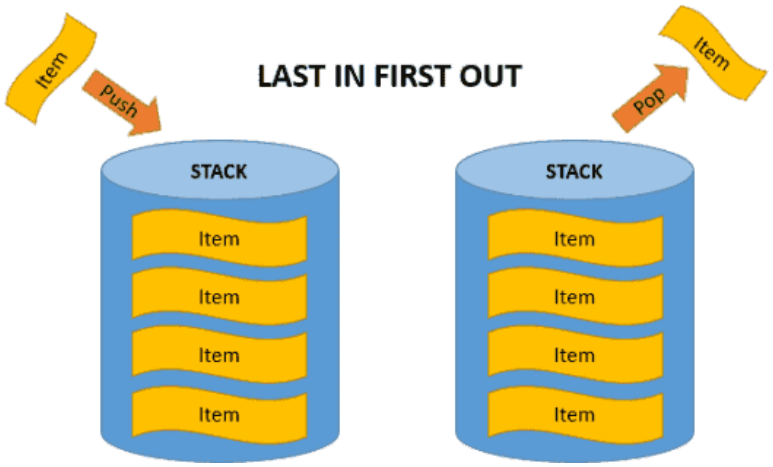
использованные кванты, процессы получают привилегии при последующем обслуживании (создают две очереди готовых процессов, прежде всего просматривается вторая очередь; если она пуста, квант выделяется процессу из первой очереди)

**Выбор новых процессов может быть построен по принципам**:

- FIFO (очередь).




- LIFO (стек).




**Алгоритмы, основанные на приоритетах**

*Приоритет* – число, характеризующее степень привилегированности процесса (обычно выражается числом). В каждой ОС это число трактуется по своему, оно может быть фиксированным или изменяться. В случае если изменяется, то называется динамическим (начальное значение устанавливает администратор) в отличие от неизменяемых, фиксированных приоритетов. Чем выше приоритет, тем выше привелегия, тем меньше времени проводит поток в очереди.

Существует 2 разновидности таких алгоритмов:

- Использующие относительные приоритеты.
- Использующие абсолютные приоритеты.

**Алгоритмы планирования с относительными приоритетами** – активный процесс выполняется пока не завершится или не перейдет в состояние ожидания.

**Алгоритмы планирования с абсолютными приоритетами** – смена процесса происходит в тот момент, когда в системе появляется процесс, приоритет которого выше приоритета выполняемого процесса.

Реально используются смешанные схемы планирования.

Контрольные вопросы

1. Какие алгоритмы планирования процессов бывают?
2. Опишите особенности принципа FIFO и LIFO;
3. Опишите алгоритмы, основанные на квантовании времени
4. Опишите алгоритмы, основанные на приоритетах.

## Мультипрограммирование в системе пакетной обработки, разделения времени, реального времени

План

1. Определение мультипрограммирования;
2. Типы мультипрограммирования;
3. Особенности пакетной мультипрограммной обработки.

**Мультипрограммирование**

Мультипрограммирование, или многозадачность (multitasking), — это способ организации вычислительного процесса, при котором на одном процессоре попеременно выполняются сразу несколько программ. Эти программы совместно используют не только процессор, но и другие ресурсы компьютера: оперативную и внешнюю память, устройства ввода-вывода, данные. Мультипрограммирование призвано повысить эффективность использования вычислительной системы, однако эффективность может пониматься по-разному. Наиболее характерными критериями эффективности вычислительных систем являются:

- пропускная способность — количество задач, выполняемых вычислительной системой в единицу времени;
- удобство работы пользователей, заключающееся, в частности, в том, что они имеют возможность интерактивно работать одновременно с несколькими приложениями на одной машине;
- реактивность системы — способность системы выдерживать заранее заданные (возможно, очень короткие) интервалы времени между запуском программы и получением результата.

В зависимости от выбранного критерия эффективности ОС делятся на системы пакетной обработки, системы разделения времени и системы реального времени. Каждый тип ОС имеет специфические внутренние механизмы и особые области применения. Некоторые операционные системы могут поддерживать одновременно несколько режимов, например часть задач может выполняться в режиме пакетной обработки, а часть — в режиме реального времени или в режиме разделения времени.

При использовании мультипрограммирования для повышения пропускной способности компьютера главной целью является минимизация простоев всех устройств компьютера, и прежде всего центрального процессора., Такие простои могут возникать из-за приостановки задачи по ее внутренним причинам, связанным, например, с ожиданием ввода данных для обработки. Данные могут храниться на диске или же поступать от пользователя, работающего за терминалом, а также от измерительной аппаратуры,

установленной на внешних технических объектах. При возникновении такого рода блокировки выполняемой задачи естественным решением, ведущим к повышению эффективности использования процессора, является переключение процессора на выполнение другой задачи, у которой есть данные для обработки. Такая концепция мультипрограммирования положена в основу так называемых пакетных систем.

Системы пакетной обработки предназначались для решения задач в основном вычислительного характера, не требующих быстрого получения результатов. Главной целью и критерием эффективности систем пакетной обработки является максимальная пропускная способность, то есть решение максимального числа задач в единицу времени.

Для достижения этой цели в системах пакетной обработки используется следующая схема функционирования: в начале работы формируется пакет заданий, каждое задание содержит требование к системным ресурсам; из этого пакета заданий формируется мультипрограммная смесь, то есть множество одновременно выполняемых задач. Для одновременного выполнения выбираются задачи, предъявляющие разные требования к ресурсам, так, чтобы обеспечивалась сбалансированная загрузка всех устройств вычислительной машины. Например, в мультипрограммной смеси желательно одновременное присутствие вычислительных задач и задач с интенсивным вводом-выводом. Таким образом, выбор нового задания из пакета заданий зависит от внутренней ситуации, складывающейся в системе, то есть выбирается «выгодное» задание. Следовательно, в вычислительных системах, работающих под управлением пакетных ОС, невозможно гарантировать выполнение того или иного задания в течение определенного периода времени.

Рассмотрим более детально совмещение во времени операций ввода-вывода и вычислений.

Такое совмещение может достигаться разными способами. Один из них характерен для компьютеров, имеющих специализированный процессор ввода-вывода. В компьютерах класса мэйнфреймов такие процессоры называют каналами. Обычно канал имеет систему команд, отличающуюся от системы команд центрального процессора. Эти команды специально предназначены для управления внешними устройствами, например «проверить состояние устройства», «установить магнитную головку», «установить начало листа», «напечатать строку». Канальные программы могут храниться в той же оперативной памяти, что и программы центрального процессора. В системе команд центрального процессора предусматривается специальная инструкция, с помощью которой каналу передаются параметры и указания на то, какую программу ввода-вывода он должен выполнить. Начиная с этого момента центральный процессор и канал могут работать параллельно (рис. 4.1, а).

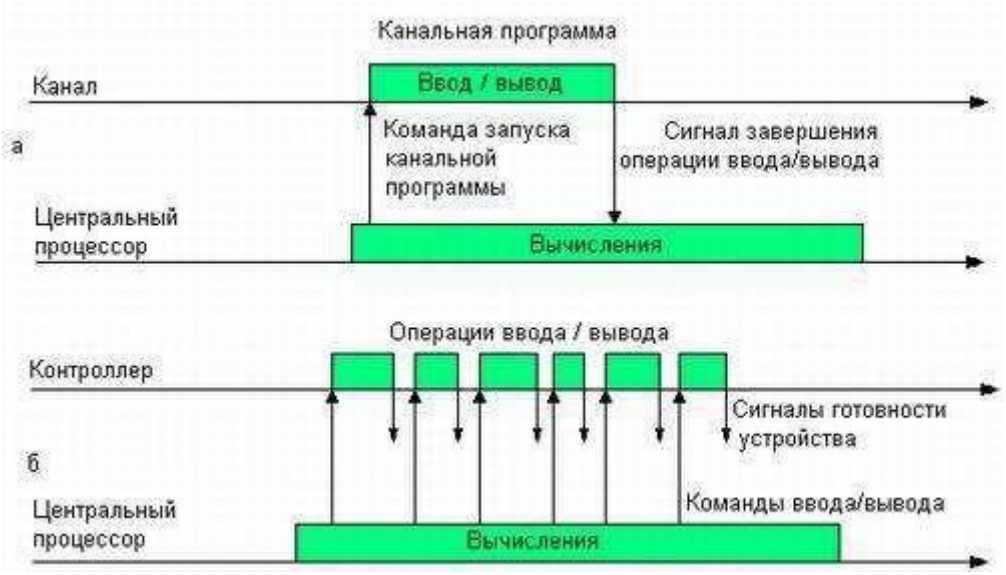



Рис.1. Параллельное выполнение вычислений и операций ввода-вывода

Другой способ совмещения вычислений с операциями ввода-вывода реализуется в компьютерах, в которых внешние устройства управляются не процессором ввода-вывода, а контроллерами. Каждое внешнее устройство (или группа внешних устройств одного типа) имеет свой собственный контроллер, который автономно отрабатывает команды, поступающие от центрального процессора. При этом контроллер и центральный процессор работают асинхронно. Поскольку многие внешние устройства включают электромеханические узлы, контроллер выполняет свои команды управления устройствами существенно медленнее, чем центральный процессор — свои. Это обстоятельство используется для организации параллельного выполнения вычислений и операций ввода-вывода: в промежутке между передачей команд, контроллеру центральный процессор может выполнять вычисления (рис. 4.1, б). Контроллер может сообщить центральному процессору о том, что он готов принять следующую команду, сигналом прерывания либо центральный процессор узнает об этом, периодически опрашивая состояние контроллера.

Максимальный эффект ускорения достигается при наиболее полном перекрытии вычислений и ввода-вывода. Рассмотрим случай, когда процессор выполняет только одну задачу. В этой ситуации степень ускорения зависит от природы данной задачи и от того, насколько тщательно был выявлен возможный параллелизм при ее программировании. В задачах, в которых преобладают либо вычисления, либо ввод-вывод, ускорение почти отсутствует. Параллелизм в рамках одной задачи невозможен также, когда для продолжения вычислений необходимо полное завершение операции ввода-вывода, например, когда дальнейшие вычисления зависят от вводимых данных. В таких случаях неизбежны простои центрального процессора или канала.

Если же в системе выполняются одновременно несколько задач, появляется возможность совмещения вычислений одной задачи с вводом-выводом другой. Пока одна задача ожидает какого-либо события (заметим, что таким событием в мультипрограммной системе может быть не только завершение ввода-вывода, но и, например, наступление определенного момента времени, разблокирование файла или загрузка с диска недостающей страницы программы), процессор не простаивает, как это происходит при последовательном выполнении программ, а выполняет другую задачу.

Общее время выполнения смеси задач часто оказывается меньше, чем их суммарное время последовательного выполнения (рис. 4.2, а). Однако выполнение отдельной задачи в мультипрограммном режиме может занять больше времени, чем при монопольном выделении процессора этой задаче. Действительно, при совместном использовании процессора в системе могут возникать ситуации, когда задача готова выполняться, но процессор занят выполнением другой задачи. В таких случаях задача, завершившая ввод-вывод, готова выполняться, но вынуждена ждать освобождения процессора, и это удлиняет срок ее выполнения. Так, из рис. 4.2 видно, что в однопрограммном режиме задача А выполняется за 6 единиц времени, а в мультипрограммном — за 7. Задача В также вместо 5 единиц времени выполняется за 6. Но зато время выполнения обеих задач в мультипрограммном режиме составляет всего 8 единиц, что на 3 единицы меньше, чем при последовательном выполнении.

В системах пакетной обработки переключение процессора с выполнения одной задачи на выполнение другой происходит по инициативе самой активной задачи, например, когда она отказывается от процессора из-за необходимости выполнить операцию ввода-вывода. Поэтому существует высокая вероятность того, что одна задача может надолго занять процессор и выполнение интерактивных задач станет невозможным. Взаимодействие пользователя с вычислительной машиной, на которой установлена система пакетной обработки, сводится к тому, что он приносит задание, отдает его диспетчеру-оператору, а в конце дня после выполнения всего пакета заданий получает результат. Очевидно, что такой порядок повышает эффективность функционирования аппаратуры, но снижает эффективность работы пользователя.

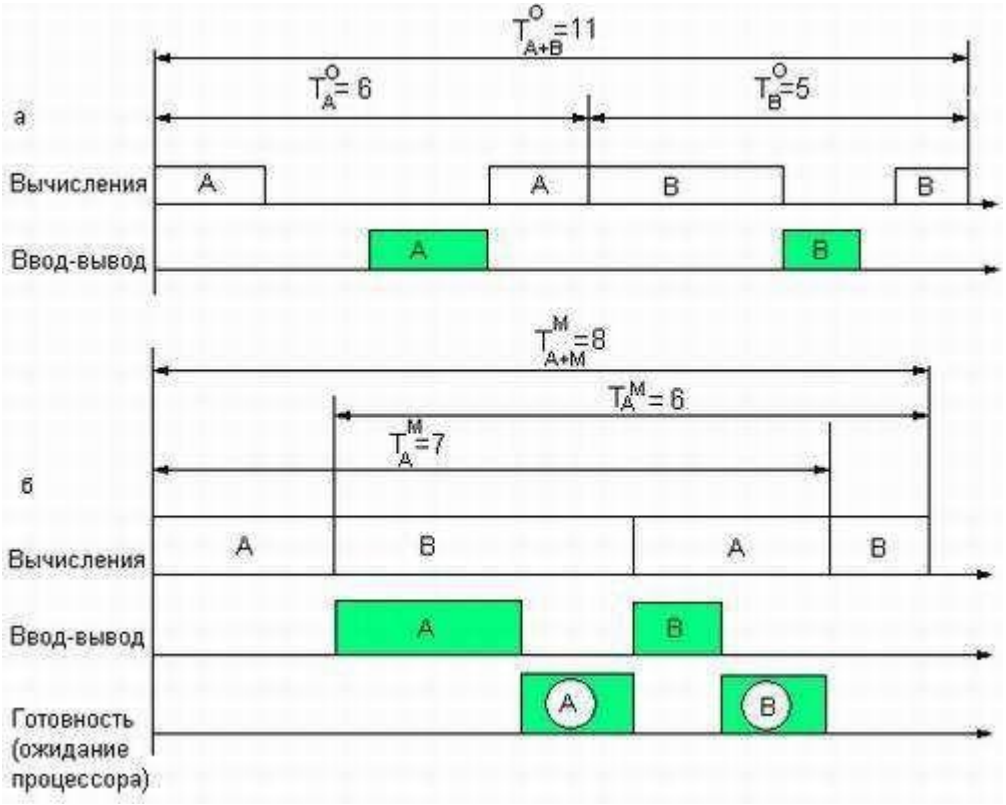



Рис. 2. Время выполнения двух задач: в однопрограммной системе (а), в мультипрограммной системе (б)

1. Дайте определение мультипрограммирования;
2. Какие типы мультипрограммирования вы знаете;

3. Назовите особенности пакетной мультипрограммной обработки.

Контрольные вопросы

## Мультипроцессорная обработка

План

1. Симметричная обработка;
2. Ассиметричная обработка;
3. Параллельная обработка;
4. Конвейерная обработка.

*Мультипроцессорная обработка* — это способ организации вычислительного процесса в системах с несколькими процессорами. В отличие от организации мультипрограммирования на одном процессоре мультипроцессорная обработка предполагает действительно одновременное выполнение нескольких процессов. Это приводит к усложнению всех алгоритмов ОС.

*Симметричная архитектура* предполагает однородность всех процессоров и единообразное их включение в общую схему. Традиционно все процессоры при этом разделяют одну память и как следствие находятся в одном корпусе.

При *асимметричной архитектуре* процессоры могут отличаться своими техническими характеристиками и функциональной ролью. Требование единого корпуса отсутствует. Система может состоять из нескольких корпусов (в каждом может быть один или несколько процессоров). Такие устройства называются *кластерами.*

Для симметричной архитектуры вычислительный процесс может строиться симметричным образом (все процессоры равноправны) или асимметричным (процессоры различаются функционально). Дтя асимметричной архитектуры возможен только асимметричный способ ор-

В мультипроцессорных компьютерах имеется несколько процессоров, каждый из которых может относительно независимо от остальных выполнять свою программу. В мультипроцессоре существует общая для всех процессоров операционная система, которая оперативно распределяет вычислительную нагрузку между процессорами. Взаимодействие между отдельными процессорами организуется наиболее простым способом — через общую оперативную память.

Основное достоинство мультипроцессора — высокая производительность, которая достигается за счет параллельной или конвейерной работы нескольких процессоров. Так как при наличии общей памяти взаимодействие процессоров происходит очень быстро, мультипроцессоры могут эффективно выполнять даже приложения с высокой

Еще одним важным свойством мультипроцессорных систем является отказоустойчивость, т. е. способность к продолжению работы при отказах некоторых элементов, например процессоров или блоков памяти. При этом производительность, естественно, снижается, но не до нуля, как в обычных системах, в которых отсутствует избыточность.

*Параллельная обработка.* Необходимость параллельной обработки может возникнуть, когда требуется уменьшить время решения данной задачи, увеличить пропускную способность, улучшить использование

Для распараллеливания необходимо соответствующим образом организовать вычисления. Сюда входит следующее:

- составление параллельных программ, т. е. отображение в явной форме параллельной обработки с помощью надлежащих конструкций языка, ориентированного на параллельные вычислений;

- автоматическое обнаружение параллелизма. Последовательная программа автоматически анализируется, и в результате может быть выявлена явная или скрытая параллельная обработка. Последняя должна быть преобразована в явную.

Рассмотрим граф, описывающий последовательность процессов большой программы, представленный на рис. 3. Видно, что выполнение процесса $P_1$ не может начаться до завершения процессов $P_2$ и $P_3$ и, в свою очередь, выполнение процессов $P_2$ и $P_i$ не может начаться до завершения процесса $P_y$ В данном случае для выполнения программы достаточно трех процессоров.

Ускорение обработки на мультипроцессоре определяется отношением времени однопроцессорной обработки к времени многопроцессорной обработки:

$$U = \frac{T_1}{N_M}.$$

Конвейерная обработка улучшает использование аппаратных ресурсов для заданного набора процессов, каждый из которых применяет эти ресурсы заранее предусмотренным способом. Хорошим примером конвейерной организации является сборочный транспортер на производстве, на котором изделие последовательно проходит все стадии вплоть до готового продукта. Преимущество этого способа состоит в том, что каждое изделие вдоль своего пути использует одни и те же ресурсы, и как только некоторый ресурс освобождается данным изделием, он сразу же может быть использован следующим изделием, не

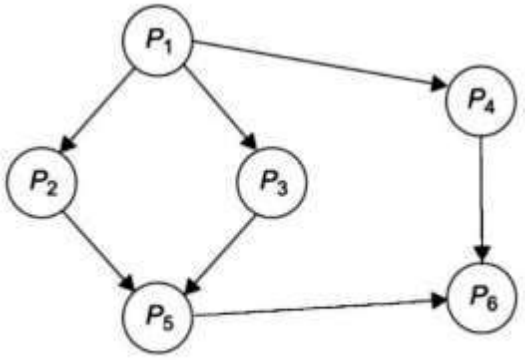



**Рис. 3.** Граф выполнения большой программы

ожидая, пока предыдущее изделие достигнет конца сборочной линии. Если транспортер несет аналогичные, но не тождественные изделия, то это последовательный конвейер; если же все изделия одинаковы, это векторный конвейер.

**Последовательные конвейеры.** На рис. 4, *а* показано устройство обработки команд, в котором имеется четыре ступени: выборка команды из памяти, декодирование, определение адреса и выборка операнда, исполнение.

Ускорение обработки в данном устройстве измеряется отношением времени *Ts*, необходимого для последовательного выполнения *L* заданий (т. е. выполнения *L* циклов на одной обрабатывающей ступени), ко времени *Tp* выполнения той же обработки на конвейере.

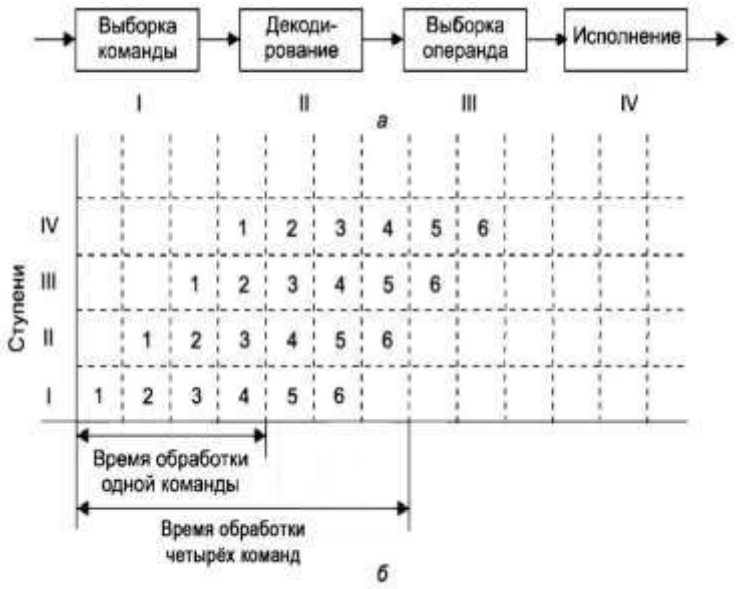



**Рис. 4.** Четырехступенное устройство обработки команд: *а* — ступени конвейера; *б* — временная диаграмма работы

**Классификация архитектур многопроцессорных систем.** Мультипроцессоры, ориентированные на достижение сверхбольших скоростей работы, могут содержать по нескольку сравнительно простых процессоров с упрощенными блоками управления. Удачной для различных мультипроцессоров является классификация Флина, которая строится по признаку одинарности или множественности потоков команд и данных

*Структура* ОКОД (один поток команд, один поток данных) — *однопроцессорная ЭВМ (рис. 5).*

Структура *ОКМД (один поток команд, много потоков данных)* — матричная многопроцессорная структура. Система содержит некоторое число одинаковых сравнительно простых быстродействующих процессоров, соединенных друг с другом и с памятью данных регулярным образом так, что образуется сетка (матрица), в узлах которой размещаются процессоры (рис. 6). Возникает сложная задача распараллеливания алгоритмов решаемых задач для обеспечения загрузки процессоров. В ряде случаев эти вопросы лучше решаются в конвейерной системе.

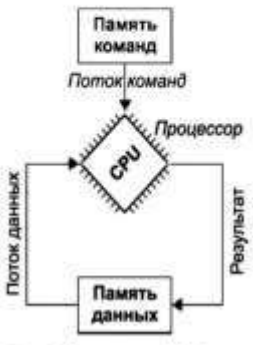



**Рис. 5.** Структура ОКОД

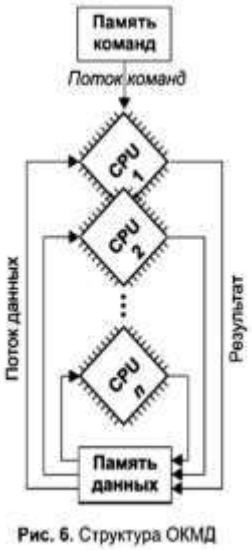



**Рис. 6.** Структура ОКМД

Структура *МКОД (много потоков команд, один поток данных)* — конвейерная многопроцессорная структура (рис. 7). Система имеет регу-

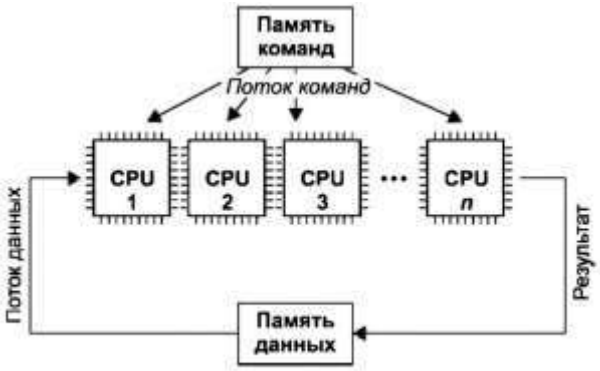



**Рис. 7.** Структура МКОД

лярную структуру в виде цепочки последовательно соединенных процессоров, так что информация на выходе одного процессора является входной информацией для следующего в конвейерной цепочке.

+Процессоры образуют конвейер, на вход которого *одинарный поток данных* доставляет операнды из памяти. Каждый процессор обрабатывает соответствующую часть задачи, передавая результаты соответствующему процессору, который использует их в качестве исходных данных. Таким образом, решение задач для некоторых исходных данных развертывается последовательно в конвейерной цепочке. Это обеспечивает подведение к каждому процессору своего потока команд, т. е. имеется множественный поток команд. Структура *МКМД (много потоков команд, много потоков данных*) представлена на рис. 8.

Существует несколько типов МКМД: мультипроцессорные системы, системы с мультиобработкой, многомашинные системы, компьютерные сети.

Контрольные вопросы

1. Что такое симметричная обработка?
2. Что такое ассиметричная обработка?
3. Что такое параллельная обработка?
4. Что такое конвейерная обработка?
5. Какие типы очередей бывают при мультипроцессорной обработке?

**Методы синхронизации: взаимное исключение, блокирующие переменные**

План

1. Цели и средства синхронизации.
2. Механизмы синхронизации.

*1.Цели и средства синхронизации*

Существует достаточно обширный класс средств операционной системы, с помощью которых обеспечивается взаимная синхронизация процессов и потоков. Потребность в синхронизации потоков возникает только в мультипрограммной операционной системе и связана с совместным использованием аппаратных и информационных ресурсов вычислительной системы. Синхронизация необходима для исключения гонок и тупиков при обмене данными между потоками, разделении данных, при доступе к процессору и устройствам ввода-вывода.

Во многих операционных системах эти средства называются средствами межпроцессного взаимодействия — Inter Process Communications (IPC), что отражает историческую первичность понятия «процесс» по отношению к понятию «поток». Обычно к средствам IPC относят не только средства межпроцессной синхронизации, но и средства межпроцессного обмена данными.

Выполнение потока в мультипрограммной среде всегда имеет асинхронный характер. Очень сложно с полной определенностью сказать, на каком этапе выполнения будет находиться процесс в определенный момент времени. Даже в однопрограммном режиме не всегда можно точно оценить время выполнения задачи. Это время во многих случаях существенно зависит от значения исходных данных, которые влияют на количество циклов, направления разветвления программы, время выполнения операций ввода-вывода и т. п. Так как исходные данные в разные моменты запуска задачи могут быть разными, то и время выполнения отдельных этапов и задачи в целом является весьма неопределенной величиной.

Еще более неопределенным является время выполнения программы в мультипрограммной системе. Моменты прерывания потоков, время нахождения их в очередях к разделяемым ресурсам, порядок выбора потоков для выполнения — все эти события являются результатом стечения многих обстоятельств и могут быть интерпретированы как случайные. В лучшем случае можно оценить вероятностные характеристики вычислительного процесса, например вероятность его завершения за данный период времени.

Таким образом, потоки в общем случае (когда программист не предпринял специальных мер по их синхронизации) протекают независимо, асинхронно друг другу. Это справедливо как по отношению к потокам одного процесса, выполняющим общий программный код, так и по отношению к потокам разных процессов, каждый из которых выполняет собственную программу.

Любое взаимодействие процессов или потоков связано с их *синхронизацией,* которая заключается в согласовании их скоростей путем приостановки потока до наступления некоторого события и последующей его активизации при наступлении этого события. Синхронизация лежит в основе любого взаимодействия потоков, связано ли это взаимодействие с разделением ресурсов или с обменом данными. Например, поток-получатель должен обращаться за данными только после того, как они помещены в буфер потоком-отправителем. Если же поток-получатель обратился к данным до момента их поступления в буфер, то он должен быть приостановлен.

При совместном использовании аппаратных ресурсов синхронизация также совершенно необходима. Когда, например, активному потоку требуется доступ к последовательному порту, а с этим портом в монопольном режиме работает другой поток, находящийся в данный момент в состоянии ожидания, то ОС приостанавливает активный поток и не активизирует его до тех пор, пока нужный ему порт не освободится. Часто нужна также синхронизация с событиями, внешними по отношению к вычислительной системе, например реакции на нажатие комбинации клавиш Ctrl+C.

Ежесекундно в системе происходят сотни событий, связанных с распределением и освобождением ресурсов, и ОС должна иметь надежные и производительные средства, которые бы позволяли ей синхронизировать потоки с происходящими в системе событиями.

Для синхронизации потоков прикладных программ программист может использовать как собственные средства и приемы синхронизации, так и средства операционной системы. Например, два потока одного прикладного процесса могут координировать свою работу с помощью доступной для них обоих глобальной логической переменной, которая устанавливается в единицу при осуществлении некоторого события, например выработки одним потоком данных, нужных для продолжения работы другого. Однако во многих случаях более эффективными или даже единственно возможными являются средства синхронизации, предоставляемые операционной системой в форме системных вызовов. Так, потоки, принадлежащие разным процессам, не имеют возможности вмешиваться каким-либо образом в работу друг друга. Без посредничества операционной системы они не могут приостановить друг друга или оповестить о произошедшем событии. Средства синхронизации используются операционной системой не только для синхронизации прикладных процессов, но и для ее внутренних нужд.

Обычно разработчики операционных систем предоставляют в распоряжение прикладных и системных программистов широкий спектр средств синхронизации. Эти средства могут образовывать иерархию, когда на основе более простых средств строятся более сложные, а также быть функционально специализированными, например средства для синхронизации потоков одного процесса, средства для синхронизации потоков разных процессов при обмене данными и т. д. Часто функциональные возможности разных системных вызовов синхронизации перекрываются, так что для решения одной задачи программист может воспользоваться несколькими вызовами в зависимости от своих личных предпочтений.

**Необходимость синхронизации и гонки**

Пренебрежение вопросами синхронизации в многопоточной системе может привести к неправильному решению задачи или даже к краху системы. Рассмотрим, например (рис. 4.16), задачу ведения базы данных клиентов некоторого предприятия. Каждому клиенту отводится отдельная запись в базе данных, в которой среди прочих полей имеются поля Заказ и Оплата. Программа, ведущая базу данных, оформлена как единый процесс, имеющий несколько потоков, в том числе поток А, который заносит в базу данных информацию о заказах, поступивших от клиентов, и поток В, который фиксирует в базе данных сведения об оплате клиентами выставленных счетов. Оба эти потока совместно работают над общим файлом базы данных, используя однотипные алгоритмы, включающие три шага.

1. Считать из файла базы данных в буфер запись о клиенте с заданным идентификатором.

2. Внести новое значение в поле Заказ (для потока А) или Оплата (для потока В).

3. Вернуть модифицированную запись в файл базы данных.

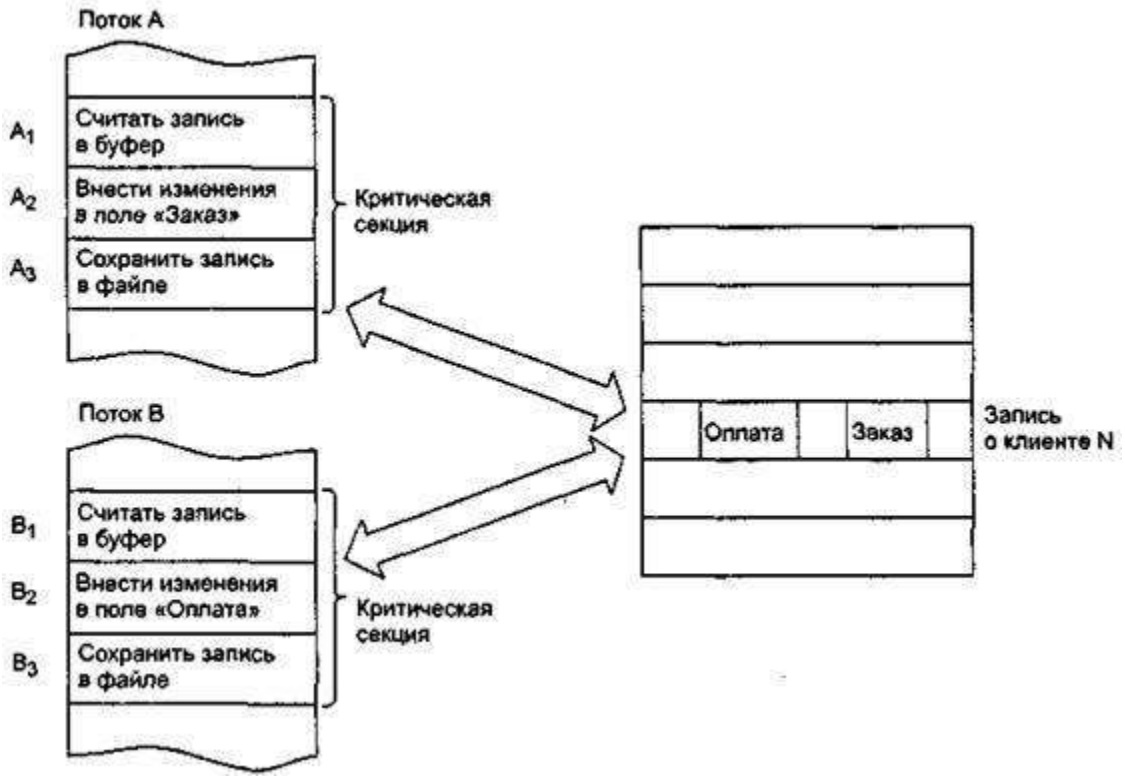



**Рис. 4.16.** Возникновение гонок при доступе к разделяемым данным

Обозначим соответствующие шаги для потока А как Al, А2 и А3, а для потока В как 81, В2 и В3. Предположим, что в некоторый момент поток А обновляет поле Заказ записи о клиенте N. Для этого он считывает эту запись в свой буфер (шаг А1), модифицирует значение поля Заказ (шаг А2), но внести запись в базу данных (шаг А3) не успевает, так как его выполнение прерывается, например, вследствие завершения кванта времени.

Предположим также, что потоку В также потребовалось внести сведения об оплате относительно того же клиента N. Когда подходит очередь потока В, он успевает считать запись в свой буфер (шаг В1) и выполнить обновление поля Оплата (шаг В2), а затем прерывается. Заметим, что в буфере у потока В находится запись о клиенте N, в которой поле Заказ имеет прежнее, не измененное значение.

Когда в очередной раз управление будет передано потоку А, то он, продолжая свою работу, запишет запись о клиенте N с модифицированным полем Заказ в базу данных (шаг А3). После прерывания потока А и активизации потока В последний запишет в базу данных поверх только что обновленной записи о клиенте N свой вариант записи, в которой обновлено значение поля Оплата. Таким образом, в базе данных будут зафиксированы сведения о том, что клиент N произвел оплату, но информация о его заказе окажется потерянной (рис. 4.17, а).

Сложность проблемы синхронизации кроется в нерегулярности возникающих ситуаций. Так, в предыдущем примере можно представить и другое развитие событий: могла быть потеряна информация не о заказе, а об оплате (рис. 4.17, *б)*

или, напротив, все исправления были успешно внесены (рис. 4.17, *в).* Все определяется взаимными скоростями потоков и моментами их прерывания. Поэтому отладка взаимодействующих потоков является сложной задачей. Ситуации, подобные той, когда два или более потоков обрабатывают разделяемые данные и конечный результат зависит от соотношения скоростей потоков, называются *гонками.*

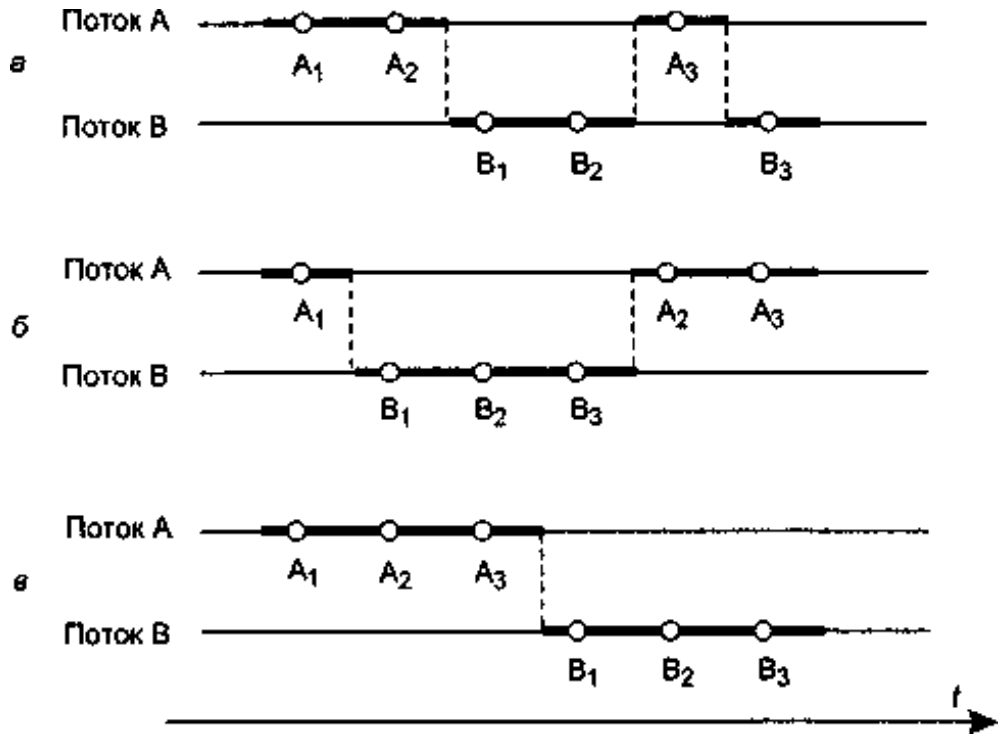



**Рис. 4.** Влияние относительных скоростей потоков на результат решения задачи

**Критическая секция**

Важным понятием синхронизации потоков является понятие «критической секции» программы. *Критическая секция* — это часть программы, результат выполнения которой может непредсказуемо меняться, если переменные, относящиеся к этой части программы, изменяются другими потоками в то время, когда выполнение этой части еще не завершено. Критическая секция всегда определяется по отношению к определенным *критическим данным,* при несогласованном изменении которых могут возникнуть нежелательные эффекты. В предыдущем примере такими критическими данными являлись записи файла базы данных. Во всех потоках, работающих с критическими данными, должна быть определена критическая секция. Заметим, что в разных потоках критическая секция состоит в общем случае из разных последовательностей команд.

Контрольные вопросы

1.Как организована синхронизация процессов в ОС?

2. ОбЪясните необходимость синхронизации и гонки;

3. Что такое критическая секция.

**Моделирование взаимоблокировок. Методы борьбы с взаимоблокировками**

План

1. Условия необходимые для взаимоблокировки;

2. Моделирование взаимоблокировок;

3. Методы борьбы с взаимоблокировками.

**Взаимоблокировка процессов** может происходить, когда несколько процессов борются за один ресурс.

Ресурсы бывают выгружаемые и невыгружаемые, аппаратные и программные.

**Выгружаемый ресурс** - этот ресурс безболезненно можно забрать у процесса (например: память).

**Невыгружаемый ресурс -** этот ресурс нельзя забрать у процесса без потери данных (например: принтер).

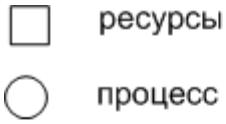
Проблема взаимоблокировок процессов возникает при борьбе за невыгружаемый ресурсы.

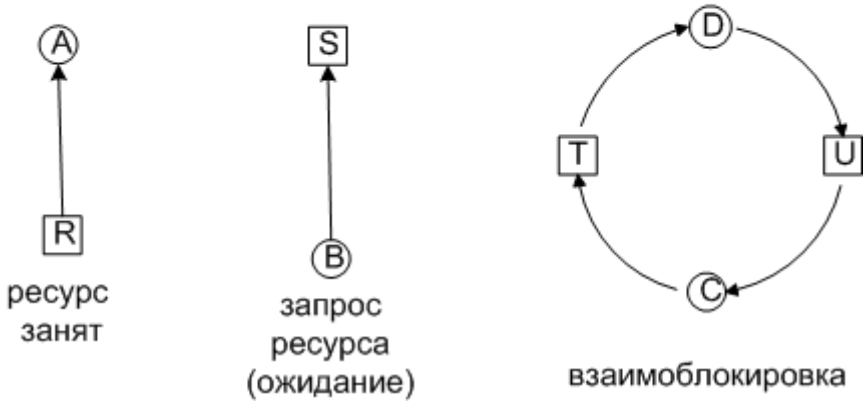
**Условия необходимые для взаимоблокировки:**

1. Условие взаимного исключения - в какой-то момент времени, ресурс занят только одним процессом или свободен.
2. Условие удержания и ожидания - процесс, удерживающий ресурс может запрашивать новые ресурсы.
3. Условие отсутствия принудительной выгрузки ресурса.
4. Условие циклического ожидания - должна существовать круговая последовательность из процессов, каждый, из которого ждет доступа к ресурсу, удерживаемому следующим членом последовательности.

## 5.2 Моделирование взаимоблокировок

Моделирование тупиков с помощью графов.



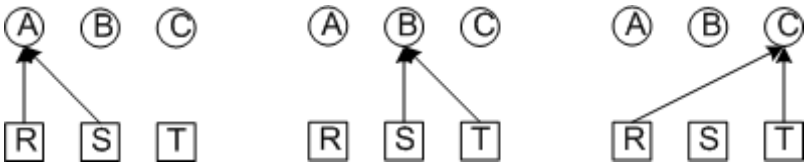

Условные обозначения

На такой модели очень хорошо проверить возникает ли взаимоблокировка. Если есть цикл, значит, есть и взаимоблокировка.

Рассмотрим простой пример:

три процесса A, B, C

три ресурса R, S, T

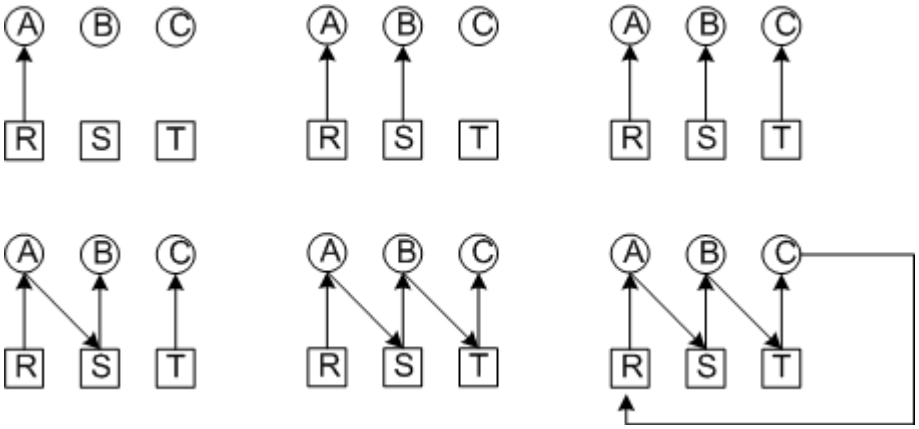



Последовательное выполнение процессов, взаимоблокировка не возникает

Рассмотрим циклический алгоритм:

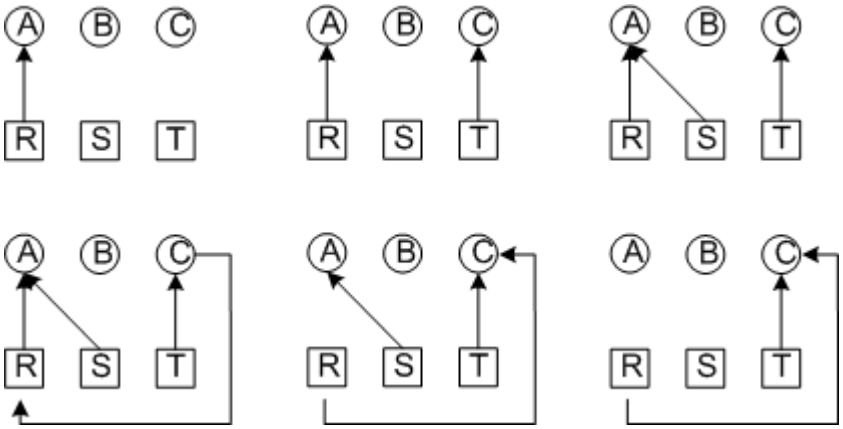
три процесса A, B, C

три ресурса R, S, T




Возникает взаимоблокировка

Рассмотрим тот же самый случай, но допустим, что система, зная о предстоящей взаимоблокировке, заблокирует процесс **B.**




Взаимоблокировка не возникает.

### 5.3 Методы борьбы с взаимоблокировками

Четыре стратегии избегания взаимоблокировок:

1. Пренебрежением проблемой в целом (вдруг пронесет).
2. Обнаружение и устранение (взаимоблокировка происходит, но оперативно ликвидируется).
3. Динамическое избежание тупиков.
4. Предотвращение четырех условий, необходимых для взаимоблокировок.

#### 5.3.1 Пренебрежением проблемой в целом (страусовый алгоритм)

Если вероятность взаимоблокировки очень мала, то ею легче пренебречь, т.к. код исключения может очень усложнить ОС и привести к большим ошибкам. Также многие взаимоблокировки тяжело обнаружить.

Этот алгоритм используется как в UNIX, так и в Windows.

Поэтому (и не только) на серверах часто устанавливают автоматическую перезагрузку (раз в сутки, как правило ночью), если возникнет взаимоблокировка, то после перезагрузки ее не будет.

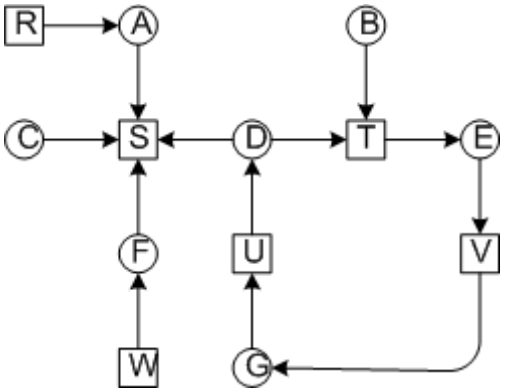
#### 5.3.2 Обнаружение и устранение взаимоблокировок

Система не пытается предотвратить взаимоблокировку, а пытается обнаружить ее и устранить.

**Обнаружение взаимоблокировки при наличии одного ресурса каждого типа**

Под одним ресурсом каждого типа, подразумевается один принтер, один сканер и один плоттер и т.д.

Рассмотрим систему из 7-ми процессов и 6-ти ресурсов.




Обнаружение взаимоблокировки при наличии одного ресурса каждого типа

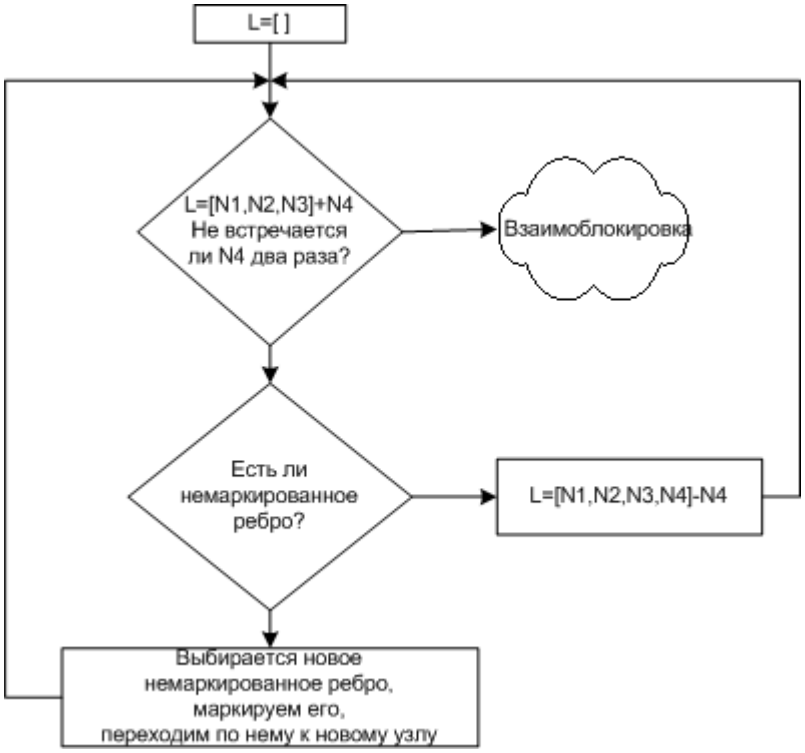
Визуально хорошо видна взаимоблокировка, но нам нужно чтобы ОС сама определяла взаимоблокировку.

Для этого нужен алгоритм.

Рассмотрим один из алгоритмов.

Для каждого узла N в графе выполняется пять шагов.

1. Задаются начальные условия: L-пустой список, все ребра не маркированы.
2. Текущий узел добавляем вконец списка L и проверяем количество появления узла в списке. Если он встречается два раза, значит цикл и взаимоблокировка.
3. Для заданного узла смотрим, выходит ли из него хотя бы одно немаркированное ребро. Если да, то переходим к шагу 4, если нет, то переходим к шагу 5.
4. Выбираем новое немаркированное исходящее ребро и маркируем его. И переходим по нему к новому узлу и возвращаемся к шагу 3.
5. Зашли в тупик. Удаляем последний узел из списка и возвращаемся к предыдущему узлу. Возвращаемся к шагу 3. Если это первоначальный узел, значит, циклов нет, и алгоритм завершается.




Алгоритм обнаружения взаимоблокировок

Для нашего случая тупик обнаруживается в списке L=[B,T,E,V,G,U,D,T]

Контрольные вопросы

1. Какие условия необходимые для взаимоблокировки?
2. Как осуществляется моделирование взаимоблокировок;
3. Поясните методы борьбы с взаимоблокировками.

## Организация памяти

План

1. Функции управления памятью в ОС
2. Типы адресов
3. Методы распределения памяти в ОС
4. Принцип кэширования данных в ОС

Функции управления памятью в ОС

Операционная система решает следующие задачи:

- Отслеживание свободной и занятой памяти.
- Выделение и освобождение памяти по запросам процессов.
- Обеспечение настройки адресов.
- Поддержка механизма виртуальной памяти

Типы адресов

Для идентификации переменных и команд используются символьные имена (метки), виртуальные адреса и физические адреса.

**Символьные имена**

Символьные имена присваивает пользователь при написании программы.

**Виртуальные адреса**

Виртуальные адреса вырабатывает компилятор. Так как не известно, в какое место оперативной памяти будет загружена программа, то компилятор присваивает переменным и командам виртуальные (условные) адреса, обычно считая по умолчанию, что программа будет размещена, начиная с нулевого адреса. Совокупность виртуальных адресов процесса называется **виртуальным адресным пространством**. Каждый процесс имеет собственное виртуальное адресное пространство.

**Физические адреса**

Физические адреса соответствуют номерам ячеек оперативной памяти, где в действительности расположены или будут расположены переменные и команды. Переход от виртуальных адресов к физическим может осуществляться двумя способами.

В первом случае замену виртуальных адресов на физические делает специальная системная программа - перемещающий загрузчик. Перемещающий загрузчик на основании имеющихся у него исходных данных о начальном адресе физической памяти, в которую предстоит загружать программу, и информации, предоставленной компилятором об адресно-зависимых константах программы, выполняет загрузку программы, совмещая ее с заменой виртуальных адресов физическими.

Второй способ заключается в том, что программа загружается в память в неизмененном виде в виртуальных адресах, при этом операционная система фиксирует смещение действительного расположения программного кода относительно виртуального адресного пространства. Во время выполнения программы при каждом обращении к оперативной памяти выполняется преобразование виртуального адреса в физический.

Второй способ является более гибким, он допускает перемещение программы во время ее выполнения, в то время как перемещающий загрузчик жестко привязывает программу к первоначально выделенному ей участку памяти. Вместе с тем использование перемещающего загрузчика уменьшает накладные расходы, так как преобразование каждого виртуального адреса происходит только один раз во время загрузки, а во втором случае - каждый раз при обращении по данному адресу.

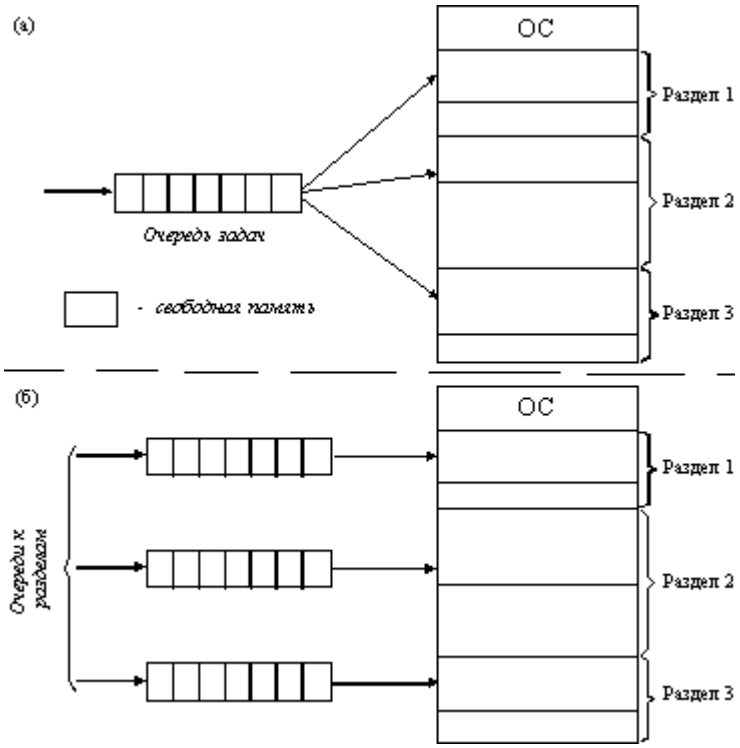
Иногда (обычно в специализированных системах) заранее точно известно, в какой области оперативной памяти будет выполняться программа, и компилятор выдает исполняемый код сразу в физических адресах.

Методы распределения памяти в ОС

Выделяют следующие методы распределения памяти:

**Методы распределения памяти без использования дискового пространства**

**Распределение памяти фиксированными разделами**




Подсистема управления памятью в этом случае выполняет следующие задачи:

- сравнивая размер программы, поступившей на выполнение, и свободных разделов, выбирает подходящий раздел,
- осуществляет загрузку программы и настройку адресов.

**Достоинства**:

- с общей очередью - простота
- с отдельными очередями - б**о**льшая производительность.

**Недостатки**:

- Неэффективное распределение памяти (большие незаполненные фрагменты).
- Размер приложения может быть больше размера раздела.
- Перед запуском можно переразбить разделы.

**Распределение памяти разделами переменной величины**

**С неперемещаемыми разделами**

ОС создает под каждую задачу раздел требуемого размера, когда задача завершается, то раздел освобождается.

**Достоинства**:

- Снимается необходимость организации очередей.
- Больше шансов получить память нужного размера.
- Экономичнее.

**Недостатки**:

- Фрагментация (свободный блок памяти оказывается разрезан) - не всякое приложение может быть запущено.

**С перемещаемыми разделами**

Фрагментация сжатия (перемещения): при каждом освобождении памяти разделы смещаются в сторону старших (младших) адресов.

**Достоинство**: нет фрагментации.

**Недостаток**: снижение производительности.

**Способы упорядочивания адресов**

Выделяют следующие способы упорядочивания адресов:

1. Произведение настройки адресов, когда приложение запущено (перемещающийся загрузчик).
2. Динамическое преобразование виртуальных адресов в физические.

**Способы борьбы с фрагментацией**

В определенные моменты времени происходит сжатие в сторону младших адресов. ОС последовательно просматривает занятые приложениями блоки, находит разрыв и смещает адрес. Данная схема эффективна в случае динамического преобразования; если используется перемещающийся загрузчик, то происходит полный пересчет адресов, который является очень ресурсоемким.

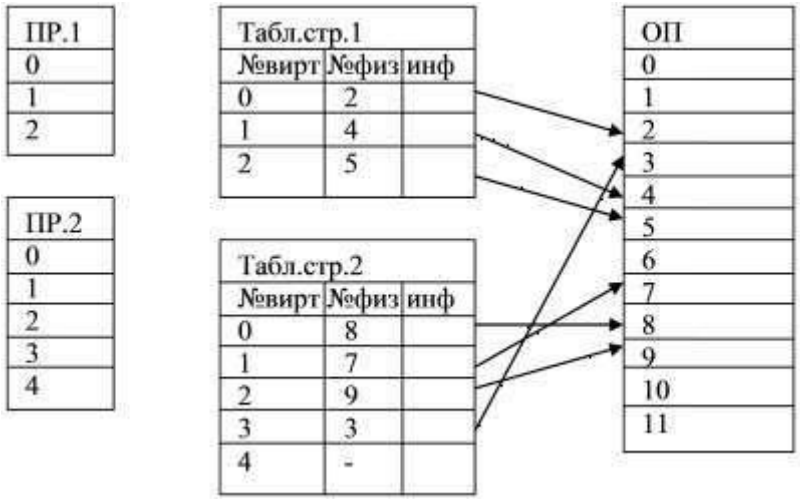
**Методы распределения памяти с использования дискового пространств**

**Метод оверлеев**

Был использован и реализован в ОС MS DOS. Программист разбивает приложение на несколько частей (оверлеев), умещающихся в доступную память (640 кб). Сначала происходит загрузка 0-го оверлея, затем он выгружается и загружается 1-ый, и так далее. Программа содержится на диске, активный оверлей - в оперативной памяти, а все механизмы обеспечиваются программистом. Механизм требует тщательного проектирования программ.

**Страничное распределение виртуальной памяти**

Адресное пространство процесса делится на страницы фиксированных разделов. Физическая память в системе тоже делится на страницы, аналогичные виртуальной памяти. Для каждого процесса ОС создает служебную структуру - таблицу страниц (дает однозначное отображение виртуальных страниц в физические).



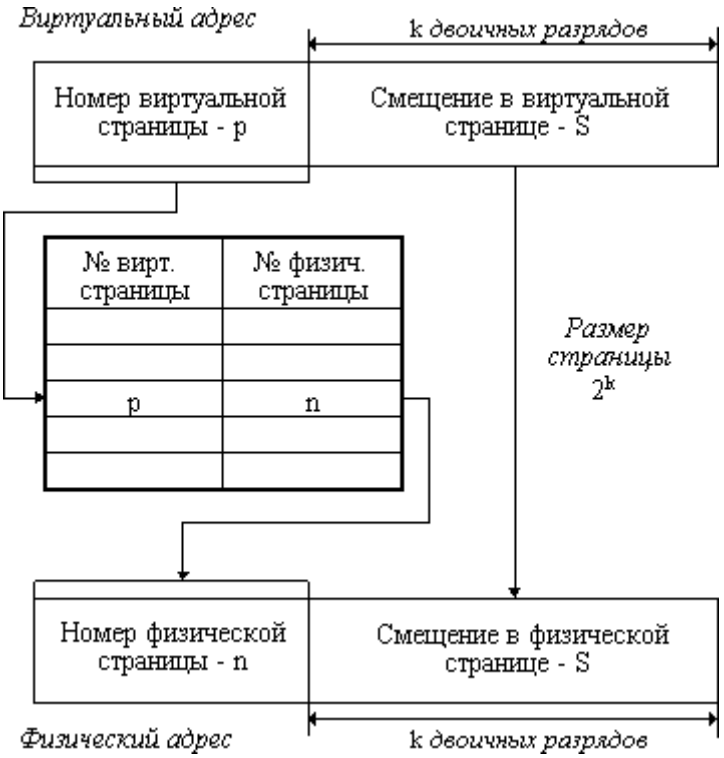

При активизации очередного процесса в специальный регистр процессора загружается адрес таблицы страниц данного процесса.

При каждом обращении к памяти происходит чтение из таблицы страниц информации о виртуальной странице, к которой произошло обращение. Если данная виртуальная страница находится в оперативной памяти, то выполняется преобразование виртуального адреса в физический. Если же нужная виртуальная страница в данный момент выгружена на диск, то происходит так называемое страничное прерывание. Выполняющийся процесс переводится в состояние ожидания, и активизируется другой процесс из очереди готовых. Параллельно программа обработки страничного прерывания находит на диске требуемую виртуальную страницу и пытается загрузить ее в оперативную память. Если в памяти имеется свободная физическая страница, то загрузка выполняется немедленно, если же свободных страниц нет, то решается вопрос, какую страницу следует выгрузить из оперативной памяти. Страницы выбираются согласно информации, заносимой в колонну **Флаги**.

**Виды флагов**:

- время обработки
- было ли обращение или нет
- признак невыгружаемых страниц
- частота использования

Рассмотрим механизм преобразования виртуального адреса в физический при страничной организации памяти.




Виртуальный адрес при страничном распределении может быть представлен в виде пары (p, s), где p - номер виртуальной страницы процесса (нумерация страниц начинается с 0), а s - смещение в пределах виртуальной страницы. Учитывая, что размер страницы равен 2 в степени к, смещение s может быть получено простым отделением k младших разрядов в двоичной записи виртуального адреса. Оставшиеся старшие разряды представляют собой двоичную запись номера страницы p.

При каждом обращении к оперативной памяти аппаратными средствами выполняются следующие действия:

- на основании начального адреса таблицы страниц, номера виртуальной страницы и длины записи в таблице страниц определяется адрес нужной записи в таблице,
- из этой записи извлекается номер физической страницы
- к номеру физической страницы присоединяется смещение.

Таблицу страниц стремятся размещать в "быстрых" запоминающих устройствах. Это может быть, например, набор специальных регистров или память, использующая для уменьшения времени доступа ассоциативный поиск и кэширование данных.

Страничное распределение памяти может быть реализовано в упрощенном варианте, без выгрузки страниц на диск. В этом случае все виртуальные страницы всех процессов постоянно находятся в оперативной памяти. Такой вариант страничной организации хотя и не предоставляет пользователю виртуальной памяти, но почти исключает фрагментацию.

**Достоинства**:

- нет фрагментации
- память используется оптимально
- механизм не требует никаких действий со стороны программы
- быстрое преобразование виртуальных адресов в физические.

**Недостатки**:

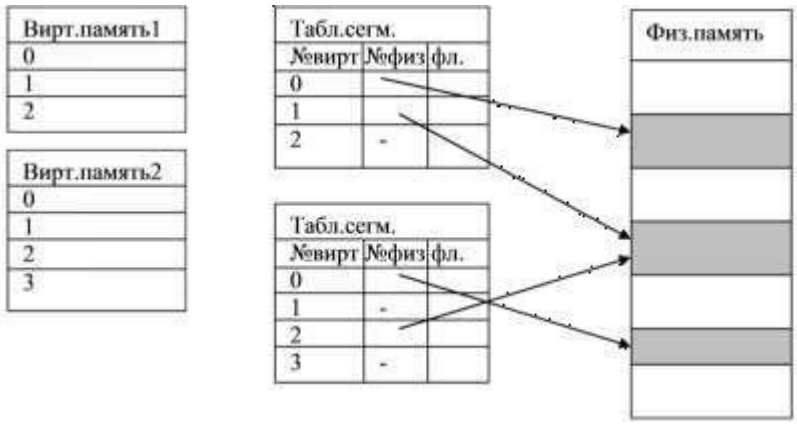
- при малых страницах высокие расходы при хранении таблицы страниц

- нет возможности указать тип содержащейся информации, поэтому нельзя установить права доступа.

**Сегментное распределение памяти**

Виртуальное адресное пространство процесса делится на сегменты, размер которых определяется программистом с учетом смыслового значения содержащейся в них информации. Отдельный сегмент может представлять собой подпрограмму, массив данных и т.п. Иногда сегментация программы выполняется по умолчанию компилятором.

При загрузке процесса часть сегментов помещается в оперативную память (при этом для каждого из этих сегментов операционная система подыскивает подходящий участок свободной памяти), а часть сегментов размещается в дисковой памяти. Сегменты одной программы могут занимать в оперативной памяти несмежные участки. Во время загрузки система создает таблицу сегментов процесса (аналогичную таблице страниц), в которой для каждого сегмента указывается начальный физический адрес сегмента в оперативной памяти, размер сегмента, правила доступа, признак модификации, признак обращения к данному сегменту за последний интервал времени и некоторая другая информация. Если виртуальные адресные пространства нескольких процессов включают один и тот же сегмент, то в таблицах сегментов этих процессов делаются ссылки на один и тот же участок оперативной памяти, в который данный сегмент загружается в единственном экземпляре.



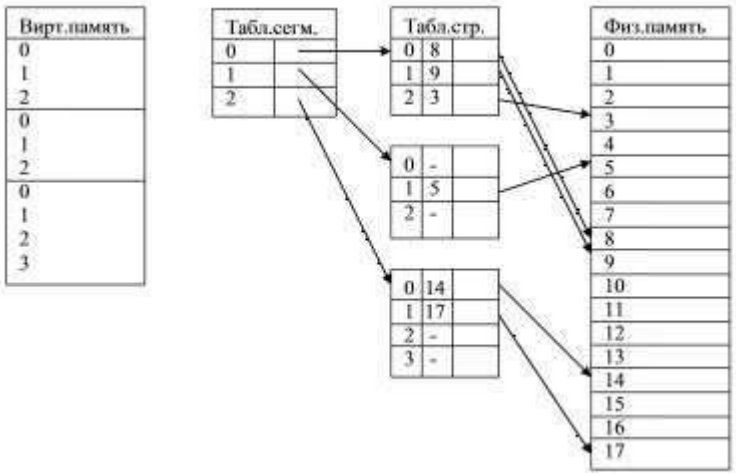

Система с сегментной организацией функционирует аналогично системе со страничной организацией: время от времени происходят прерывания, связанные с отсутствием нужных сегментов в памяти, при необходимости освобождения памяти некоторые сегменты выгружаются, при каждом обращении к оперативной памяти выполняется преобразование виртуального адреса в физический. Кроме того, при обращении к памяти проверяется, разрешен ли доступ требуемого типа к данному сегменту.

Виртуальный адрес при сегментной организации памяти может быть представлен парой (g, s), где g - номер сегмента, а s - смещение в сегменте. Физический адрес получается путем сложения начального физического адреса сегмента, найденного в таблице сегментов по номеру g, и смещения s.

**Недостатком** данного метода распределения памяти является фрагментация на уровне сегментов и более медленное по сравнению со страничной организацией преобразование адреса.

**Сегментно-страничная организация разделения памяти**

Данный метод представляет собой комбинацию страничного и сегментного распределения памяти и, вследствие этого, сочетает в себе достоинства обоих подходов. Виртуальное пространство процесса делится на сегменты, а каждый сегмент в свою очередь делится на виртуальные страницы, которые нумеруются в пределах сегмента. Оперативная память делится на физические страницы. Загрузка процесса выполняется операционной системой постранично, при этом часть страниц размещается в оперативной памяти, а часть на диске. Для каждого сегмента создается своя таблица страниц, структура которой полностью совпадает со структурой таблицы страниц, используемой при страничном распределении. Для каждого процесса создается таблица сегментов, в которой указываются адреса таблиц страниц для всех сегментов данного процесса. Адрес таблицы сегментов загружается в специальный регистр процессора, когда активизируется соответствующий процесс.



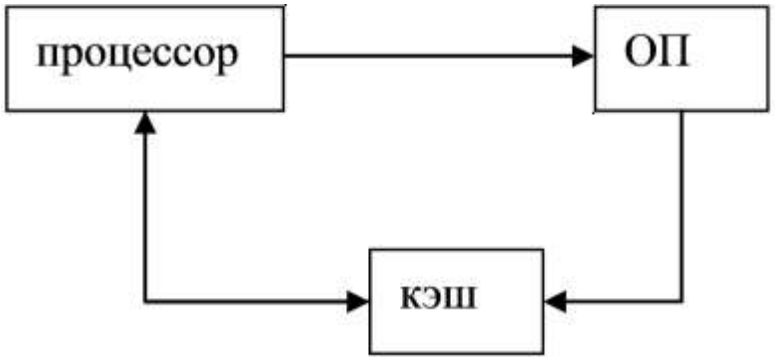

**Достоинство**: данный механизм поддерживается процессорами, поэтому работает быстрее.

**Недостаток**: большие объемы таблиц.

Принцип кэширования данных в ОС

**Кэш-память** - это способ организации совместного функционирования двух типов запоминающих устройств, отличающихся временем доступа и стоимостью хранения данных, который позволяет уменьшить среднее время доступа к данным за счет динамического копирования в "быстрое" ЗУ наиболее часто используемой информации из "медленного" ЗУ.

Кэш-памятью часто называют не только способ организации работы двух типов запоминающих устройств, но и одно из устройств - "быстрое" ЗУ. Оно стоит дороже и, как правило, имеет сравнительно небольшой объем. Важно, что механизм кэш-памяти является прозрачным для пользователя, который не должен сообщать никакой информации об интенсивности использования данных и не должен никак участвовать в перемещении данных из ЗУ одного типа в ЗУ другого типа, все это делается автоматически системными средствами.




В системах, оснащенных кэш-памятью, каждый запрос к оперативной памяти выполняется в соответствии со следующим алгоритмом:

1. Просматривается содержимое кэш-памяти с целью определения, не находятся ли нужные данные в кэш-памяти; кэш-память не является адресуемой, поэтому поиск нужных данных осуществляется по содержимому - значению поля "адрес в оперативной памяти", взятому из запроса.
2. Если данные обнаруживаются в кэш-памяти, то они считываются из нее, и результат передается в процессор.
3. Если нужных данных нет, то они вместе со своим адресом копируются из оперативной памяти в кэш-память, и результат выполнения запроса передается в процессор. При копировании данных может оказаться, что в кэш-памяти нет свободного места, тогда выбираются данные, к которым в последний период было меньше всего обращений, для вытеснения из кэш-памяти. Если вытесняемые данные были модифицированы за время нахождения в кэш-памяти, то они переписываются в оперативную память. Если же эти данные не были модифицированы, то их место в кэш-памяти объявляется свободным.

Контрольные вопросы

1. Расскажите про функции управления памятью в ОС?
2. Какие типы адресов используют ОС?
3. Охарактеризуйте методы распределения памяти в ОС?
4. Что такое кэширования данных в ОС?

## Алгоритмы посещения страниц

План

1. Страничная организация памяти (также таблицы страниц)
2. Сегментная организация пямяти (таблицы сегментов)
3. Страничное прерывание (page fauet) и виртуальная память
4. Механизмы замещения страниц: Алгоритм Belady, FIFO Алгоритм LRU (Last recently used) – наименее используемое, Not Frequently Used (алгоритм clock – часовой)
5. Рабочее множество
6. Алгоритм Page Fault Frequency

Организация памяти в виде страниц борется с двумя проблемами:

1. *Внешней фрагментацией* – используются блоки фиксированного размера в виртуальной и физической памяти, т.е. все запросы на выделение памяти будут кратны, не будет оставаться некратных зон.
2. *Внутренней фрагментацией* – блоки достаточно малого размера, поэтому (К) будет мал.

**С точки зрения программиста**:

- Процессам виртуальное адресное пространство предоставляется непрерывным, от байта 0 до байта N;
- N зависит от аппаратной поддержки (например 32бита — адрресное пространство 4Гб), делится соответственно.
- В реальности виртуальные страницы распределены по страницам физической памяти далеко не непрерывно и не один к одному. Это два разных мира – физические страницы и виртуальные страницы. Это ключевой аспект, который надо понимать.

**С точки зрения менеджера памяти:**

- Эффективное использование памяти из-за очень низкой внутренней фрагментации.
- Внешняя фрагментация полностью отсутствует и не нужно дефрагментировать.

**С точки зрения защиты:**

- Процесс имеет доступ только к своему адресному пространству.

Допущение – все страницы виртуальной памяти всегда находятся в страницах физической памяти.

Не будем думать, что есть только виртуальные страницы, а физических – их нет, т.е. полное отображение виртуальной и физической памяти.

Предположим, что все страницы резидентно в памяти, это необходимо, чтобы понять как работает **трансляция адресов**.

**Трансляция адресов**

**Трансляция виртуального адреса:** Виртуальный адрес состоит из двух частей: **номер виртуальной страницы (VPN)** и смещение внутри страницы

**Номер виртуальной страницы (VPN- virtual page number)** это индекс в таблице страниц (Pagetable).

**Запись в таблице страниц (PTE – page table entry)** содержит номер фрейма (**PFN** –page frame number).

**Номер фрейма** – это номер физической страницы.

**Фрейм** – это страница физической памяти.

Смысл таблицы страниц – одна запись в таблице страниц (PTE) на одну страницу виртуального адресного пространства (VPN), отображает VPN на PFN.

Какая **виртуальная страница** соответствует какому **фрейму физической** памяти.


Трансляция адресов

Есть виртуальный адрес, состоящий из двух частей:

1. **№виртуальной страницы**, по нему идет поиск в таблице страниц и находится № фрейма, он составляет одну часть физического адреса;
2. **Смещение** – берется напрямую – вторая часть физического адреса.

По новому адресу осуществляется поиск уже в физической памяти и доступ к данным.

Существуют два процесса 0 и 1, у 0 есть две страницы у 1 процесса четыре страницы. И мы видим, что они могут отображаться как угодно, вплоть до того, что две виртуальные страницы могут отображаться на одной физической.

Зачастую это бывает полезно, простор для манипуляций ограниченный и в этом состоит вся прелесть виртуального адресного пространства.

**Страничная организация памяти — Пример**

-32битная разрядность адресов

-Размер страницы 4096байт

-VPN длиной 20бит, смещение 12бит (20+12=32бита)

-Преобразуем виртуальный адрес **0* 43456 323**

— **№ вирт. страницы смещение внутри страницы**

-VPN=43456 смещение=323

-Допустим, что в ячейке таблицы страниц по индексу 0*43456 находится значение фрейма PFN= 0*1002.

Получаем физический адрес **0*01002323**

**Таблица страниц (PTE)**

Если есть таблица страниц, которая содержит преобразование адреса, то необходимо воспользоваться и нагрузить ее дополнительными функциями:

- Добавить защиту доступа;
- Добавить дополнительную вспомогательную информацию (например, используется эта вирт.страница или нет, был ли к ней когда-либо осуществлен доступ, была ли в нее осуществлена запись…);

Таблица страниц превращается в сложную структуру данных, которые начинает использоваться множеством всяких дополнительных полей.

Все это нужно, чтобы сохранить память для других процессов, делать все быстро и не плодить десятки новых таблиц с данными – все по возможности хранить внутри таблицы PTE.

Приведем пример стректуры таблицы страниц PTE.

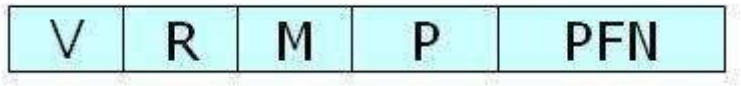



таблица страниц PTE

Если мы посмотрим на запись в PTE, то мы увидим, что туда можно поместить:

- **V**— может ли использоваться данная запись PTE (valid or not) – бит валидности, бит присутствия;
- **R-** был ли доступ к этой странице;
- **M**– была ли страница модифицирована;
- **P**– какие операции разрешены (битовая маска операций);
- **PFN**– номер фрейма (как основной, как основная нагрузка).

**Преимущества страничной памяти**

- **Легко выделять физическую память**

— Списки свободных фреймов, выделить фрейм
– просто удалить из списка свободных; — Внешняя фрагментация не проблема, т.к. фреймы одного размера;

- Естественный подход

— Всей программе не нужно быть резидентной – это «побочный» продукт;
— Все страницы одного размера;
— Основа – устранение внешней фрагментации.

**Недостатки страничной памяти**

- **Внутренняя фрагментация**

— Процессам может быть нужны размеры, некратные размеру страницы;
— По сравнению с размером адресного пространства, размер страницы очень мал.

- **Накладные расходы при обращении к памяти**

– вначале к таблице страниц, а затем уже к памяти.
**Решение:** аппаратный КЭШ для обращений к таблице страниц (**TLB** translation lookaside buffer – буфер внутри процессора).

- Большой объем памяти, требуемый для хранения таблиц страниц.

Один PTE на одну страницу в виртуальном адресном пространстве

Пример:

Архитектура x86

32-битное АП с 4КБ страницами = 2 в 20 степени PTE= 1048576записей PTE

4 байта на PTE = 4Мб памяти на таблицу страниц.

В ОС создаются отдельные таблицы страниц для каждого процесса.

Итого, например 25процессов * 4Мб = 100Мб

Соответственно нужны отдельные таблицы страниц для каждого процесса.

**Решение: хранить таблицы страниц в страничной памяти)**

**Страничная организация памяти (обобщение)**

- Решает разные проблемы, типа фрагментации

- Адресное пространство – линейный массив байтов
- Разделяется на страницы одинакового размера (например 4Кб)
- Использует таблицы страниц для отображения виртуальной страницы на физический фрейм

**Сегментное распределение памяти (Сегментация)**

Сегментация подходит к распределению памяти более «продвинуто», чем страничная — выделяются разные логические сегменты памяти: стек, код программы, куча, т.е. адресное пространство делится на логические блоки.

Блоки имеют свой размер, расположение и права доступа, например, куча – для динамической памяти программы – все это **отдельные логические блоки памяти**.

Их не нужно мешать в одно, поэтому следующим этапом было разделение всего названного и ввод виртуального адреса в виде пары: **Сегмент, смещение.**

**Виртуальный адрес в виде – сегмент + смещение.**

Страничная организация предполагает куски памяти, как в примере выше, 4096байт (4Кб), таких кусков можно брать сколько угодно.

**Сегментация:**

- Позволяет разделить разные участки памяти в соответствии с их назначением
- Динамический (изменяемый) размер у сегментов

**Два варианта**

1.Один сегмент на процесс – переменный раздел

2.Много сегментов на процесс — сегментация

**Аппаратная поддержка сегментации**

- Одна пара база-предел(лимит) на сегмент
- Сегменты идентифицируются №, который является индексом в таблице
- Виртуальный адрес = пара <СЕГМЕНТ, СМЕЩЕНИЕ>
- Физический адрес = база сегмента + смещение

**Недостатки:**

- Все недостатки, присущие организации памяти разделами переменной длины присущи и сегментной организации
- Внешняя фрагментация

Лучшим, как можно заметить является организация этих подходов в один. Давайте **объединим страничную и сегментную адресацию.**

Архитектура x86 поддерживает и страничную и сегментную адресацию.

- Сегменты используются для управления логическими блоками, обычно сегменты большие (помещают множество страниц).
- Сегменты разбиваются на страницы, у каждого сегмента своя таблица страниц

В современных ОС достаточно ограниченно применяются сегменты, много их не применяется. **Пример ОС Linux**

- Один сегмент кода ядра, Один сегмент данных ядра
- Один сегмент кода пользователя, Один сегмент данных пользователя
- Все сегменты организованы странично.

**Страничная виртуальная память**

Мы предполагали ранее, что вся память резидентная — не рассматривался вариант, что какой-либо страницы может не быть в физической памяти.

Сейчас, допустим, что адресное пространство может и не быть полностью резидентным. На практике так в основном и есть – память никогда не резидентна.

Процессов сотни, на них выделяется много виртуальной памяти и ее не хватит на отображение в физической памяти.

Например, 100 процессов 100П*4Гб=400ГБ ОП.

Соответственно какая то часть адресного пространства находится в физической памяти, какая то часть в некоторой **вторичной памяти** (понимается дисковая подсистема), абстрагируется от физической реализации.

С точки зрения опрерационной системы важно, что ОС использует основную память, как КЭШ.

У ОС есть медленная память (это дисковая подсистема), она может туда загружать процессы и выгружать их оттуда, а основная память используется как ограниченный ресурс, сродни классическому кэшированию.

Принцип работы достаточно простой – **нужная страница перемещается в свободный фрейм физической памяти из вторичной памяти**. **А если свободных фреймов нет, то какая-либо страница выгружается на диск, таким образом освобождается драгоценный фрейм физической памяти.**

Важно отметить, запись во вторичной памяти происходит только тогда, когда страница в основной памяти была модифицирована, если она не была модифицирована, то соответственно и выгружать нечего.

Весь процесс происходит прозрачно для программы. Никакая программа не управляет напрямую, какую страницу ей выгрузить в основную память, какую загрузить.

**Менеджер памяти операционной системы** все абстрагирует и делает все прозрачным для программиста, чтобы он не «напрягался» на данную тему, это сложно реализовывать самостоятельно.

Контрольные вопросы

1. Как функционирует страничная организация памяти?
2. Как функционирует сегментная организация пямяти?
3. Что такое страничное прерывание и виртуальная память
4. Какие механизмы замещения страниц вы знаете?

**Сегментация памяти**

План

1. Определение сегментации памяти;
2. Аппаратная реализация;
3. Сегментация без разбиения на страницы;
4. Сегментация с разбиением на страницы;
5. Совместное использование сегментов

Сегментация - это деление памяти на сегменты. Это механизм адресации, обеспечивающий существование нескольких независимых адресных пространств как в пределах одной задачи, так и в системе в целом для защиты задач от взаимного влияния. С точки зрения разработчиков программного обеспечения, сегментация дает удобный способ совместного использования информации несколькими процессами. Конкретный сегмент может использоваться совместно с другими без нарушения требований его

защиты. Сегментация также предполагает естественное разделение программных строк и данных и отделение модуля от модуля. [1]

Аппаратная реализация

В системе, использующей сегментацию, адреса памяти компьютера состоят из идентификатора сегмента и смещения в сегменте. Аппаратный блок управления памятью (MMU) ответственен за перевод сегмента и смещения в адрес физической памяти, и за выполнение проверок, чтобы удостовериться, что перевод может быть произведен и что ссылка на сегмент и смещение разрешены.

У каждого сегмента есть длина и связанный с ним набор полномочий (например, чтение, запись, выполнение). Процессу позволяют сделать ссылку в сегмент в том случае, если тип ссылки разрешен полномочиями, и если смещение в сегменте находится в диапазоне, определенном длиной сегмента. Иначе возникает ошибка сегментации.

Сегменты могут также использоваться, чтобы реализовать виртуальную память. В этом случае у каждого сегмента есть связанный флаг, указывающий, присутствует ли сегмент в оперативной памяти или нет. Если сегмент, к которому получают доступ, не присутствует в оперативной памяти, выбрасывается исключение, и операционная система считает сегмент в память из внешнего хранилища.

Сегментация - это один метод реализации защиты памяти. Разбивка на страницы - другой, и они могут быть объединены. Размер сегмента памяти обычно не фиксирован и может иметь размер в 1 байт.

Сегментация была реализована несколькими различными способами на различных аппаратных средствах, с или без разбивки на страницы. Сегментация памяти Intel x86 не соответствует ни одной модели и обсуждена отдельно ниже.

**Сегментация без разбиения на страницы**

Связанная с каждым сегментом информация, которая указывает, где сегмент расположен в памяти— база сегмента. Когда программа ссылается на ячейку памяти, смещение добавляется к базе, чтобы генерировать адрес физической памяти.

Реализация виртуальной памяти в системе, используя сегментацию без разбивки на страницы требует, чтобы все сегменты перемещались между оперативной памятью и внешней памятью. Когда сегмент загружен, операционная система должна выделить достаточное количество непрерывной свободной памяти, чтобы содержать весь сегмент. Часто результатом фрагментации является невозможность выделить именно непрерывный участок заданной памяти.

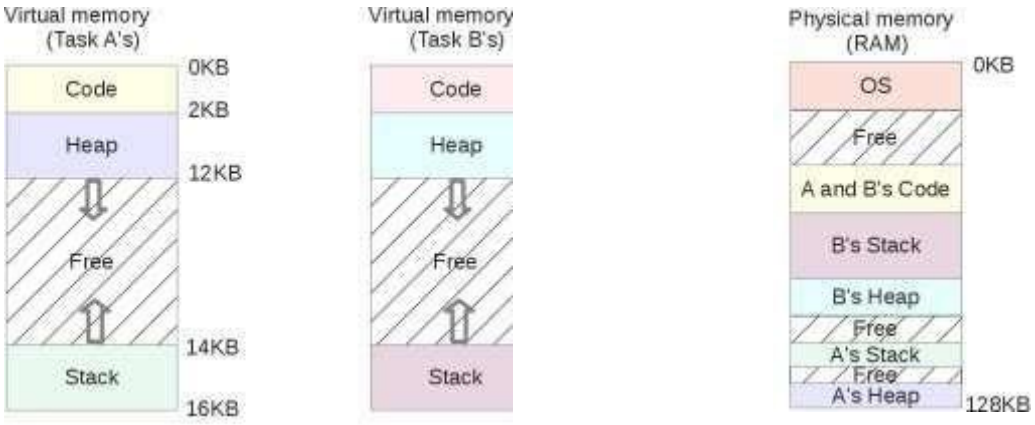
**Сегментация с разбиением на страницы**

Вместо фактической ячейки памяти информация о сегменте включает адрес таблицы страниц для сегмента. Когда программа ссылается на ячейку памяти, смещение переводится в адрес памяти, используя таблицу страниц. Сегмент может быть расширен, просто выделением другой страницы памяти и добавлением ее к таблице страниц сегмента.

Реализация виртуальной памяти в системе, используя сегментацию с разбивкой на страницы обычно только перемещает отдельные страницы назад и вперед между оперативной памятью и внешней памятью, подобно простой страничной реализации адресации памяти. Страницы сегмента могут быть расположены где угодно в оперативной памяти и не должны быть непрерывными. Обыкновенно это приводит к уменьшению ввода/вывода между основной и внешней памятью, а также к уменьшению фрагментации памяти.

Совместное использование сегментов

Сегментирование физической памяти не только не позволяет виртуальной памяти отъедать физическую, но также даёт возможность совместного использования физических сегментов с помощью виртуальных адресных пространств разных процессов.




Если дважды запустить задачу A, то кодовый сегмент у них будет один и тот же: в обеих задачах выполняются одинаковые машинные инструкции. В то же время у каждой задачи будут свои стек и куча, поскольку они оперируют разными наборами данных.

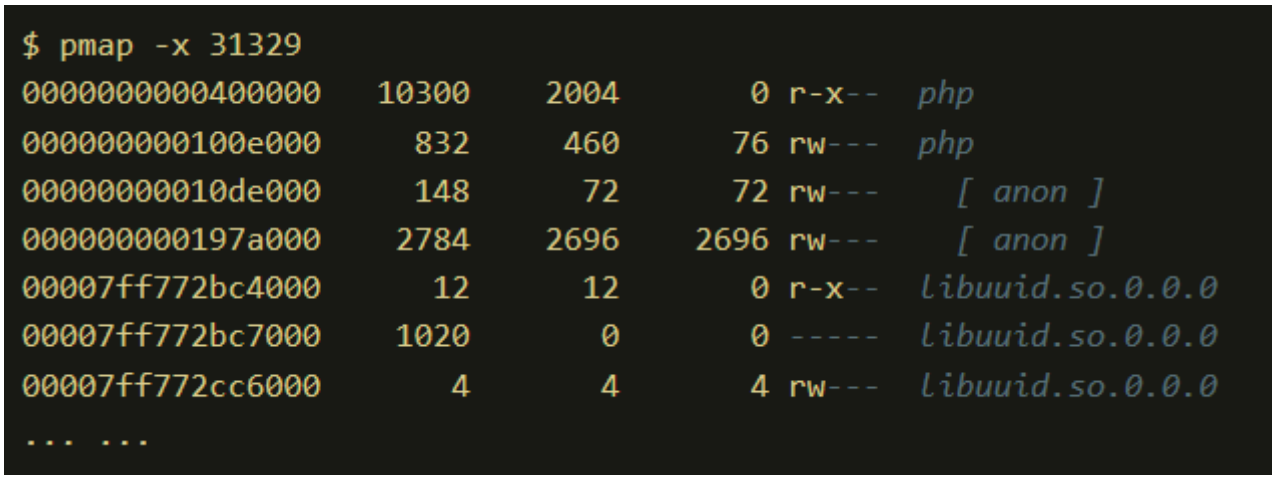
При этом оба процесса не подозревают, что делят с кем-то свою память. Такой подход стал возможен благодаря внедрению битов защиты сегмента (segment protection bits).

Для каждого создаваемого физического сегмента ОС регистрирует значение bounds, которое используется MMU для последующей переадресации. Но в то же время регистрируется и так называемый флаг разрешения (permission flag). Поскольку сам код нельзя модифицировать, то все кодовые сегменты создаются с флагами RX. Это значит, что процесс может загружать эту область памяти для последующего выполнения, но в неё никто не может записывать. Другие два сегмента — куча и стек — имеют флаги RW, то есть процесс может считывать и записывать в эти свои два сегмента, однако код из них выполнять нельзя. Это сделано для обеспечения безопасности, чтобы злоумышленник не мог повредить кучу или стек, внедрив в них свой код для получения root-прав. Так было не всегда, и для высокой эффективности этого решения требуется аппаратная поддержка. В процессорах Intel это называется “NX bit”.

Флаги могут быть изменены в процессе выполнения программы, для этого используется mprotect().

Под Linux все эти сегменты памяти можно посмотреть с помощью утилит */proc/{pid}/maps* или */usr/bin/pmap*.

Вот пример на PHP:




Здесь есть все необходимые подробности относительно распределения памяти. Адреса виртуальные, отображаются разрешения для каждой области памяти. Каждый совместно используемый объект (.so) размещён в адресном пространстве в виде нескольких частей (обычно код и данные). Кодовые сегменты являются исполняемыми и совместно используются в физической памяти всеми процессами, которые разместили подобный совместно используемый объект в своём адресном пространстве.

*Shared Objects* — это одно из крупнейших преимуществ Unix- и Linux-систем, обеспечивающее экономию памяти.

Также с помощью системного вызова *mmap()* можно создавать совместно используемую область, которая преобразуется в совместно используемый физический сегмент. Тогда у каждой области появится индекс s, означающий shared.

Ограничения сегментации

Итак, сегментация позволила решить проблему неиспользуемой виртуальной памяти. Если она не используется, то и не размещается в физической памяти благодаря использованию сегментов, соответствующих именно объёму используемой памяти.

Но это не совсем верно.

Допустим, процесс запросил у кучи 16 Кб. Скорее всего, ОС создаст в физической памяти сегмент соответствующего размера. Если пользователь потом освободит из них 2 Кб, тогда ОС придётся уменьшить размер сегмента до 14 Кб. Но вдруг потом программист запросит у кучи ещё 30 Кб? Тогда предыдущий сегмент нужно увеличить более чем в два раза, а возможно ли это будет сделать? Может быть, его уже окружают другие сегменты, не позволяющие ему увеличиться. Тогда ОС придётся искать свободное место на 30 Кб и перераспределять сегмент.

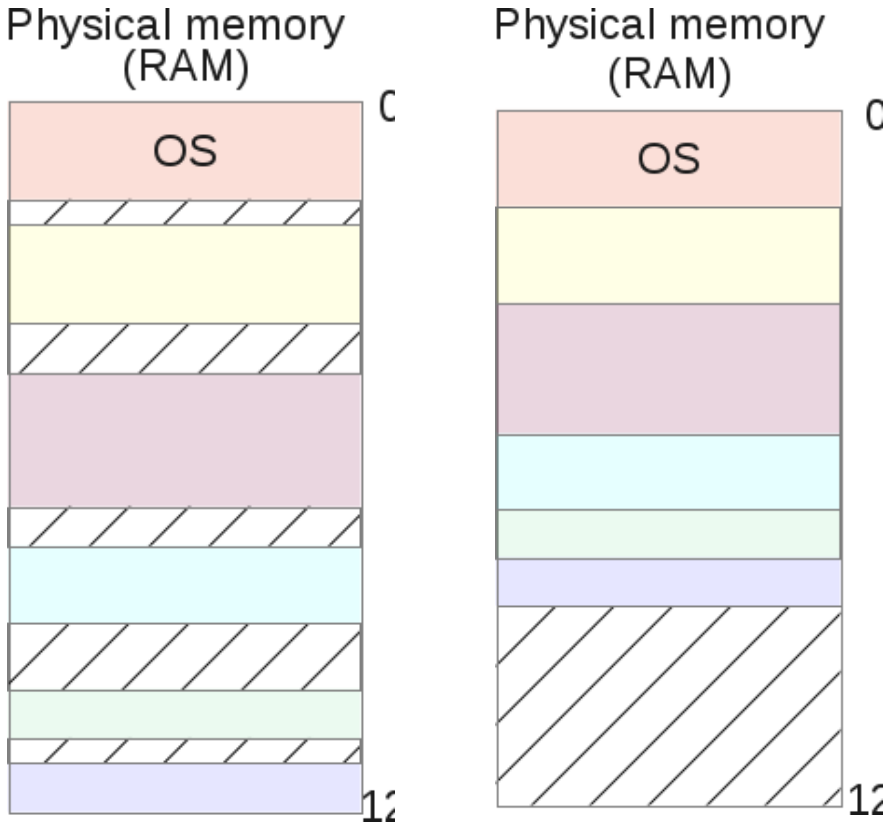
Главный недостаток сегментов заключается в том, что из-за них физическая память сильно фрагментируется, поскольку сегменты увеличиваются и уменьшаются по мере того, как пользовательские процессы запрашивают и освобождают память. А ОС приходится поддерживать список свободных участков и управлять ими.

Фрагментация может привести к тому, что какой-нибудь процесс запросит такой объём памяти, который будет больше любого из свободных участков. И в этом случае ОС придётся отказать процессу в выделении памяти, даже если суммарный объём свободных областей будет существенно больше.

ОС может попытаться разместить данные компактнее, объединяя все свободные области в один большой чанк, который в дальнейшем можно использовать для нужд новых процессов и перераспределения.

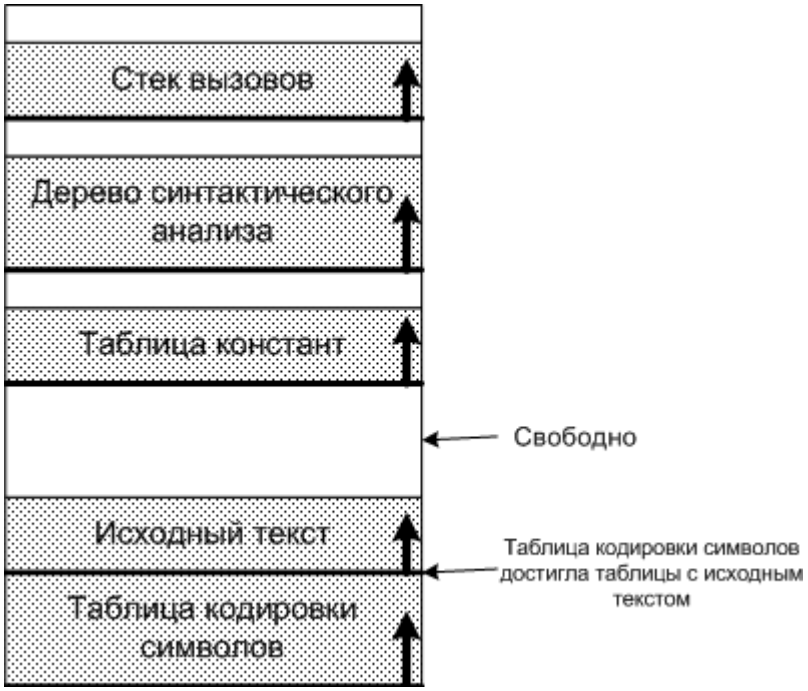
Но подобные алгоритмы оптимизации сильно нагружают процессор, а ведь его мощности нужны для выполнения пользовательских процессов. Если ОС начинает реорганизовывать физическую память, то система становится недоступной.

Так что сегментация памяти влечёт за собой немало проблем, связанных с управлением памятью и многозадачностью.Нужно как-то улучшить возможности сегментации и исправить недостатки. Это достигается с помощью ещё одного подхода — страниц виртуальной памяти.




Сравнение сегментированной памяти с использованием одного адресного пространства.

**Одно адресное пространство**



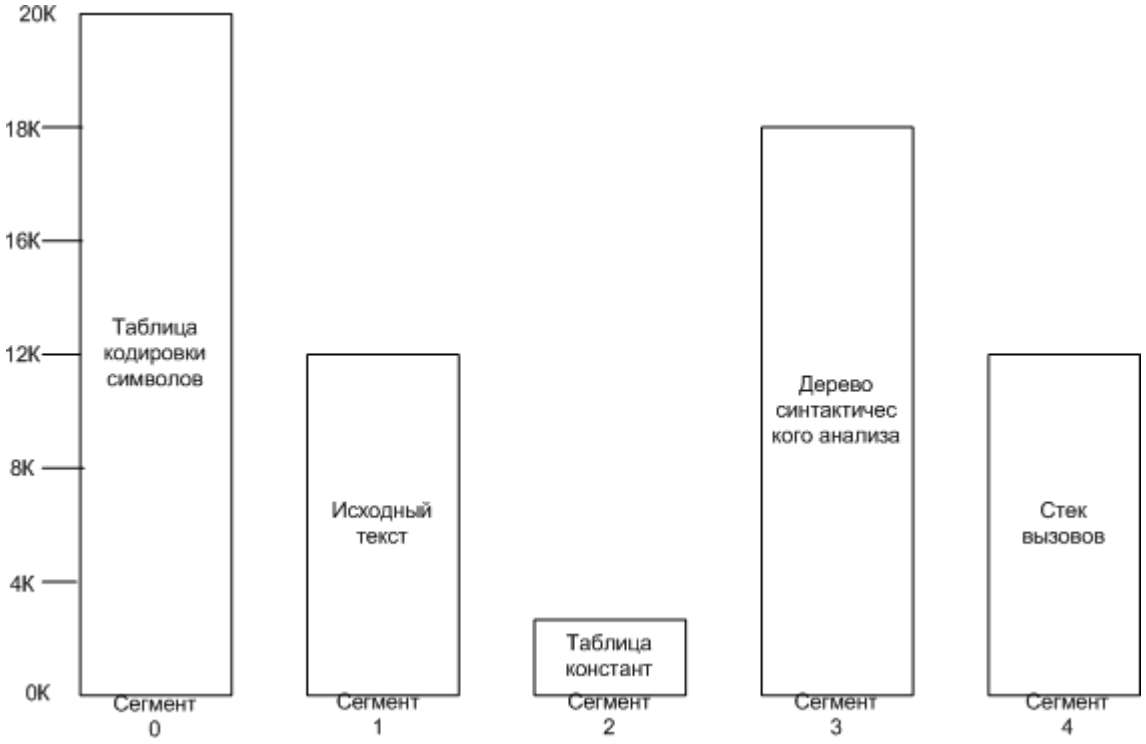

Недостатки такой системы:

1. Один участок может полностью заполниться, но при этом останутся свободные участки. Можно конечно перемещать участки, но это очень сложно.
2. Эти проблемы можно решить, если дать каждому участку независимое адресное пространство, называемое сегментом.[1]

**Сегментированная память**

Каждый сегмент может расти или уменьшаться независимо от других. Сегмент - это логический объект. В этом случае адрес имеет две части:

- номер сегмента
- адрес в сегменте




Преимущества сегментации:

1. Сегменты не мешают друг другу.

2. Начальный адрес процедуры всегда начинается с (n,0). Что упрощает программирование.
3. Облегчает совместное использование процедур и данных.
4. Раздельная защита каждого сегмента (чтение, запись). [2]

Контрольные вопросы
1. Что такое сегментация памяти?
2. Как реализуется аппаратная сегментация?
3. Сегментация без разбиения на страницы?
4. Как работает сегментация с разбиением на страницы?
5. Как работает совместное использование сегментов?

## Основные концепции организации ввода-вывода

План
1. Система ввода-вывода;
2. Устройства ввода-вывода;
3. Менеджмент ввода-вывода;
4. Средства организации ввода-вывода.

Как известно, ввод / вывод считается одной из самых сложных областей проектирования операционных систем, в которой сложно применить общий подход из-за изобилия частных методов. Сложность возникает из-за огромного числа устройств ввода / вывода разнообразной природы, которые должна поддерживать ОС. При этом перед создателями ОС встает очень непростая задача - не только обеспечить эффективное управление устройствами ввода / вывода, но и создать удобный и эффективный виртуальный интерфейс устройств ввода / вывода, позволяющий прикладным программистам просто считывать или сохранять данные, не обращая внимание на специфику устройств и проблемы распределения устройств между выполняющимися задачами. Система ввода / вывода, способная объединить в одной модели широкий набор устройств, должна быть универсальной. Она должна учитывать потребности существующих устройств, от простой мыши до клавиатур, принтеров, графических дисплеев, дисковых накопителей, компакт-дисков и даже сетей. С другой стороны, необходимо обеспечить доступ к устройствам ввода / вывода для множества параллельно выполняющихся задач, причем так, чтобы они как можно меньше мешали друг другу.

Поэтому самым главным является следующий принцип: любые операции по управлению вводом / выводом объявляются привилегированными и могут выполняться только кодом самой ОС. Для обеспечения этого принципа в большинстве процессоров даже вводятся режимы пользователя и супервизора. Как правило, в режиме супервизора выполнение команд ввода / вывода разрешено, а в пользовательском режиме - запрещено. Использование команд ввода / вывода в пользовательском режиме вызывает исключение, и управление через механизм прерываний передаётся коду ОС. Хотя возможны и более сложные системы, в которых в ряде случаев пользовательским программам разрешено непосредственное выполнение команд ввода / вывода.


Еще раз подчеркнем, что, прежде всего, мы говорим о мультипрограммных ОС, для которых существует проблема разделения ресурсов. Одним из основных видов ресурсов являются устройства ввода / вывода и соответствующее программное обеспечение, с помощью которого осуществляется управление обменом данными между внешними устройствами и оперативной памятью. Помимо разделяемых устройств ввода / вывода (эти устройства допускают разделение посредством механизма доступа) существуют неразделяемые устройства. Примерами разделяемого устройства могут служить накопитель на магнитных дисках, устройство для чтения компакт-дисков. Это устройства с прямым доступом. Примеры неразделяемых устройств - принтер, накопитель на магнитных лентах. Это устройства с последовательным доступом. Операционные системы должны управлять и теми и другими устройствами, предоставляя возможность параллельно выполняющимся задачам использовать различные устройства ввода / вывода.

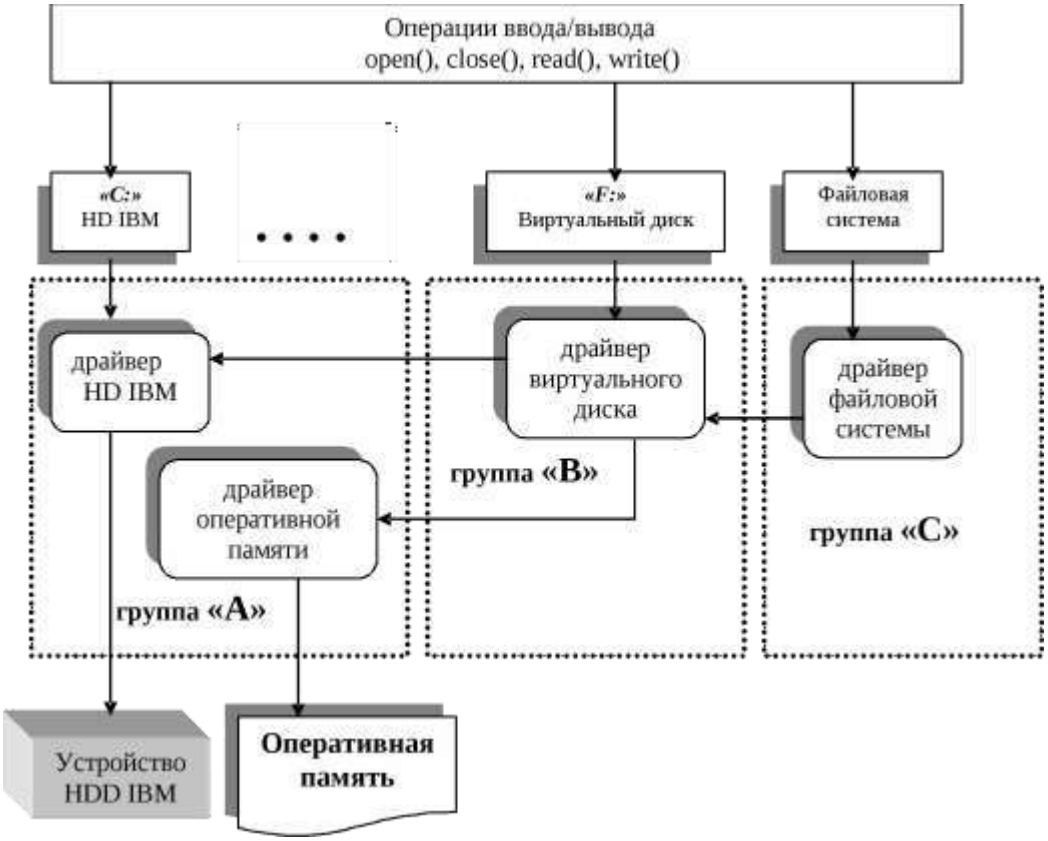
Можно назвать три основные причины, по которым нельзя разрешать каждой отдельной пользовательской программе обращаться к внешним устройствам непосредственно:

Необходимость разрешать возможные конфликты доступа к устройствам ввода / вывода. Например, две параллельно выполняющиеся программы пытаются вывести на печать результаты своей работы. Если не предусмотреть внешнее управление устройством печати, то в результате мы можем получить абсолютно нечитаемый текст, так как каждая программа будет время от времени выводить свои данные, которые будут перемежаться данными другой программы. Другой пример: ситуация, когда одной программе необходимо прочитать данные с некоторого сектора магнитного диска, а другой - записать результаты в другой сектор того же накопителя. Если операции ввода / вывода не будут отслеживаться каким-то третьим (внешним) процессом-арбитром, то после позиционирования магнитной головки для первого запроса может тут же появиться команда позиционирования головки для второй задачи, и обе операции ввода / вывода не смогут быть выполнены корректно.

Желание увеличить эффективность использования этих ресурсов. Например, у накопителя на магнитных дисках время подвода головки чтения/записи к необходимой дорожке и обращение к определенному сектору может значительно (до тысячи раз)

превышать время пересылки данных. В результате, если задачи по очереди обращаются к цилиндрам, далеко отстоящим друг от друга, то полезная работа, выполняемая накопителем, может быть существенно снижена.

Ошибки в программах ввода / вывода могут привести к краху всех вычислительных процессов, ибо часть операций ввода / вывода осуществляется для самой операционной системы. В ряде ОС системный ввод/вывод имеет существенно более высокие привилегии, чем ввод/вывод задач пользователя. Поэтому системный код, управляющий операциями ввода / вывода, очень тщательно отлаживается и оптимизируется для повышения надёжности вычислений и эффективности использования оборудования.




Итак, управление вводом / выводом осуществляется операционной системой, компонентом, который чаще всего называют супервизором ввода / вывода. В перечень основных задач, возлагаемых на супервизор, входят следующие:

супервизор ввода / вывода получает запросы на ввод/вывод от прикладных задач и от программных модулей самой операционной системы. Эти запросы проверяются на корректность, и если запрос выполнен по спецификациям и не содержит ошибок, он обрабатывается дальше, в противном случае пользователю (задаче) выдается соответствующее диагностическое сообщение о недействительности (некорректности) запроса;

супервизор ввода / вывода вызывает соответствующие распределители каналов и контроллеров, планирует ввод/вывод (определяет очерёдность предоставления устройств ввода / вывода задачам, затребовавшим их). Запрос на ввод/вывод либо тут же выполняется, либо ставится в очередь на выполнение;

супервизор ввода / вывода инициирует операции ввода / вывода (передаёт управление соответствующим драйверам) и в случае управления вводом/выводом с

использованием прерываний предоставляет процессор диспетчеру задач с тем, чтобы передать его первой задаче, стоящей в очереди на выполнение;

при получении сигналов прерываний от устройств ввода / вывода супервизор идентифицирует их и передаёт управление соответствующей программе обработки прерывания;

супервизор ввода / вывода осуществляет передачу сообщений об ошибках, если таковые происходят в процессе управления операциями ввода / вывода;

супервизор ввода / вывода посылает сообщения о завершении операции ввода / вывода запросившему эту операцию процессу и снимает его с состояния ожидания ввода / вывода, если процесс ожидал завершения операции.

В случае, если устройство ввода / вывода является инициативным, управление со стороны супервизора ввода / вывода будет заключаться в активизации соответствующего вычислительного процесса (перевод его в состояние готовности к выполнению).

Таким образом, прикладные программы (а в общем случае - все обрабатывающие программы) не могут непосредственно связываться с устройствами ввода / вывода независимо от использования устройств (монопольно или совместно). Установив соответствующие значения параметров в запросе на ввод/вывод, определяющих требуемую операцию и количество потребляемых ресурсов, они могут передать управление супервизору ввода / вывода, который и запускает необходимые логические и физические операции.

Упомянутый выше запрос на ввод/вывод должен удовлетворять требованиям API той операционной системы, в среде которой выполняется приложение. Параметры, указываемые в запросах на ввод/вывод, передаются не только в вызывающих последовательностях, создаваемых по спецификациям API, но и как данные, хранящиеся в соответствующих системных таблицах. Все параметры, которые будут стоять в вызывающей последовательности, поставляются компилятором и отражают требования программиста и постоянные сведения об операционной системе и архитектуре компьютера в целом. Переменные сведения о вычислительной системе (её конфигурация, состав оборудования, состав и особенности системного программного обеспечения) содержатся в специальных системных таблицах. Процессору, каналам прямого доступа в память, контроллерам необходимо передавать конкретную двоичную информацию, с помощью которой и осуществляется управление оборудованием. Эта конкретная двоичная информация в виде кодов и данных часто готовится с помощью препроцессоров, но часть её хранится в системных таблицах.

Контрольные вопросы

1. Зачем нужна система ввода-вывода?
2. Какие устройства ввода-вывода могут находится в ОС?
3. Как осуществляется менеджмент ввода-вывода?
4. Как организуется ввод-вывод?

## Работа ОС с устройствами ввода-вывода

План

1. режим обмена с опросом готовности УВВ;
2. режим обмена с прерываниями;

3. Структура ввода-вывода;
4. Совместное использования устройств.

В области технического обеспечения выделяется несколько основных принципов взаимодействия внешних устройств с вычислительной системой, в основе которого лежит единый интерфейс для их подключения, позволяющий возложить все специфические действия на контроллеры самих устройств. Тем самым конструкторы вычислительных систем переложили головную боль, связанную с подключением внешней аппаратуры, на разработчиков самой аппаратуры, заставляя их придерживаться определенного стандарта.

Похожий подход оказался продуктивным и в области программного подключения устройств ввода-вывода. Достаточно разделить устройства на относительно небольшое число типов, отличающихся по набору операций, которые могут быть ими выполнены, считая все остальные различия несущественными. Затем специфицировать интерфейсы между ядром операционной системы, осуществляющим некоторую общую политику ввода-вывода, и программными частями, непосредственно управляющими устройствами, для каждого из таких типов. При этом разработчики операционных систем получают возможность освободиться от написания и тестирования этих специфических программных частей, получивших название драйверов, передав эту деятельность производителям самих внешних устройств (рис.1).

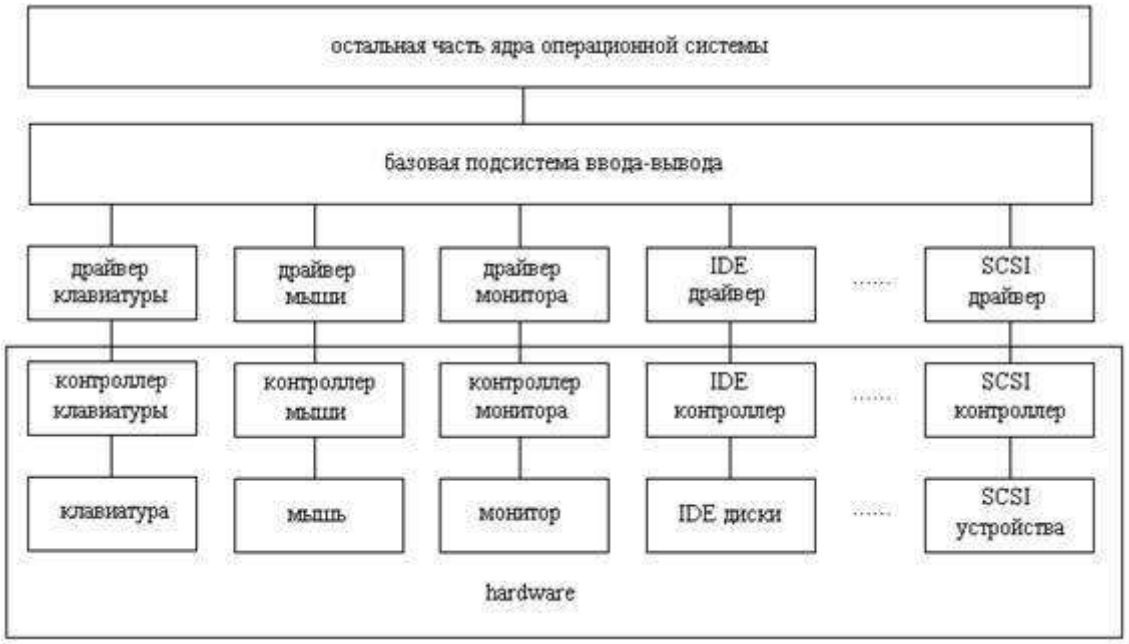



Рис 1. Структура системы ввода-вывода

Два нижних уровня этой системы составляет hardware: сами устройства, непосредственно выполняющие операции, и их контроллеры, служащие для организации совместной работы устройств и остальной вычислительной системы. Следующий уровень составляют драйвера устройств ввода-вывода, скрывающие от разработчиков операционных систем особенности функционирования конкретных приборов и обеспечивающие четко определенный интерфейс между hardware и вышележащим уровнем – уровнем базовой подсистемы ввода-вывода, которая, в свою очередь,

предоставляет механизм взаимодействия между драйверами и программной частью вычислительной системы в целом.

Самым главным является следующий принцип организации управления вводом/выводом: любые операции по управлению вводом/выводом объявляются привилегированными и могут выполняться только кодом операционной системы. Для обеспечения этого принципа в большинстве процессоров вводятся режим пользователя и режим супервизора. В режиме супервизора выполнение команд ввода/вывода разрешено, а пользовательском режиме – запрещено.

В системах мультипрограммирования одним из основных видов ресурсов являются УВВ и соответствующее программное обеспечение, с помощью которого осуществляется управление обменом данными между внешними устройствами и оперативной памятью. Помимо разделяемых устройств ввода/вывода (эти устройства допускают разделение посредством механизма доступа) существуют неразделяемые устройства. Примерами разделяемого устройства являются HDD, CD-ROM – устройства с прямым доступом. К неразделяемым устройствам относятся принтеры – устройства с последовательным доступом.

Управление вводом/выводом осуществляется операционной системой, для чего в ее состав включается супервизор ввода/вывода, основными функциями которого является:

· получение запросов на ввод/вывод от прикладных задач и программных модулей самой ОС. Эти запросы проверяются на корректность, и если запрос выполнен по спецификациям и не содержит ошибок, он обрабатывается дальше, в противном случае пользователю (программе) выдается соответствующее диагностическое сообщение о недействительности (некорректности) запроса;

· вызов соответствующих распределителей каналов и контроллеров, планирование ввода/вывода (определяет очередность предоставления УВВ задачам). Запрос на ввод/вывод либо сразу выполняется, либо ставится в очередь на выполнение;

· инициирование операции ввода/вывода (передает управление соответствующим драйверам) и в случае управления вводом/выводом с использованием прерываний предоставление процессора диспетчеру задач с тем, чтобы передать его первой задаче, стоящей в очереди на выполнение;

· идентификация сигналов прерываний от УВВ и передача управления соответствующей программе обработке прерываний;

· передача сообщений об ошибках, если таковые происходят в процессе управления операциями ввода/вывода;

· посылка сообщения о завершении операции ввода/вывода запросившему эту операцию процессу и снятие его с состояния ожидания ввода/вывода, если процесс ожидал завершения операции ввода/вывода.

Запросы на ввод/вывод должны удовлетворять требованиям API той ОС, в среде которой выполняется приложение.

Имеются два основных режима ввода/вывода:

1. режим обмена с опросом готовности УВВ;

2. режим обмена с прерываниями.

## 2. Режимы управления вводом/выводом. Основные системные таблицы ввода/вывода.

1. Управление вводом/выводом осуществляет центральный процессор. В этом случае имеет место программный канал обмена данными между внешними устройствами и оперативной памятью). ЦП посылает устройству управления команду выполнить некоторое действие устройству ввода/вывода. Последнее исполняет команду, транслируя сигналы, понятные центральному процессору и устройству управления в сигналы, в сигналы понятные УВВ. Однако, быстродействие УВВ намного меньше быстродействия центрального процессора (рис. 2).


Рис. 2.

Поэтому сигнал готовности (транслируемый или генерируемый устройством управления и сигнализирующий процессору о том, что команда ввода/вывода выполнена и можно выдать новую команду для продолжения обмена данными) приходится очень долго ожидать, постоянно опрашивая существующую линию интерфейса на наличие или отсутствие нужного сигнала. Посылать новую команду, не дождавшись сигнала готовности, сообщающего об исполнении предыдущей команды, бессмысленно. До тех пор, пока не появится сигнал готовности, драйвер в цикле опрашивает УВВ, расходуя при этом ресурс процессора.

2. Гораздо выгоднее выдав команду ввода/вывода на время «забыть» об УВВ и перейти на выполнение другой программы. А появление сигнала готовности трактовать как запрос на прерывание от УВВ.

Режим обмена с прерываниями по своей сути является режимом асинхронного управления. Для того чтобы не потерять связь с устройством (после того как процессор выдал очередную команду по управлению обменом данными и переключился на выполнение других программ), может быть запущен отсчет времени, в течение которого устройство должно обязательно выполнить команду и выдать сигнал запроса на прерывание. Максимальный интервал времени, в течение УВВ или его контроллер должны выдать сигнал запроса на прерывание, называют установкой тайм-аута. Если это время истекло после выдачи устройству очередной команды, а устройство так и не ответило, то делается вывод о том, что связь с устройством потеряна и управлять им больше нет возможности. Пользователь и/или задача получают соответствующее диагностическое сообщение.

Многие устройства не допускают совместного использования. Такие устройства могут быть закрепленными, т.е. предоставленными некоторому вычислительному процессу. При этом вычислительные процессы не могут выполняться параллельно, т.к. они ожидают освобождения устройств ввода/вывода. Для организации использования многим параллельно выполняющимися задачами устройств ввода/вывода, которые не

могут быть разделяемыми, вводится понятие виртуальных устройств, позволяющие повысить эффективность вычислительной системы.

Понятие виртуального устройства основывается на понятии SPOOLing (simultaneous peripheral operation on-line – имитация работы с устройством в режиме «он-лайн»). Главная задача спулинга создать видимость параллельного разделение УВВ с последовательным доступом, которое фактически должно использоваться только монопольно. Например, каждому вычислительному процессу можно предоставить не реальный, а виртуальный принтер и поток выводимых символов сначала направлять в специальный файл. Затем, по окончании виртуальной печати, в соответствии с принятой дисциплиной обслуживания и приоритетами приложений выводить содержимое спул-файла на принтер. Системный процесс, который управляет спул-файлом, называется спулером.

Контрольные вопросы

1. Как работает режим обмена с опросом готовности УВВ?
2. Как работает режим обмена с прерываниями?
3. Что из себя представляет собой структура ввода-вывода?
4. Как реализуется совместное использования устройств?

## Логическая и физическая организация файловой системы

План

1. Логическая организация ФС;
2. Физическая организация ФС;
3. Таблицы размещения файлов;
5. Нежурналируемые и журналируемые ФС.

*Файловая* система (ФС) — это часть операционной системы, включающая совокупность всех файлов на диске, служебные структуры, включая каталоги, системные программные средства.

*Файл* — это именованная область внешней памяти, в которую можно записывать и из которой можно считывать данные. Файловые системы поддерживают функционально различные типы файлов, напри-

- обычные файлы (ОС не контролирует содержимое этих файлов);
- каталоги (содержат системную информацию о наборе файлов);
- специальные файлы (фиктивные файлы, соответствующие устройствам ввода-вывода);
- отображаемые в память файлы и т. д.

**Иерархическая структура файловой системы.** Большинство файловых систем имеют иерархическую структуру. Структура может быть организованна как *дерево* (рис. 12, *а)* или как *сеть* (Unix) (рис. 12, б).

В древовидной структуре действует принцип: *один файл — одно полное имя.* В сетевой: *один файл — много полных имен* (за счет наличия уникального цифрового имени).

**Атрибуты файлов.** Понятие *файл* включает в себя не только данные, но и атрибуты. *Атрибуты* — это информация, описывающая свойства файла (тип, владелец,

пароль, информация для авторизации доступа, время создания и доступа, размер, признаки и пр.)

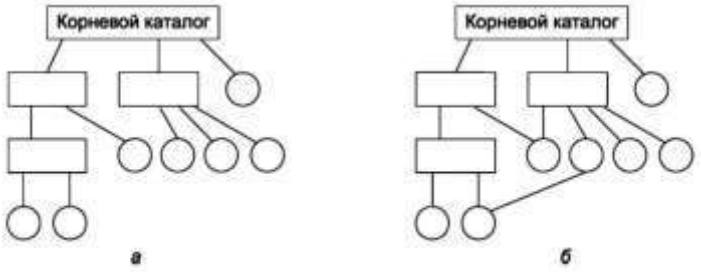



**Рис. 12.** Структура файловых систем: *а* — иерархическая; *б* — сетевая

Набор атрибутов зависит от ОС. Значения атрибутов могут храниться в каталогах (FAT) или в специальных таблицах, ссылки на которые хранятся в каталогах (NTFS).

**Физическая организация файловой системы.** Представление пользователя о ФС и физическое хранение файлов на диске имеют мало общего. Диск в общем случае состоит из пакета пластин. На каждой пластине — две ***поверхности***. На каждой поверхности размечены ***дорожки***, на которых хранятся данные. Дорожки нумеруются с нуля, начиная от края к середине. Для каждой поверхности пластины имеется магнитная ***головка***, которая, переме-щаясь, может позиционироваться над каждой дорожкой. Все головки закреплены на одном механизме и перемещаются синхронно. Дорожки одного радиуса на всех поверхностях называются ***цилиндром***. Каждая дорожка делится на фрагменты, называемые ***секторами***. Чаще размер сектора равен 512 байтам. Сектор — наименьшая адресуемая единица обмена. Для поиска контроллер должен задать: номер цилиндра, поверхности и сектора.

ОС для работы с диском использует собственную единицу — ***кластер***, или ***блок***. Раз-мер кластера чаще всего равен 1024 байта. Увеличение объемов дисковых накопителей, а также размеров обрабатываемых файлов обусловило переход от понятия «кластер», понимаемого как минимальный объем единовременного обмена данными в ПК (запись/чтение/ передача), к более общему понятию «***порция***», или ***«отрезок»***. В ФС FAT32 ***порцией*** *назы-вается участок дискового пространства, состоящий из одного или нескольких кластеров (блоков), содержащих непрерывную часть данных файла.*

Дорожки и секторы создаются в результате низкоуровневого форматирования диска и не зависят от типа ОС. Диск может быть разделен на логические устройства — ***разделы*** (***тома***): a, b, c, ... Разметку раздела под конкретный тип файловой системы выполняют процедуры логического форматирования. При этом определяется размер кластера и записывается информация о границах файлов и каталогов, поврежденных областях, о доступном пространстве. Также записывается ***загрузчик ОС***. В одном разделе может быть создана только одна ФС, но любого доступного типа (FAT16, FAT32, NTFS и пр.).

На этапе разбиения диска на разделы в блоке данных первого физического сектора дис-ка (0 цилиндр, 0 поверхность, 1 сектор) с адреса 1BEh формируется таблица разделов (Partition table), состоящая из 4-х шестнадцатибайтных строк. Обычно системную информацию, записанную в блок данных этого сектора в процессе форматирования, называют ***Master Boot Record (MBR)***.

**Физическая организация FAT.**

На этапе логического форматирования раздела (тома) диска под ФС FAT создаются четыре логических области:

• ***загрузочный сектор (boot sector),*** в который помещается программа загрузки ОС;

• ***таблица FAT***, которая является таблицей размещения файлов (создаются её основная и резервная копии FAT1 и FAT2);

• ***корневой каталог*** — содержит 256 записей (для FAT16), или 65 535 записей (для FAT32) по 32 байта каждая (для FAT32 он может находиться в любом месте диска);

• ***область данных*** — кластеры размером от 1 до 128 секторов.

Разрядность элемента таблицы FAT равна 16 бит (2 байта) для ФС FAT16 и 32 бита (4 байта) для ФС FAT32. Она определяет количество кластеров, к которым может физически адресоваться ФС. В области данных для FAT16 это 65525 кластеров. Для FAT32 это 288 435 445 кластеров.

Элемент в таблице FAT может принимать следующие значения: кластер свободен, за-нят и не последний, занят и последний, дефектный, резервный.

При запуске компьютера программа, запускаемая из главной загрузочной записи (*MBR*), ищет активный раздел на жестком диске. Если активный раздел найден, то управле-ние передаётся программе находящейся в загрузочном секторе (*boot sector*) этого раздела, которая запускает операционную систему, установленную в этом разделе жесткого диска.

При необходимости обращения к какому-либо файлу, адрес начального кластера корне-вого каталога раздела считывается из загрузочного сектора раздела. Часть корневого каталога, относящаяся к искомому файлу, отыскивается в корневом каталоге по имени файла. Каждый элемент корневого каталога содержит для каждого файла, расположенного на диске: имя файла, адрес первого кластера файла, размер файла, дату и время создания, модификации, последнего доступа к файлу и номер элемента таблицы FAT, содержащего адрес второго кластера файла. Каждый элемент таблицы FAT содержит (указывает) номер следующего кластера файла. Поэтому элемент таблицы FAT называется ***указателем***. Одновременно, этот номер является номером элемента таблицы FAT, который содержит адрес третьего кластера файла и т. д. до последнего кластера файла.

**Существуют три основных способа физического размещения файлов на диске[2]:**

• способ непрерывного размещения;

• способ связанных блоков (цепочечный);

• способ индексированных блоков.

Рассматривая ранее физическую организацию ФС FAT, мы говорили о том, что в каж-дом элементе таблицы FAT хранится адрес (номер) кластера файла. Так было, когда объёмы жестких дисков были сравнительно не велики. Сейчас объёмы жестких дисков существенно выросли и достигают нескольких терабайт. Поэтому для реальных больших жестких дисков в каждом элементе таблицы FAT хранится не адрес отдельного кластера. Содержимое элемента таблицы FAT зависит от способа физического размещения файла на диске.

*1. Способ непрерывного размещения (рис. 13а, 13б).*

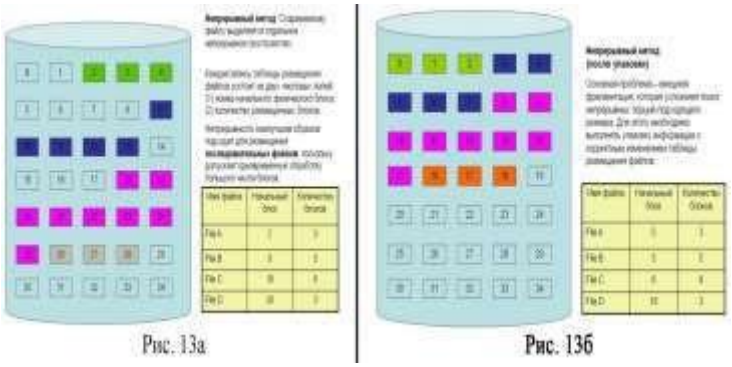


Рис. 13а Рис. 13б

В элементе таблицы FAT содержится адрес первого кластера порции данных на диске и количества кластеров в порции. Количество кластеров в порции в данном случае равно количеству кластеров в файле. Этот способ используется для хранения коротких файлов.

2. *Способ связанных блоков (цепочечный). См. рис. 13в, 13г.* Для хранения больших файлов.

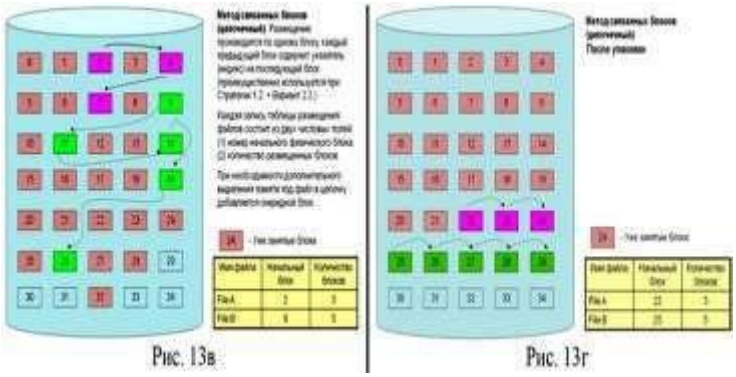


Рис. 13в Рис. 13г

В элементе таблицы FAT содержится адрес первого кластера порции данных файла на жестком диске, количество блоков в порции и номер элемента таблицы FAT, содержащего адрес (номер) первого кластера (блока) следующей порции данных файла. Каждый кластер в порции данных содержит внутри себя ***индекс*** – указатель на последующий блок (кластер) внутри порции. На рис. 13в, 13 г для простоты изображен случай, когда количество кластеров в файле не превышает количества кластеров в порции.

3. *Способ индексированных блоков (рис. 13д, рис. 13е)*. Для хранения сверхбольших файлов.

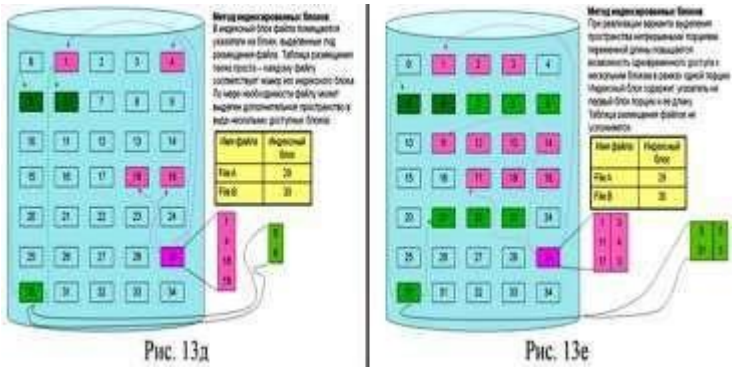


Рис. 13д Рис. 13е

В элементе таблицы FAT содержится адрес первого индексного блока порции данных файла на жестком диске, номер элемента таблицы FAT, содержащего адрес индексного блока следующей порции данных файла (в случае больших файлов, для которых количество указателей на блоки одной порции столь велико, что не помещается в

один индексный блок (кластер)). На рис. 13в, 13 г количество кластеров в файле не превышает количества класте-ров в порции.

**Физическая организация NTFS.**

Файловая система NTFS была разработана в качестве основной для Windows NT. Ее особенностями являются поддержка больших файлов и дисков (до 264 Гбайт), восстанавливаемость после сбоев, низкий уровень фрагментации.

Непрерывная область кластеров в NTFS называется ***отрезком***. Порядковый номер кластера тома называется ***логическим номером кластера*** (***Logical Cluster Number - LCN***). Порядковый номер кластера внутри файла называется ***виртуальным номером кластера*** (***Virtual Cluster Number - VCN***).

Часть файла в отрезке характеризуется числом, образованным значениями трех параметров: LCN, VCN, К, где ***К — длина отрезка***. Для хранения номера кластера используются 64-разрядные указатели. Весь том (раздел) — это последовательность кластеров.

**Файловая система NTFS** представляет собой один и более файлов. Каталог тоже файл. Основа структуры тома NTFS – главная таблица файлов ***MFT (Master File Table).*** Это тоже файл. MFT содержит, по крайней мере, одну запись для каждого файла тома, включая одну запись для самой себя. MFT состоит из записей, размер которых равен размеру кластера и зависит от размера тома -1, 2, или 4 Кбайта. *По умолчанию 2 Кбайта*. Порядковый номер записи в MFT является номером файла в томе. Изначально под зону MFT отводится 12,5% объема тома NTFS.

Структура тома NTFS показана на рисунке 14. Загрузочный блок тома NTFS располагается в начале тома, а его копия - в середине области данных тома. Загрузочный блок содержит стандартный блок параметров BIOS, количество блоков в томе, а также начальные логические номера кластеров основной копии MFT и зеркальной копии MFT.

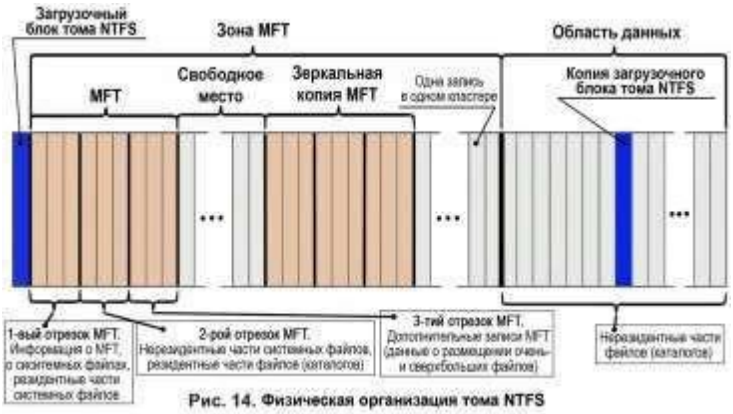


Рис. 14. Физическая организация тома NTFS

В NTFS файл целиком размещается в записи таблицы MFT, если это позволяет сделать его размер. В том же случае, когда размер файла больше размера записи MFT, в запись помещаются только некоторые ***атрибуты файла***, а остальная часть файла размещается в **отдельном отрезке тома (или нескольких отрезках)**. ***Часть файла, размещаемая в записи MFT,*** *называется* ***резидентной частью,*** *а остальные части -* ***нерезидентными.*** Адресная информация об отрезках, содержащих нерезидентные части файла, размещается в атрибутах резидентной части.

**MFT делится на несколько отрезков.**

В первый отрезок помещаются 16 стандартных, создаваемых при форматировании записей о системных файлах NTFS и системные файлы NTFS. Некоторые системные файлы являются полностью резидентными, а некоторые имеют и нерезидентные части,

которые располагаются после первого отрезка MFT. Нулевая запись MFT содержит описание самой MFT, в том числе и такой ее важный атрибут, как адреса всех ее отрезков.

Во второй отрезок помещаются нерезидентные части системных файлов и резидентные части обычных файлов и каталогов.

В третьем отрезке находятся дополнительные записи MFT, используемые при размещении очень больших и сверхбольших файлов.

Из приведенного описания видно, что сама таблица MFT рассматривается как файл, к которому применим метод размещения в томе в виде набора произвольно расположенных нескольких отрезков.

Каждый файл и каталог в томе NTFS состоит из набора двенадцати системных атрибутов, определяемых файловой системой NTFS и неопределенного количества атрибутов, которые для каждого файла или каталога могут быть определены пользователем. Важно отметить, что ***имя файла и его данные*** также ***рассматриваются как атрибуты файла.*** То есть в трактовке NTFS, кроме атрибутов у файла нет никаких других компонентов.

Существуют два способа хранения атрибутов файла - резидентное хранение в записях таблицы MFT и нерезидентное хранение вне ее, во внешних отрезках. Таким образом, ***резидентная часть файла состоит из резидентных атрибутов, а нерезидентная - из нерезидентных атрибутов.*** *Сортировка может осуществляться только по резидентным атрибутам.*

Основными системными атрибутами всех файлов являются:

− имя файла (FN — file name) - этот атрибут содержит длинное имя файла в формате Unicode, а также номер входа в таблице MFT для родительского каталога; если этот файл содержится в нескольких каталогах, то у него будет несколько атрибутов типа File Name; этот атрибут файла всегда резидентный;

− данные (Data) - содержит обычные данные файла;

− дескриптор безопасности (SD — security descriptor) - этот атрибут содержит информацию о защите файла: список прав доступа ACL и поле аудита, которое определяет, какого рода операции над этим файлом нужно регистрировать;

+- список атрибутов (AL - attribute list) - список атрибутов, из которых состоит файл; содержит ссылки на номер записи MFT, где расположен каждый атрибут; этот редко используемый атрибут нужен только в том случае, если атрибуты файла не умещаются в основной записи и занимают дополнительные записи MFT;

− стандартная информация (SI — standard information) - этот атрибут хранит всю остальную стандартную информацию о файле, которую трудно связать с каким-либо другим атрибутом файла, например, время создания файла, время обновления и другие.

**Файлы NTFS** в зависимости от размера делятся на небольшие, большие, очень боль-шие и сверхбольшие. Каждый тип файла имеет свои особенности размещения в томе NTFS.

Контрольные вопросы

1. Что такое логическая организация ФС;
2. Что такое физическая организация ФС;
3. Как работают таблицы размещения файлов;
5. Назовиет отличия нежурналируемых и журналируемых ФС.

# Программное обеспечение таймеров

План

1. Служба времени в ПК и ОС;
2. Программное обеспечение таймеров;
3. Функции управления таймерами

Функциональность, связанная с использованием и контролем времени, скорее всего, будет доступна в ОС реального времени. Возможности будут отличаться в зависимости от ОСРВ, но мы рассмотрим общедоступные. В любом случае таймер реального времени — это обязательный элемент для функционирования любого из этих сервисов.

## Системное время

Простое системное время, или «тактовый таймер», доступно почти всегда. Это просто счетчик (как правило, 32 бита), который увеличивается с помощью процедуры обслуживания прерываний в режиме реального времени и может устанавливаться и считываться через вызовы API.

## Тайм-ауты вызова служб

Обычно ОСРВ разрешает блокирующие вызовы API, то есть вызывающая задача приостанавливается (блокируется), пока запрашиваемый сервис не будет предоставлен. Обычно эта блокировка является неопределенной, но некоторые ОСРВ предлагают тайм-аут, за время выполнения которого вызов возвращается, когда истекает время ожидания, если сервис продолжает оставаться недоступным. Таймауты вызовов API поддерживаются не всеми ОСРВ.

## Состояние сна задачи

Обычно задачи имеют возможность приостанавливать себя на фиксированный период времени. Это обсуждалось ранее в разделе «Управление задачами».

## Программные таймеры

Чтобы программные задачи выполняли функции отсчета времени, большинство ОСРВ предлагают объекты таймера. Это независимые таймеры, обновляемые обработчиком прерываний таймера реального времени, которые могут контролироваться вызовами API. Такие вызовы настраивают, контролируют и отслеживают работу таймера. Как правило, они могут быть установлены для однократного срабатывания или автоматического перезапуска. Также обычно поддерживается подпрограмма истечения срока действия, функция, которая выполняется каждый раз, когда таймер завершает цикл. В следующей статье будет больше информации о программных таймерах и описание их реализации.

Использование таймеров

Программные таймеры могут быть настроены на однократное срабатывание, то есть они запускаются, а затем, после указанного периода времени, просто завершают цикл. Либо таймер может быть настроен на перезапуск: после завершения счета таймер автоматически перезапускается. Время работы после перезапуска может отличаться от первичного времени работы. Кроме того, таймер можно опционально настроить на выполнение специальной функции завершения, которая выполняется когда (или каждый раз когда) таймер завершает рабочий цикл.

### Настройка таймеров

Количество таймеров

Как и для большинства аспектов Nucleus SE, настройка таймеров управляется директивами **#define** в **nuse_config.h**. Основным параметром является **NUSE_TIMER_NUMBER**, который определяет сконфигурированные в приложении таймеров. По умолчанию это значение равно нулю (то есть в приложении таймеры не используются), и может принимать значения вплоть до 16. Некорректное значение приведет к ошибке компиляции, которая будет сгенерирована проверкой в файле **nuse_config_check.h** (этот файл входит в **nuse_config.c** и компилируется вместе с ним), что приведет к срабатыванию директивы **#error**.

Выбор ненулевого значения является главным активатором таймеров. Этот параметр используется при определении структур данных, и от его значения зависит их размер. Кроме того, ненулевое значение активирует настройки API.

### Активация функции завершения

В Nucleus SE я пытался найти возможность сделать функционал опциональным, там, где это позволит сэкономить память. Хорошим примером является поддержка функций завершения таймеров. Помимо того, что эта возможность опциональна для каждого таймера, механизм может быть активирован (или нет) для всего приложения при помощи параметра **NUSE_TIMER_EXPIRATION_ROUTINE_SUPPORT** в **nuse_config.h**. Присвоение этому параметру значения **FALSE** блокирует определение двух структур данных в ПЗУ, которые будут подробно описаны в этой статье.

### Активация API

Каждая функция API (служебный вызов) в Nucleus SE имеет активирующую директиву #define в nuse_config.h. Для таймеров к таким символам относятся:
**NUSE_TIMER_CONTROL**
**NUSE_TIMER_GET_REMAINING**
**NUSE_TIMER_RESET**
**NUSE_TIMER_INFORMATION**
**NUSE_TIMER_COUNT**

По умолчанию, всем активаторам присвоено значение **FALSE**, таким образом, все служебные вызовы отключены, блокируя включение реализующего их кода. Для настройки таймеров в приложении нужно выбрать необходимые служебные вызовы API и присвоить им значение **TRUE**

### Службы таймеров

Фундаментальные операции, которые можно выполнять с таймером, это управление (запуск и остановка) и считывание текущего значения. Nucleus RTOS и Nucleus SE предоставляют два базовых служебных вызова API для этих операций.

### Управление таймером

Служебный вызов Nucleus RTOS API для управления таймером позволяет активировать и деактивировать таймер (запускать и останавливать). Nucleus SE предоставляет аналогичный функционал.

***Вызов для управления таймером в Nucleus RTOS***

Прототип служебного вызова:

**STATUS NU_Control_Timer (NU_TIMER *timer, OPTION enable);**

### Считывание таймера

Для получения оставшегося времени таймера служебный вызов Nucleus RTOS API возвращает количество тактов до истечения срока его действия. Nucleus SE предоставляет аналогичный функционал.

***Вызов для получения оставшегося времени в Nucleus RTOS***

Прототип служебного вызова:

**STATUS NU_Get_Remaining_Time (NU_TIMER *timer, UNSIGNED *remaining_time);**

Параметры:

**timer** – указатель на предоставленный пользователем блок управления таймером;
**remaining_time** – указатель на хранилище значения оставшегося времени, которое является переменной типа **UNSIGNED**.

Возвращаемое значение

**NU_SUCCESS** – вызов был успешно завершен;
**NU_INVALID_TIMER** – некорректный указатель на таймер.

***Вызов для получения оставшегося времени в Nucleus SE***

Этот вызов API поддерживает полный функционал Nucleus RTOS API.

### Сброс таймера

Этот вызов API сбрасывает таймер в исходное, неиспользуемое состояние. Таймер может быть активирован или деактивирован после завершения этого вызова. Его можно использовать только после того, как таймер был отключен (при помощи **NUSE_Timer_Control()**). В следующий раз когда таймер будет активирован, он будет инициализирован с параметром **NUSE_Timer_Initial_Time[]**. Nucleus RTOS позволяет предоставить новое исходное состояние и провести перепланирование времени, а также указать функцию завершения при сбросе таймера. В Nucleus SE эти значения устанавливаются во время настройки и не могут быть изменены, поскольку они хранятся в ПЗУ.

Контрольные вопросы

1. Как организована служба времени в ПК и ОС?
2. Что такое программное обеспечение таймеров?
3. Опишите основные функции управления таймерами.

## Основные понятия безопасности

План

1. Типы угроз в ОС;
2. Критерии безопасной ОС
3 Основные принципы обеспечения безопасности.

Безопасная система должна обладать свойствами конфиденциальности, доступности и целостности. Любое потенциальное действие, которое направлено на нарушение конфиденциальности, целостности и доступности информации, называется угрозой. Реализованная угроза называется атакой.

*Конфиденциальная* (confidentiality) система обеспечивает уверенность в том, что секретные данные будут доступны только тем пользователям, которым этот доступ разрешен (такие пользователи называются авторизованными).
Под *доступностью* (availability) понимают гарантию того, что авторизованным пользователям всегда будет доступна информация, которая им необходима. И наконец, *целостность* (integrity) системы подразумевает, что неавторизованные пользователи не могут каким-либо образом модифицировать данные.

Умышленные угрозы подразделяются на активные и пассивные. Пассивная угроза – несанкционированный доступ к информации без изменения состояния системы, активная – несанкционированное изменение системы. Пассивные атаки труднее выявить, так как они не влекут за собой никаких изменений данных. Защита против пассивных атак базируется на средствах их предотвращения.

Типы угроз:

1. Проникновение в систему под видом легального пользователя.
2. Нежелательные действия легальных пользователей.
3. Функционирование вирусов и прочих червей.

Базовые принципы проектирования системы безопасности ОС (по Зальтцер (Saltzer) и Шредер (Schroeder)):

· Проектирование системы должно быть открытым. Нарушитель и так все знает (криптографические алгоритмы открыты). Не должно быть доступа по умолчанию. Ошибки с отклонением легитимного доступа будут обнаружены скорее, чем ошибки там, где разрешен неавторизованный доступ.

· Нужно тщательно проверять текущее авторство. Так, многие системы проверяют привилегии доступа при открытии файла и не делают этого после. В результате пользователь может открыть файл и держать его открытым в течение недели и иметь к нему доступ, хотя владелец уже сменил защиту.

· Давать каждому процессу минимум возможных привилегий.

· Защитные механизмы должны быть просты, постоянны и встроены в нижний слой системы, это не аддитивные добавки (известно много неудачных попыток "улучшения" защиты слабо приспособленной для этого ОС MS-DOS).

· Важна физиологическая приемлемость. Если пользователь видит, что защита требует слишком больших усилий, он от нее откажется. Ущерб от атаки и затраты на ее предотвращение должны быть сбалансированы.

Внешний урованеь защиты ОС:

· возможность для администратора настраивать доступ к тем или иным ресурсам пользователей и групп пользователей;

· настройка параметров учетных записей, парметров авторизации;

· настройка пользователями прав доступа к соответствующим данным.

Глубинный уровень безопасности:

· шифрование в системе данных, подлежащих защите;

· разделение физической и виртуальной памяти для предотвращения доступа к данным не имеющими на это права пользователями и приложениями.

Функции безопасности Windows NT:

· Информация о доменных правилах безопасности и учетная информация хранятся в каталоге Active Directory.

· В Active Directory поддерживается иерархичное пространство имен пользователей, групп и учетных записей машин (учетные записи могут быть сгруппированы по организационным единицам).

· Административные права на создание и управление группами учетных записей пользователей могут быть делегированы на уровень организационных единиц (возможно установление дифференцированных прав доступа к отдельным свойствам пользовательских объектов).

Контрольные вопросы

1. Назовите и охарактеризуйте основные типы угроз в ОС;

2. Опишите критерии безопасной ОС

3 Опишите основные принципы обеспечения безопасности в ОС.