БИЛЕТ 1 Поколения компьютеров

1.1. Первое поколение компьютеров.

В них использовались электронно-вакуумные лампы.

Конец 40-х-начало 50-х годов 20 века. **ENIAC** (1946 год, Пенсильванский университет США). Для решения задач с ядерным оружием. Компьютер состоял из оперативной памяти, процессора, устройства печати данных на узкой ленте. Эти машины использовались в **однопользовательском режиме**. Программа и данные вводились в оперативную память, потом запускались. Пользователь вводил в память машины команды. Результаты появлялись на печати. В случае ошибки машина останавливалась. На лампочках загоралась ошибка. Средство программирования – машинный язык (машинный код). Пользователь должен был знать эти коды и программировать на них. В таких машинах частота аппаратных сбоев была достаточно высокой. Позднее аппаратный загрузчик упрощал ввод данных в оперативную память.

Первые решения в области операций программирования: появление ассемблеров и автокодов. Появление служебных программ, которые переводили программы с языка Ассемблер на язык машинного кода.

- зарождение класса сервисных, управляющих программ
- Зарождение языков программирования
- Однопользовательский, персональный режим

1.2. Второе поколение компьютеров.

Компьютеры второго поколения построены на полупроводниковых приборах: диодах и транзисторах. (конец 50-х вторая половина 60) Они потребляют меньше электроэнергии и являются более производительными: выполняют несколько миллионов операций в секунду. Создаются новые внешние устройства, совершенствуется программное обеспечение.

Пакетная обработка программ:

Формируется пакет программ. Это были программы для последовательного выполнения. Этот пакет мог представляться как стопка перфокарт, данные на магнитной ленте, перфоленты. Наличие специальной управляющей программы, позволяло координировать обработку заданий из пакета и определять момент, когда программа могла начать выполняться.

В случае ошибки управление также передавалось на управляющую программу. Такую систему называли мониторной.

Следующий этап – появление компьютеров, в которых поддерживался режим **мультипрограммирования.** На обработке находилось сразу несколько программ. Центральным процессором выполнялась одна программа, другие ожидали или занимались обменом. С появлением таких компьютеров появляются операционные системы.

БЭСМ-6: развита система управляющих программ. Автор математического обеспечения – Королев Л.Н. Архитектура этих машин – предел развития машин второго поколения. Расширился спектр задач, для решения которых использовались компьютеры.

Языки управления заданиями.

Появляется проблема дружественности программных объектов (интерфейсов). Командные языки упростили работу пользователя с системой.

Развиваются средства программирования, доступные для пользователя. Появляются языки высокого уровня. Начинается борьба за аппаратную независимость команд. Появилась новая задача: упростить процесс программирования посредством использования языков высокого уровня.

Появляются проблемно-ориентированные языки программирования.

Появились **первые прообразы файловых систем.** Нужно было хранить данные вне оперативной памяти. Появление файловых систем упростило процесс организации и хранения данных на внешних устройствах. Появилось понятие именованного набора данных - абстракция пользователя от внешних устройств (виртуальные устройства).

- пакетная обработка заданий
- мультипрограммирование
- языки управления заданиями
- файловые системы
- виртуальные устройства

1.3. Третье поколение – компьютеры на интегральных схемах.

Элементная база третьего поколения компьютеров (конец 60-х – начало 70-х годов 20 века) – интегральные схемы. Такие устройства потребляли меньше электроэнергии. Характерны более компактные размеры вычислительной техники. Основная особенность – **унификация программных и аппаратных узлов и устройств**. Как следствие такой унификации – появление **семейств компьютеров**. Появляется возможность аппаратной модификации компьютеров, а также унификация программных интерфейсов.

Программа, работающая на младшей модели, должна была работать и на старшей модели. Появление семейств компьютеров создало возможность увеличения сроков жизни программных систем.

В операционной системе появляются **драйверы устройств**. Появляются правила разработки драйверов, а также внесение в систему новых драйверов.

Развитие получили концепции виртуальных устройств, управляющие программы которых представляли унифицированный интерфейс.

1.4. Компьютеры четвертого поколения и далее.

Их элементная база – большие и сверхбольшие интегральные схемы. Устройства – законченные функциональные узлы компьютера. Микропроцессор – реализация функционального узла компьютера.

Происходит революция в области размеров компьютера, “дружественность” пользовательских интерфейсов, развитие и массовое использование сетей ЭВМ.

Появились новые функции в операционной системе: проблемы безопасности хранения и передачи данных. Массово формируются многопроцессорные системы. Параллельные системы становятся массовыми.

- «дружественность» пользовательских интерфейсов
- сетевые технологии
- безопасность хранения и передачи данных

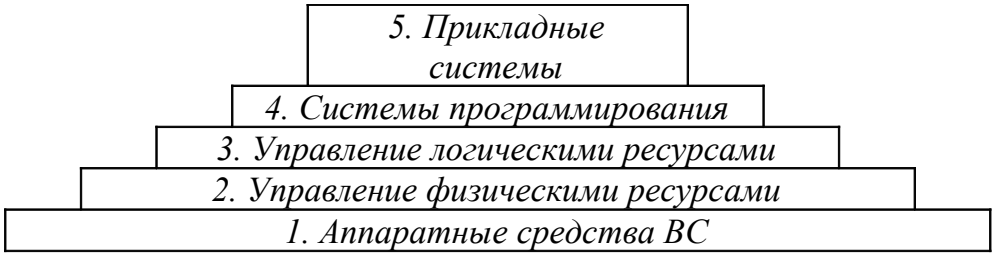
Билет №2 Структура Вычислительной системы. Ресурсы ВС – физические ресурсы, виртуальные ресурсы. Уровень операционной системы.

Вычислительная система (ВС) – совокупность аппаратных и программных средств, функционирующих в единой системе и предназначенных для решения задач определенного класса.

Вычислительная система – это результат интеграции.

Вычислительная система ориентирована для решения задач определенного класса.

Структура вычислительной системы:



Аппаратный уровень вычислительной системы

Определяется наборами аппаратных компонентов и их характеристиками, используемыми вышестоящими уровнями иерархии и определяющими воздействие на них.

Физические ресурсы: процессор, оперативная память, внешнее устройство.

Характеристики:

1. Правила программного использования, которые определяют возможность корректного использования в программе.
2. Производительность или емкость: тактовая частота, длина обрабатываемого машинного слова.
3. Степень занятости или используемости данного физического ресурса.

Нет единого правила формирования этих характеристик. Мы можем определить, какие компоненты соответствуют данному физическому ресурсу.

Средства программирования, доступные на аппаратном уровне:

1. Система команд компьютера.
2. Аппаратные интерфейсы программного взаимодействия с физическими ресурсами.

2.2. Управление физическими ресурсами

Назначение – систематизация и стандартизация правил программного использования физических ресурсов.

Драйвер физического устройства – программа, основанная на использовании команд управления конкретного физического устройства и предназначенная для организации работы с данным устройством.

Появились специализированные устройства – драйверы физических устройств.

Предоставление унифицированного интерфейса для программного использования.

Драйвер физического устройства решал задачи:

1. Сокрытие от пользователя некоторых нюансов.
2. Предоставление упрощенного интерфейса для упрощенного доступа к данному физическому ресурсу.

С помощью драйвера мы можем читать и писать поблочно. Драйвер позволяет работать с записями определенной длины. Два драйвера: один обеспечивает блочный обмен, другой – работу с записями произвольной длины.

В программном обеспечении появились драйверы, которые достаточно хорошо были отлажены. Надежность программного обеспечения повысилась.

Упрощенные интерфейсы привели к преобразованию программы для работы с другим драйвером. Программист должен был быть знаком со всеми интерфейсами и драйверами физических устройств. Возникла проблема: программа и пользователь должны были модифицироваться каждый раз при смене устройств. Для решения этих проблем появляется следующий уровень:

2.3. Управление логическими/виртуальными ресурсами.

Логическое/виртуальное устройство (ресурс) – устройство, некоторые эксплуатационные характеристики которого реализованы программным образом.

Драйвер логического/виртуального ресурса – программа, обеспечивающая существование и использование соответствующего ресурса.

Этот уровень ориентирован на пользователя. Команды данного уровня не зависят от физических устройств, они обращены к предыдущему уровню. На базе этого уровня могут создаваться новые логические ресурсы.

При организации драйвера могут использоваться драйвера физических или логических/виртуальных устройств.

Система поддерживает иерархию драйверов. Многоуровневая унификация интерфейса.

Ресурсы вычислительной системы – совокупность всех физических и виртуальных ресурсов. Одной из характеристик ресурсов является их конечность, следовательно, возникает конкуренция за обладание ресурсом между его программными потребителями.

Средства программирования, доступные на уровнях управления ресурсами ВС:

- система команд компьютера
-
- программные интерфейсы драйверов устройств (как физических, так и виртуальных
-

Операционная система – это комплекс программ, обеспечивающий управление ресурсами вычислительной системы. Пользователю же доступна система команд.

Разветвленная иерархия виртуальных и физических устройств.

Драйверы можно разделить на 3 группы:

1) драйверы физических устройств

2) драйверы устройств аппаратного типа
3)драйверы виртуальных устройств

Билет №3 Структура вычислительной системы. Ресурсы ВС – физические, виртуальные. Уровень систем программирования.

Система программирования – это комплекс программ, обеспечивающий поддержание жизненного цикла программы в вычислительной системе.

Жизненный цикл программы в вычислительной системе состоит из четырех основных этапов:

1. **Проектирование** программного продукта. Состоит из нескольких взаимосвязанных между собой действий: исследование, характеристика объектов вычислительной системы, модель функционирования, характеристика инструментальной вычислительной системы, алгоритмы и инструментальные средства, априорная оценка.
2. **Кодирование** (программная реализация). Построение кода на основании спецификаций при использовании языков программирования и трансляторов. Системы поддержки версий – фиксируют реализацию продукта в данный момент времени.
3. **Тестирование и отладка** – это проверка программы на тестовых нагрузках, Принимается решение о формировании минимального набора тестов, более полно проверяющих программу.
4. **Ввод программной системы в эксплуатацию** (внедрение) и сопровождение.

Отладка – процесс поиска, локализации и исправления зафиксированных при тестировании ошибок.

Последний этап предъявляет программному продукту целый ряд специфических требований. Этапы жизненного цикла программы могут комбинироваться.

Среди современных технологий разработки программного обеспечения можно выделить **каскадную модель, каскадную итерационную модель и спиральную модель**, которые более подробно представлены на слайдах.

Система программирования – это комплекс программ, обеспечивающий технологию автоматизации проектирования, кодирования, тестирования, отладки и сопровождения программного обеспечения.

С 90-х годов 20 века по настоящее время – появляются промышленные средства автоматизации проектирования программного обеспечения, case-средств, унифицированного языка моделирования UML. Системы программирования – интегрированные системы.

Средства программирования, доступные на уровне системы программирования – программные средства и компоненты СП, обеспечивающие поддержание жизненного цикла программы. +2 слайда

Билет №4 Структура Вычислительной системы. Ресурсы ВС- физические и виртуальные. Уровень прикладных систем.

2.5 Прикладные системы

Прикладная система – программная система, ориентированная на решение или автоматизацию решения задач из конкретной предметной области.

Этапы развития

Первый этап развития прикладных систем

Задача→ Разработка, программирование→Решение

Второй этап

Развитие систем программирования и появление средств создания и использования библиотек программ

Третий этап

характеризуется появлением *пакетов прикладных программ* ,имеющих развитые и стандартизированные интерфейсы, возможность совместного использования различных пакетов.

2.5.3 Основные тенденции в развитии современных прикладных систем

Современные прикладные системы характеризуются:

• ***Стандартизация моделей автоматизируемых бизнес - процессов***

- • ***B2B*** (business to business)
- • ***B2C*** (business to customer)
- • ***ERP*** (Enterprise Resource Planning)
- • ***CRM*** (Customer Relationship Management)

• ***Открытость системы***

- • ***API - Application Programming Interface***
- Построены на основе современных технологий: Использование интернет систем.

Категории пользователей

1.Оператор или прикладной пользователь (доступны средства пользовательского интерфейса)

2.Системный программист (пользователь компонентов прикладной системы)

3.Системный администратор

4.Оператор или прикладной пользователь (доступны средства пользовательского интерфейса)

5.Системный программист (пользователь компонентов прикладной системы)

6.Системный администратор.

. Выводы

Пользователь и уровни структурной организации вычислительной системы:

- Прикладные программы (набор функциональных средств прикладной системы)
- Системные программы (трансляторы языков высокого уровня, библиотеки)
- Управление логическими/виртуальными ресурсами (интерфейсы драйверов виртуальных устройств)
- Управление физическими ресурсами (интерфейсы драйверов физических устройств)

Аппаратные средства (система команд, аппаратные интерфейсы программного управления физическими устройствами).

Билет №5 Структура вычислительной системы. Понятие виртуальной машины.

Понятие виртуальной машины неотрывно связано с понятием виртуальных и физических ресурсов (дать понятие виртуальных и физических ресурсов из билета N2). Мы можем сделать некий срез уровня любой вычислительной системы, основываясь на иерархии и классификации по уровням. Например, мы можем рассматривать только аппаратный уровень, или только уровень операционной

системы. На каждом из этих уровней мы встретимся с понятием «виртуальной машины». Дело в том, что мы никогда не можем работать просто с «компьютером». Каждый раз нам приходится использовать некую программную прослойку между нами и машиной, будь то ассемблер или Windows 95. Совокупность программных средств, обеспечивающих в любой момент времени нашу связь с компьютером, мы и назовем виртуальной машиной. Хочется подчеркнуть, что виртуальная машина всегда разная. Например, если мы работаем с DOS, то наша виртуальная машина обладает следующими характеристиками: во-первых, она имеет систему команд ДОС, то есть в то время, как физически для нашего компьютера определена система команд низкого уровня, наша виртуальная машина обладает системой команд, которые включают в себя команды «dir» или «cd». Виртуальная машина DOS способна выполнять только одну задачу в один момент времени, она не предназначена для мультипрограммирования, хотя на деле мы можем работать за многопроцессорной рабочей станцией. С другой стороны, виртуальная машина Windows имеет больший объем оперативной памяти (речь идет о подкачке), по сравнению с компьютером, на котором она установлена. То есть можно сказать, что виртуальная машина никак или практически никак не связана с физической, за исключением, конечно же, того, что виртуальная машина в любом случае вынуждена использовать физическую. Рассмотрим виртуальные машины по уровням.

Начнем с уровня физических ресурсов. Пусть у нас есть жесткий диск и драйвер этого диска. В этом случае драйвер представляет собой виртуальную машину, ведь если подумать, драйвер никак не связан с диском, драйвер можно скопировать на дискету и унести от диска. Но драйвер, с другой стороны, и есть для пользователя диск, поскольку именно драйвер – это то, что позволяет использовать диск. Таким образом, можно сказать, что без драйвера диск - не диск. Виртуальная машина здесь – это программа, представляющая собой лишь одну часть аппаратного обеспечения.

Уровень логических ресурсов. Пусть жесткий диск поделен на два логических раздела. В этом случае, интерфейс каждого из разделов – отдельная виртуальная машина. Каждый из логических дисков имеет меньше памяти, чем весь диск в целом, то есть представленные виртуальные машины обладают меньшим количеством ресурсов по сравнению с физическими характеристиками данного компьютера.

Уровень систем программирования. СП дают возможность создать виртуальную машину, имеющую определенный набор команд. Например, компилятор gcc позволяет эмулировать компьютер, чья система команд определена стандартом ANSI C.

Уровень прикладных систем. Рассмотреть базу данных и повторить то же самое.

Билет №6 Основы архитектуры компьютера. Основные компоненты и характеристики. Структура и функционирование ЦП.

Центральный процессор

Структура, функции ЦП

ЦП обеспечивает выполнение программы, размещенной в ОЗУ. Осуществляется выбор машинного слова, содержащего очередную машинную команду, дешифрация команды, контроль корректности данных, определение исполнительных адресов операндов, получение значения операндов и исполнение машинной команды.
Регистровая память процессора – сверхоперативные запоминающие устройства, размещенные в процессоре

Регистры общего назначения (РОН)

Используются в машинных командах для организации индексирования и определения исполнительных адресов операндов, а также для хранения значений наиболее часто используемых операндов, в этом случае сокращается число реальных обращений в ОЗУ и повышается системная производительность ЭВМ.

Специальные регистры

Качественный и количественный состав специализированных регистров ЦП зависит от архитектуры ЭВМ. Ниже представлены некоторые из возможных типов регистров, обычно входящие в состав специализированных регистров. Кроме регистров, рассмотренных ниже, мы будем доопределять эту группу по ходу курса.

Регистр адреса (РА) - содержит адрес команды, которая исполняется в данный момент времени. По содержимому РА ЦП осуществляет выборку текущей команды, по завершении ее исполнения регистр адреса изменяет свое значение тем самым указывает на следующую команду, которую необходимо выполнить.

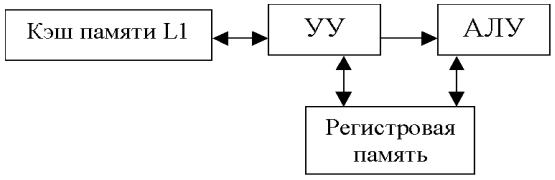
Регистр результата (РР) - содержит код, характеризующий результат выполнения последней арифметико-логической команды. Содержимое РР может характеризовать результат операции. Для арифметических команд это может быть «=0», «>0», «<0», переполнение. Содержимое РР используется для организации ветвлений в программах, а также для программного контроля результатов.

Слово – состояние процессора (ССП или PSW) - регистр, содержащий текущие «настройки» работы процессора и его состояние. Содержание и наличие этого регистра зависит от архитектуры ЭВМ. Например, в ССП может включаться информация о режимах обработки прерываний, режимах выполнения арифметических команд и т. п. Частично, содержимое ССП может устанавливаться специальными командами процессора.

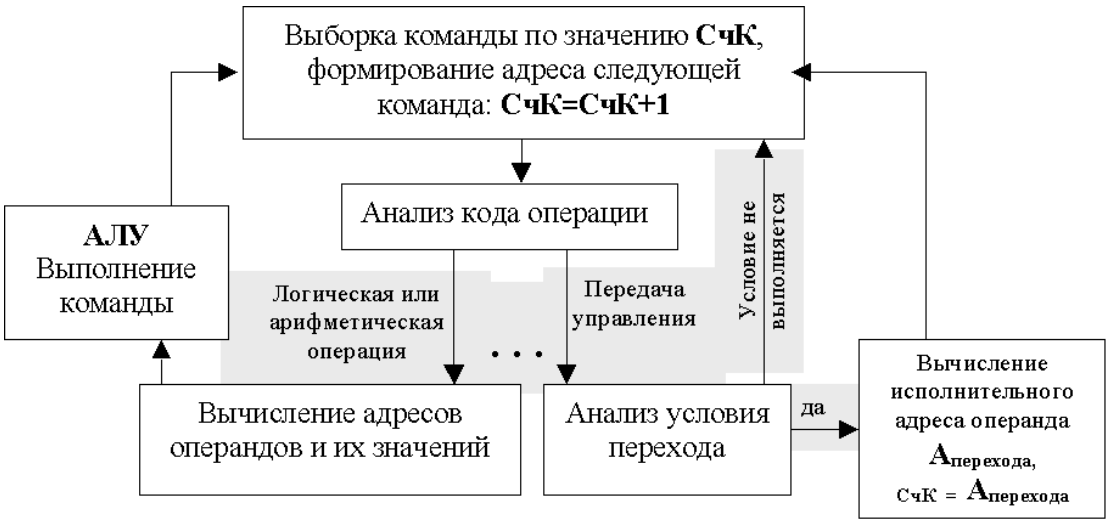
Регистры внешних устройств (РВУ) - специализированные регистры, служащие для организации взаимодействия ЦП с внешними устройствами. Через РВУ осуществляется обмен данными с ВУ и передача управляющей информации (команды управления ВУ и получения кодов результат обработки запросов к ВУ).

Регистр указатель стека - используется для ЭВМ, имеющих аппаратную реализацию стека, в данном регистре размещается адрес вершины стека. Содержимое изменяется автоматически при выполнении «стековых» команд ЦП.
Процессор или центральный процессор (ЦП) компьютера обеспечивает последовательное выполнение машинных команд, составляющих программу, размещенную в оперативной памяти.

Структура организации центрального процессора



Регистровая память – совокупность устройств памяти ЦП,предназначенных для временного хранения операндов, информации, результатов операций.
Устройство управления (control unit)– координирует выполнение команд программы процессором.
Арифметико-логическое устройство (arithmetic/logic unit) –обеспечивает выполнение команд, предусматривающих арифметическую или логическую обработку операндов.
Рабочий цикл процессора – последовательность действий, происходящая в процессоре во время выполнения программы.

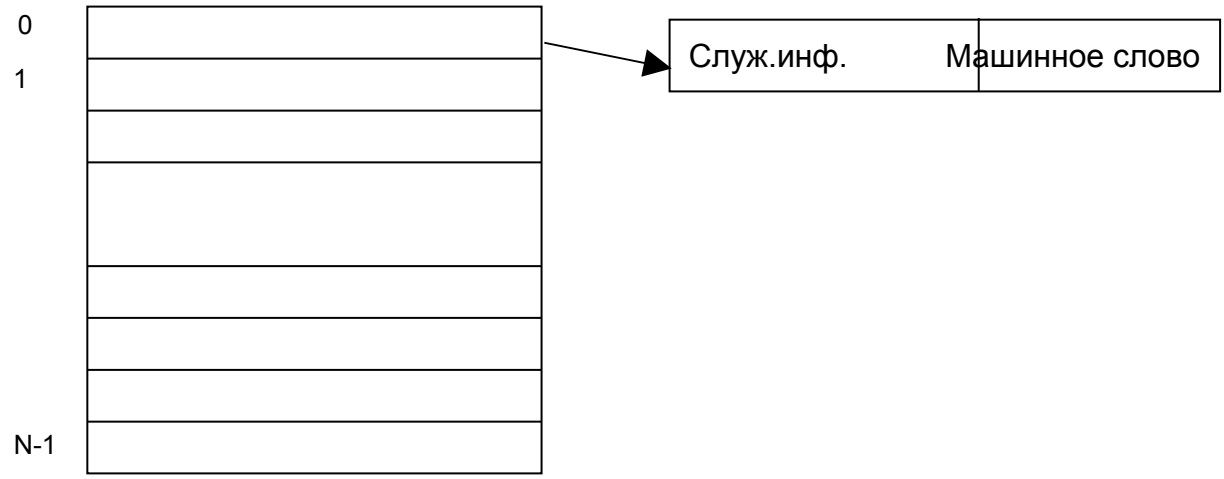


Рабочий цикл процессора определяет основной алгоритм работы процессора и его устройств при выполнении программы. В регистре счетчика команд находится адрес первой команды программы.

1. по содержимому счетчика команд выбирается машинное слово, в котором находится очередная команда.
2. счетчик команд увеличиваем на 1.
3. анализируем код операции команды. Если код операции команды некорректный, то происходит прерывание.
4. Если код операции корректный, то происходит анализ типа этой операции.
5. Вычисление адресов операндов.
6. Выбираются значения операндов, которые помещаются в арифметико-логическое устройство.
7. Выполнение команды. Если выполняемая команда есть команда передачи управления, то она передается в устройство управления и, если условия выполняются, то вычисляется исполнительный адрес (адрес перехода), который заносится в счетчик команд.Переход на начало, к шагу 1.

Билет №7 Основы архитектуры компьютера. Оперативное запоминающее устройство. Расслоение памяти.

ОЗУ - устройство, предназначенное для хранения оперативной информации**.(ОЗУ – хранение программы, выполняющейся в компьютере)** В ОЗУ размещается исполняемая в данный момент программа и используемые ею данные. ОЗУ состоит из ячеек памяти, содержащей поле машинного слова и поле служебной информации.



Машинное слово – поле программно изменяемой информации, в машинном слове могут располагаться машинные команды (или части машинных команд) или данные, с которыми может оперировать программа. Машинное слово имеет фиксированный для данной ЭВМ размер (обычно размер машинного слова – это количество двоичных разрядов, размещаемых в машинном слове).
Служебная информация (иногда ТЭГ) – поле ячейки памяти, в котором схемами контроля процессора и ОЗУ автоматически размещается информация, необходимая для осуществления контроля за целостностью и корректностью использования данных, размещаемых в машинном слове.
В поле служебной информации могут размещаться:

- разряды контроля четности машинного слова (при записи машинного слова подсчет числа единиц в коде машинного слова и дополнение до четного или нечетного в контрольном разряде), при чтении контроль соответствия;
- разряды контроля данные-команда (обеспечение блокировки передачи управления на область данных программы или несанкционированной записи в область команд);
- машинный тип данных – осуществление контроля за соответствием машинной команды и типа ее операндов;

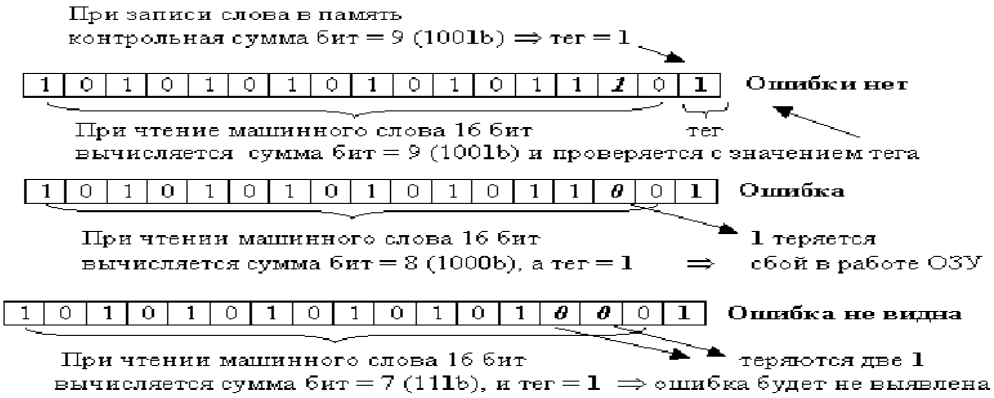
Конкретная структура, а также наличие поля служебной информации зависит от конкретной ЭВМ.

В ОЗУ все ячейки памяти имеют уникальные имена, *имя - адрес ячейки памяти*. Обычно адрес – это порядковый номер ячейки памяти (нумерация ячеек памяти возможна как подряд идущими номерами, так и номерами, кратными некоторому значению). Доступ к содержимому машинного слова осуществляется посредством использования адреса. Обычно скорость доступа к данным ОЗУ существенно ниже скорости обработки информации в ЦП.

Необходимо, чтобы итоговая скорость выполнения команды процессором как можно меньше зависела от скорости доступа к коду команды и к используемым в ней операндам из памяти. Это составляет проблему, которая системным образом решается на уровне архитектуры ЭВМ.

1. *Расслоение ОЗУ* – один из аппаратных путей решения проблемы дисбаланса в скорости доступа к данным, размещенным в ОЗУ и производительностью ЦП. Суть расслоения ОЗУ состоит в следующем. Все ОЗУ состоит из k

блоков, каждый из которых может работать независимо. Ячейки памяти распределены между блоками таким образом, что у любой ячейки ее соседи размещаются в соседних блоках. ***Контроль за целостностью данных.***



Пример контроля за целостностью данных по четности

2. ***Контроль доступа к командам/данными***.

Контроль осуществляется при помощи тегов. Если команда захочет рассмотреть данные в качестве команда то будет прерывание. Происходит проверка на семантическую правильность.

3. ***Контроль доступа к машинным типам данных***.

Тип данных – определенный формат данных, с конкретным набором операций, известных для этого формата. Наличие и формат тега зависит от реализации. Доступ к содержимому машинного слова может быть прямым или косвенным.

Производительность оперативной памяти - скорость доступа процессора к данным, размещенным в ОЗУ:

•**время доступа (access time- t_{access}) -** время между запросом на чтение слова из оперативной памяти и получением содержимого этого слова.

•**длительность цикла памяти (cycle time - t_{cycle})** - минимальное время между началом текущего и последующего обращения к памяти.

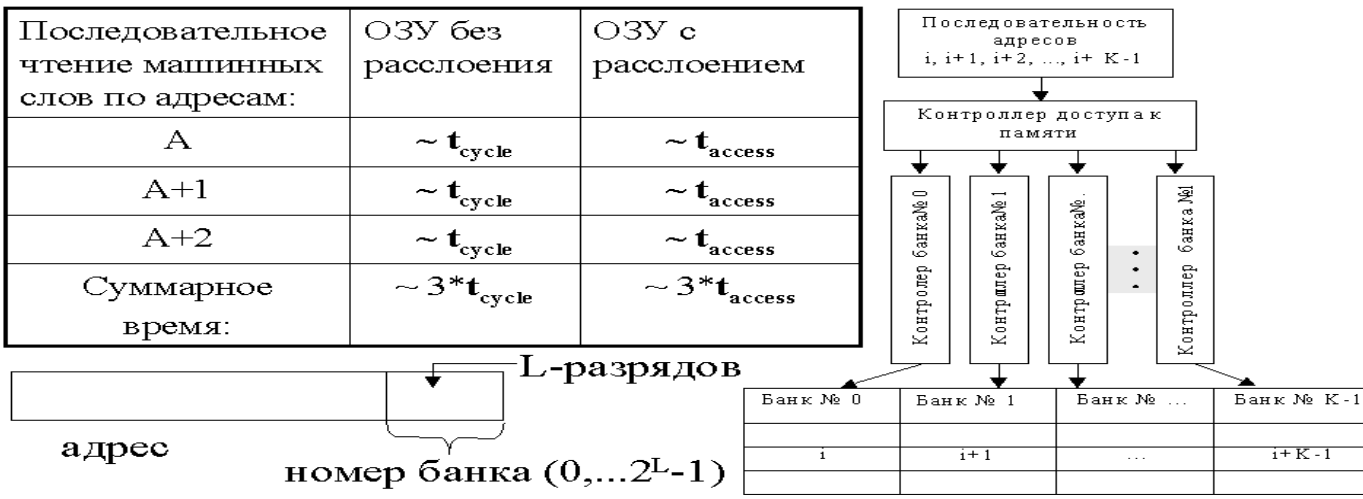
$$(t_{cycle} > t_{access})$$

Расслоение памяти

Расслоение ОЗУ - один из путей аппаратного решения проблемы дисбаланса между скоростью доступа к данным, находящимся в ОП, и производительностью процесса. Так же используются контроллеры.

ОЗУ делится на **K** независимых банков памяти, где $K = 2^L$

Последовательное чтение машинных слов по адресам:	ОЗУ без расслоения	ОЗУ с расслоением
A	$\sim t_{cycle}$	$\sim t_{access}$
A+1	$\sim t_{cycle}$	$\sim t_{access}$
A+2	$\sim t_{cycle}$	$\sim t_{access}$
Суммарное время:	$\sim 3*t_{cycle}$	$\sim 3*t_{access}$



Использование расслоения памяти. Физически ОЗУ представимо в виде объединения k устройств, способных хранить одинаковое количество информации и способных взаимодействовать с процессором независимо друг от друга. При этом адресное пространство ВС организовано таким образом, что подряд идущие адреса, или ячейки памяти, находятся в соседних устройствах (блоках) оперативной памяти. Программа состоит (в большей степени) из линейных участков. Если использовать этот параллелизм, то можно организовать в процессоре еще один буфер, который организован так же, но в котором размещаются машинные команды. За счет того, что есть параллельно работающие устройства, то этот буфер автоматически заполняется вперед. Т.е. за одно обращение можно прочесть k машинных слов и разместить их в этом буфере. Далее, действия с буфером команд похожи на действия с буфером чтения/записи. Когда нужна очередная команда (ее адрес находится в счетчике команд), происходит ее поиск (по адресу) в буфере, и если такая команда есть, то она считывается. Если такой команды нет, то опять-таки работает внутренний алгоритм выталкивания строки, новая строка считывается из памяти и копируется в буфер команд. Расслоение памяти в идеале увеличивает скорость доступа в k раз, плюс буфер команд позволяет сократить обращения к ОЗУ.

Билет №8 Основы архитектуры компьютера. Основные компоненты и характеристики. Кэширование ОЗУ

Вернемся к проблеме дисбаланса скорости доступа к ОЗУ и скорости обработки информации ЦП.

Первое решение – использовать программные средства. Программист может разместить наиболее часто используемые операнды в РОН, тем самым сокращается количество «медленных» обращений в ОЗУ. Результат решения во многом зависит от качества программирования.

Второе решение – использование в архитектуре ЭВМ специальных *регистровых буферов или КЭШ памяти*.

Регистровые буфера или КЭШ память предназначены для разрешения проблемы несоответствия скоростей работы ОЗУ и ЦП, на аппаратном уровне, т.е. эта форма оптимизации в системе организована аппаратно и работает всегда, вне зависимости от исполняемой программы. Следует отметить, что результат этой оптимизации, в общем случае зависит от характеристик программы (об этом несколько позднее). Традиционно, в развитых ЭВМ используется аппаратная буферизация доступа к операндам команд, а также к самим командам.

Буферизация работы с операндами

Буфер операндов – аппаратная таблица, логически являющаяся компонентом ЦП (физически это может быть и отдельное от ЦП устройство), призванная аппаратно минимизировать количество обращений к «медленному» ОЗУ при записи и чтении операндов.

Таблица состоит из фиксированного числа строк. Каждая строка имеет следующие поля:

Адрес	Значение	Признак изменения	Код стирания

- адрес – физический адрес машинного слова в ОЗУ;
- значение – значение машинного слова, соответствующего адресу;
- признак изменения – код, характеризующий факт изменения поля значения (в соответствующей ячейке ОЗУ значение отличается от значения в таблице);
- код старения – код, характеризующий интенсивность обращений к данной строке. По значению поля определяются наиболее «популярные» строки. Конкретный алгоритм изменения данного поля зависит от ЭВМ.

Примерные алгоритмы использования буфера операндов

Алгоритм для чтения данных из ОЗУ

Пусть имеется команды чтения данных из машинного слова по физическому адресу $A_{исп}$.

1. Поиск по таблице строки, содержащей адрес, совпадающий с $A_{исп}$. Если такой строки нет, то на п. 3.
2. Происходит обновление кода старения. Результатом команды чтения является содержимое поля «Значение».
3. По значениям поля «Код старения» осуществляется поиск строки, используемой наименее интенсивно.
4. Анализируется код изменения. Если значение изменялось в таблице, то происходит запись значения по адресу в ОЗУ.
5. Считывается машинное слово из ОЗУ по адресу $A_{исп}$ и заполняется данная строка.
6. Считанное значение является результатом выполнения команды чтения.

Алгоритм для записи данных в ОЗУ

Пусть имеется команда записи значения в машинное слово по физическому адресу $A_{исп}$.

1. Поиск по таблице строки, содержащей адрес, совпадающий с $A_{исп}$. Если такой строки нет, то на п. 3.
2. Значение записывается в поле «Значение». Происходит обновление полей «Признак изменения», «Код старения». Выполнения команды записи завершено.
3. По значению поля «Код старения» осуществляется поиск строки, используемое наименее интенсивно.

4. Анализируется код изменения. Если значение изменялось в таблице, то происходит запись значения по адресу в ОЗУ.
5. Происходит обновление содержимого полей строки в соответствии с командой записи. Выполнение команды завершено.

Буферизация выборки команд

Буфер команд – минимизация обращений в ОЗУ за машинными командами.

Адрес	Значение	Код старения

Интерпретация одноименных полей аналогична буферу операндов.

Примерный алгоритм использования

Центральному процессору требуется для выполнения машинная команда, размещенная по физическому адресу ОЗУ $А_{исп}$.

1. Поиск по таблице строки, содержащей $А_{исп}$. Если такой не то на п. 3.
2. Обновление поля «Код старения», чтение поля «Значение» и передача его процессору для исполнения.
3. Поиск наименее интенсивно используемой строки. Чтение машинного слова из ОЗУ по адресу $А_{исп}$ и заполнение всех полей строки. Передача процессору значения для исполнения.

Конкретные реализация и алгоритмы зависят от архитектуры ЭВМ. Возможно, например, использование одного буфера.

(Регистры буферной памяти (Cache, КЭШ).

Следующая группа регистров — регистры, относящиеся к т.н. буферной памяти. Мы возвращаемся к проблеме взаимодействия процессора и оперативной памяти и сглаживанию скоростей доступа в оперативную память. Предположим, у нас есть некоторая программа, которая производит вычисление некоторого выражения, при этом, процесс вычисления этого выражения будет представим следующим образом. В какие-то моменты идут обращения за операндами в оперативную память, в какие-то моменты обработанные данные записываются в оперативную память. Есть один из нескольких путей, которые сглаживают несоответствие скоростей процессора и оперативной памяти, который заключается в сокращении реальных обращений к оперативной памяти. Процессоры содержат быстродействующую регистровую память, призванную

буферизовать обращения к оперативной памяти.

Алгоритм чтения из оперативной памяти следующий:

Проверяется наличие в специальном регистровом буфере строчки, в которой находится исполнительный адрес, совпадающий с исполнительным адресом требуемого операнда. Если такая строчка имеется, то соответствующее этому адресу значение, считается

значением операнда и передается в процессор для обработки (т.е. обращение в оперативную память не происходит).

Если такой строчки нет, то происходит обмен с оперативной памятью, и копия полученного значения помещается в регистровый буфер и помечается исполнительным адресом этого значения в оперативной памяти. Содержимое операнда поступает в процессор для обработки. При этом решается проблема размещения новой строчки. Аппаратно ищется свободная строка (но она может быть только в начале работы машины), и если таковая не найдена, запускается аппаратный процесс вытеснения из этого буфера наиболее “старой” строчки. “Старость” определяется по некоторому предопределенному критерию. Например, признаком старения может быть количество обращений к этому буферу, при котором нет обращений к этой строчке. В каждом таком случае число в третьем столбце таблицы увеличивается на единицу. Короче говоря, аппаратура решает, какую из строк надо вытолкнуть из таблицы, чтобы на ее место записать новое содержимое. При этом учитывается информация о том, были ли обращения к данной строке с использованием команд записи в память. Если такие обращения были, то перед выталкиванием происходит запись в ОЗУ по исполнительному адресу содержимого нашей строчки.

Алгоритм записи в оперативную память симметричен. Когда в программе встречается команда записи операнда в память, аппаратура выполняет следующие действия. Проверяется наличие в буфере строки с заданным исполнительным адресом. Если такая строка есть, то в поле “Содержимое” записывается новое значение и аппаратно корректируется признак старения строк. Если такой строчки нет, то запускается

описанный выше процесс выталкивания, и затем информация размещается в освободившейся строке.

Этот буфер чтения/записи служит достаточно мощным средством для

минимизации обращений к ОЗУ. Наибольший эффект достигается при небольших циклах, когда все операнды размещаются в буфере, и после этого циклический процесс работает без обращений к ОЗУ. Иногда эти буфера называют КЭШ-буферами, а также ассоциативной памятью, потому что доступ к этой памяти осуществляется не по адресу (как в ОЗУ), а по значению поля. Реально, все механизмы могут быть устроены иначе,
чем мы здесь изучаем, т.к. мы изучаем некоторую обобщенную систему.

Билет №9 Основы архитектуры компьютера. Аппарат прерываний. Последовательность действий в вычислительной системе при обработке прерываний

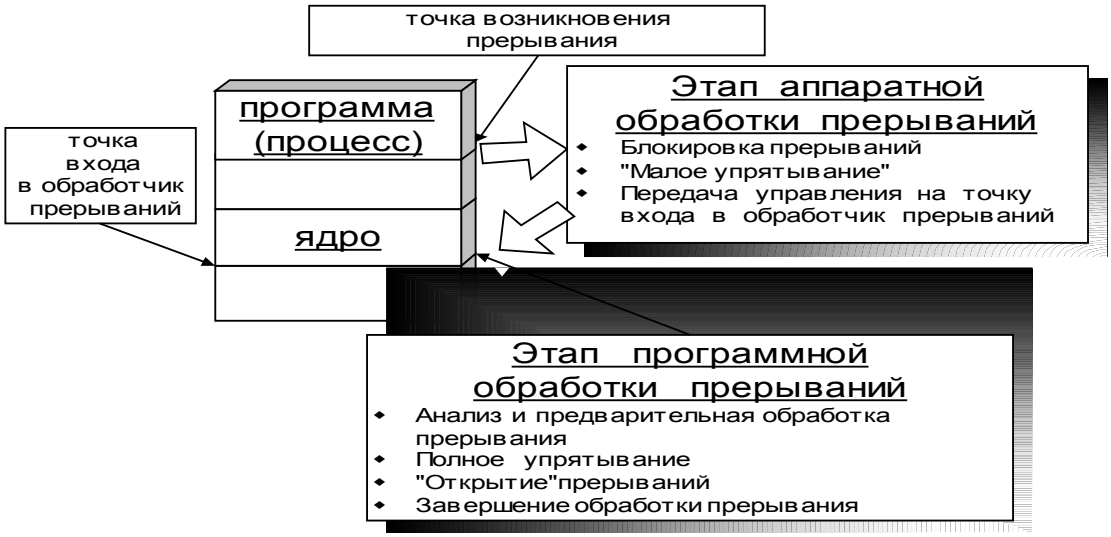
Аппарат прерываний ЭВМ - возможность аппаратуры ЭВМ стандартным образом обрабатывать возникающие в вычислительной системе события. Данные события будем называть прерываниями.

Определение. Последовательность действий при обработке

Итак, *прерывание* - одно из событий в вычислительной системе, на возникновение которого предусмотрена стандартная реакция аппаратуры ЭВМ. Количество различных типов прерываний ограничено и определяется при разработке аппаратуры ЭВМ.

Прерывания можно разделить на две группы внутренние и внешние.
Внутренние прерывания инициируются схемами контроля работы процессора. Это может быть реакция ЦП на программную ошибку. Например, деление на ноль.
Внешние прерывания – это средство, позволяющее ЭВМ корректно взаимодействовать с внешними устройствами. Это может быть событие, связанное с поступлением новой информации от ВУ или возникновение ошибки во ВУ.

Аппарат прерываний ЦП обеспечивает стандартную реакцию аппаратуры при возникновении прерывания. Тем самым обеспечивается возможность корректной обработки прерываний в ВС.
Рассмотрим обобщенную (и упрощенную) модель последовательности действий, происходящих в ВС при возникновении прерывания.



При обработке события, связанного с возникновением прерывания на первом этапе работает аппаратура ВС. При этом аппаратно (без участия программы) выполняются следующие действия:

1. Включается режим блокировки прерываний. При этом режиме в системе запрещается инициализация новых прерываний. Возникающие в это время прерывания могут либо игнорироваться, либо откладываться (зависит от конкретной аппаратуры ЭВМ и типа прерывания).
2. Обработка прерывания предполагает сохранение возможности корректного продолжения прерванной программы (процесса) с точки прерывания. Поэтому следующий шаг аппаратной обработки – “малое упрятывание“ – копирование в специальную регистровую память ЦП (регистровый буфер, таблицу) минимального количества регистров и настроек ЦП, достаточных для запуска программы ОС, обрабатывающей прерывания. Это заведомо счетчик команд, регистр результата, некоторое количество регистров общего назначения. Следует отметить, что возможно организовать копирование (или упрятывание) всех регистров, используемых программой, но это, в общем случае, нецелесообразно, так как может потребовать значительных объемов регистровой памяти, а также потребует значительного времени работы в режиме блокировки прерываний.
3. Следующим шагом является переход на программный режим обработки прерываний. Для этих целей в аппаратуру ВМ обычно жестко “зашивается” адрес точки в ОЗУ, начиная с которой предполагается размещение части ОС, занимающейся программной обработкой прерываний – точка входа в обработчик прерываний. (Возможно определение не одной, а группы таких точек – по одной на тип или группу прерываний). Переход на программный этап обработки прерываний есть передача управления на точку входа в обработчик прерываний. Этот переход осуществляется не программно (за счет исполнения одной из команд передачи управления), а аппаратно (например, аппаратной записью в счетчик команд адреса точки входа).

Второй этап. Программная обработка прерывания. Управление передано на точку ОС, занимающуюся обработкой прерывания. При входе в эту точку часть ресурсов ЦП, используемых программами освобождена (результат малого упрятывания). Поэтому будет запущена программа ОС, которая может использовать только освобожденные малым упрятыванием ресурсы ЦП (перечень доступных в этот момент ресурсов – характеристика аппаратуры). Выполняется следующая последовательность действий:

1. Анализ и предварительная обработка прерывания. Происходит идентификация типа прерывания, определяются причины. Если прерывание " короткое" обработка не требует дополнительных ресурсов ЦП и времени, то прерывание обрабатывается, происходит разблокировка прерываний и возврат в первоначальную программу (не вдаваясь в подробности, эта последовательность действий организована так, что гарантируется корректное восстановление всех регистров и настроек ЦП).
   Если прерывание требует использования всех ресурсов ЦП, то переходим к следующему шагу.
2. “Полное упрятывание” осуществляется полное упрятывание состояния всех ресурсов ЦП, использовавшихся прерванной программой (все регистры, настройки, режимы и т.д.) в специальную программную таблицу (в контекст процесса или программы – о нем позже). То есть в данную таблицу копируется содержимое аппаратной таблицы, содержащей сохраненные

значения ресурсов ЦП после малого упрятывания, а также копируются все оставшиеся ресурсы ЦП используемые программно, но не сохраненные при малом упрятывании. После данного шага программе обработки прерываний становятся доступны все ресурсы ЦП, а прерванная программа получает статус ожидания завершения обработки прерывания. В общем случае программ или процессов, ожидающих завершения обработки прерывания может быть произвольное количество.

3. До данного момента времени все действия происходили в режиме блокировки прерываний. Почему? Потому что режим блокировки прерываний – единственная гарантия оттого, что не придет новое прерывание и при его обработке не потеряются данные, необходимые для продолжения прерванной программы (регистры, режимы, таблицы ЦП). После полного упрятывания разблокируются прерывания (то есть включается стандартный режим при котором возможно появление прерываний).
4. Заключительный этап – завершение обработки прерывания.

Вот упрощенная схема обработки прерывания, в реальных системах она может иметь отличия и быть сложнее. Но основные идеи обычно остаются неизменными. Аппарат прерываний позволяет системе фиксировать и корректно обрабатывать различные события, возникающие как внутри системы, так вне нее.

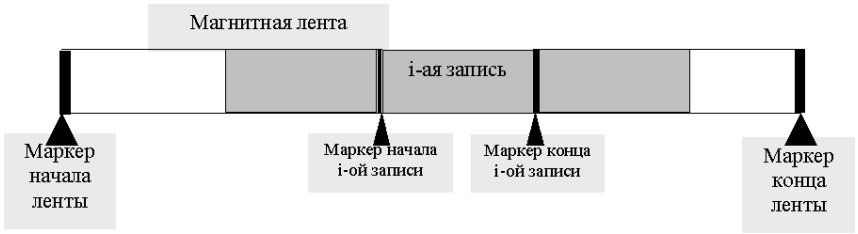
Билет №10 Основы архитектуры компьютера. Внешние устройства. Организация управления и потоков данных при обмене с внешними устройствами

3.6.1 Внешние запоминающие устройства (ВЗУ).

Обмен данными:
• записями фиксированного размера – ***блоками***
• записями произвольного размера
Доступ к данным:
• операции чтения и записи (жесткий диск, CDRW).
• только операции чтения (CDROM, DVDROM, …).
Последовательного доступа:
•Магнитная лента
Прямого доступа:
• Магнитные диски
• Магнитный барабан
• Магнито - электронные ВЗУ прямого доступа

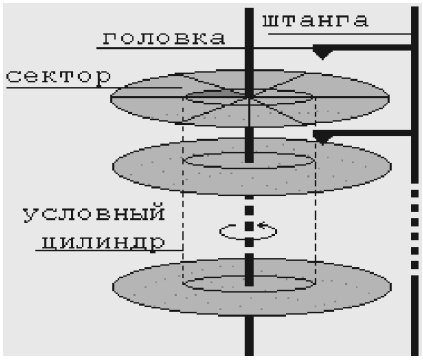
3.6.1.1 Устройство последовательного доступа

Магнитная лента



Чтобы добраться до определенной записи, нужно пройти все предыдущие.

3.6.1.2 Устройства прямого доступа

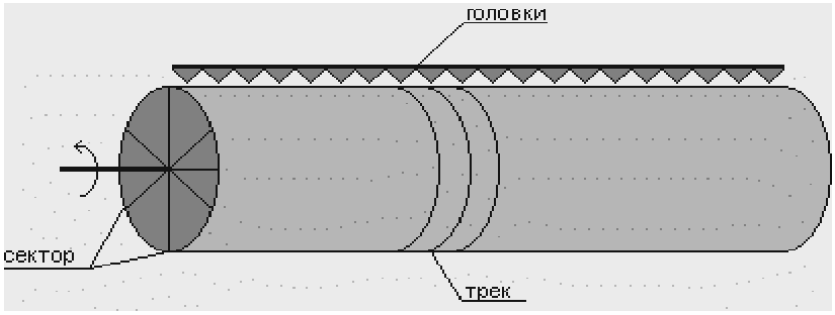


Магнитные диски

Блок может перемещаться от края к центру. Каждое устройство характеризуется фиксированным числом цилиндров. Дорожки относящиеся к одному цилиндру также пронумерованы. Дорожки образуют концентрические окружности. Все дорожки разделены на сектора. Начала одноименных секторов лежат в одной плоскости. Для задания координат определенного сектора в управляющее устройства необходимо передать:

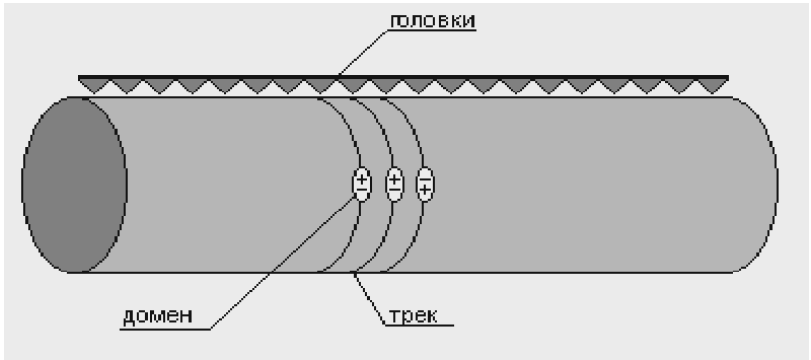
1) номер цилиндра, где расположен сектор
2) номер дорожки на которой находится сектор
3) номер сектора

Магнитный барабан



Предназначен больших вычислительных комплексов. Представляет из себя большой цилиндр длиной до метра, в диаметре 30 – 40 см. Поверхность покрыта особым веществом, над поверхностью штанга с головками над треками. Скорость доступа достаточно большая. Механическая составляющая только вращение барабана.

Магнито-электронные ВЗУ прямого доступа

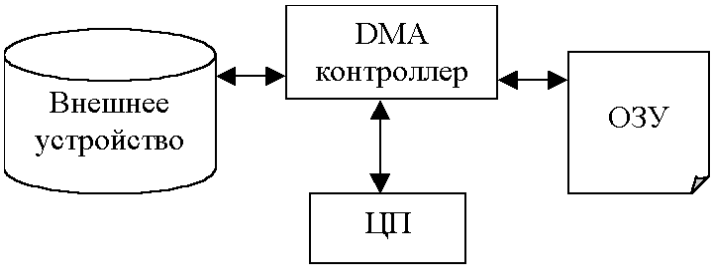


3.6.2 Организация потоков данных при обмене с внешними устройствами

Обмен данных осуществляется через центральный процессор.

Например, при чтении и получении данных из внешнего устройства они попадают на специальные регистры процессора и далее в память.

Обмен с использованием прямого доступа к памяти (direct memory access – DMA).



Нет необходимости для организации обмена использовать оперативную память. Но этот объем данных ограничен. Когда данные кончаются процессор выполняет дополнительную работу.

3.6.4 Организация управления внешними устройствами

1. **Непосредственное управление** внешними устройствами центральным процессором.

2. **Синхронное** управление внешними устройствами с использованием контроллеров внешних устройств.
3. **Асинхронное** управление внешними устройствами с использованием контроллеров внешних устройств.

4. Использование **контроллера прямого доступа** к памяти (DMA) при обмене.

Управление внешними устройствами **с использованием процессора или канала ввода/вывода**.

Прерывания: организация работы внешних устройств.

Одно из основных достижений прерываний – возможность организации асинхронной работы с внешними устройствами. Вернемся к ее рассмотрению.

Пусть в системе имеется прерывание “обращение к системе”. Оно используется для организации доступа к функциям ОС.

Синхронная работа с ВУ

При синхронной организации обмена программа будет приостановлена с момента обращения к ВУ до момента завершения обмена. Дисбаланс между скоростью выполнения машинных команд и скоростью работы ВУ колоссальный. Поэтому задержки при синхронной работе крайне и крайне ощутимы.

Асинхронная работа с ВУ

Последовательность действий следующая

1. Программа инициирует прерывание “обращение к системе”, тем самым передается заказ на выполнение обмена, (параметры заказа могут быть переданы через специальные регистры, стек и т.п.) Происходит обработка прерывания (при этом программа (процесс) находится в ожидании). При обработке прерывания конкретному драйверу устройства передается заказ на выполнение обмена (который поступает в очередь).
2. После завершения обработки прерывания “обращение к системе” программа продолжает свое выполнение до завершения обмена (на самом деле это не всегда так, почему – ответ позднее).
3. Выполнение программы приостанавливается по причине возникновебния прерывания – завершение обмена с конкретным устройством. После обработки прерывания выполнение будет продолжено.

Очевидно, что асинхронная схема обработки обращений к ВУ позволяет сглаживать системный дисбаланс в скорости выполнения машинных команд и скоростью доступа к ВУ. Это еще одно из решений объявленной в начале курса проблемы.

Представленная выше схема организации обмена является достаточно упрощенной. Она не затрагивает случаев синхронизации доступа к областям памяти, участвующим в обмене. Проблема состоит в том, что, например, записывая некую область данных на ВЗУ, после обработки заказа на обмен, но до завершения обмена, программа может попытаться обновить содержимое области, что является некорректным. Поэтому в реальных системах для синхронизации работы с областями памяти, находящимися в обмене, используется возможность ее аппаратного закрытия на чтение и/или запись. То есть при попытке обмена с закрытой областью памяти произойдет прерывание. Это позволяет остановить выполнение программы до завершения обмена, если программа попытается выполнить некорректные операции с областью памяти, находящейся в обмене (попытка чтения при незавершенной операции чтения с ВУ или записи при незавершенной операции записи данной области на ВУ).

БИЛЕТ 11 Иерархия памяти

4.4. Иерархия памяти.

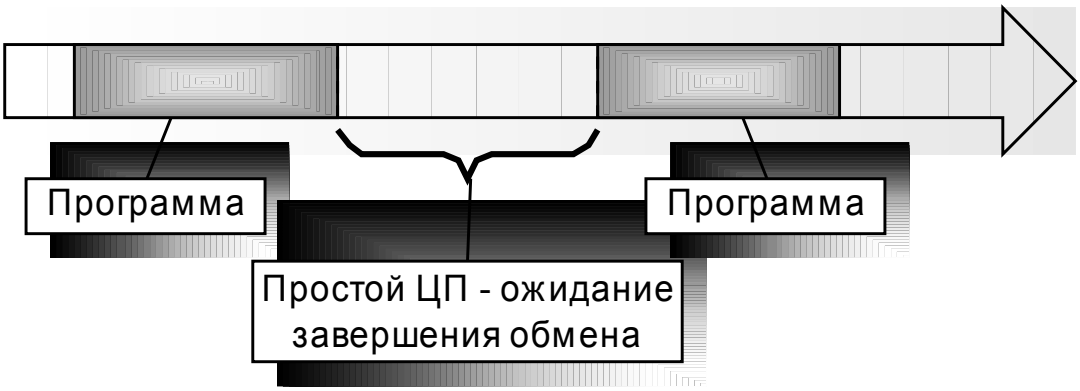
1. В центральном процессоре наиболее быстрые и наиболее дорогостоящие – регистры общего назначения и кэш-буфер.

2. Оперативное запоминающее устройство: кэш-устройства (вне центрального процессора – между оперативной памятью и центральным процессором).
3. Внешние устройства – для организации оперативного доступа к данным.
4. Устройства прямого доступа без кэш-буферизации.
5. Устройства для долговременного массового хранения данных.

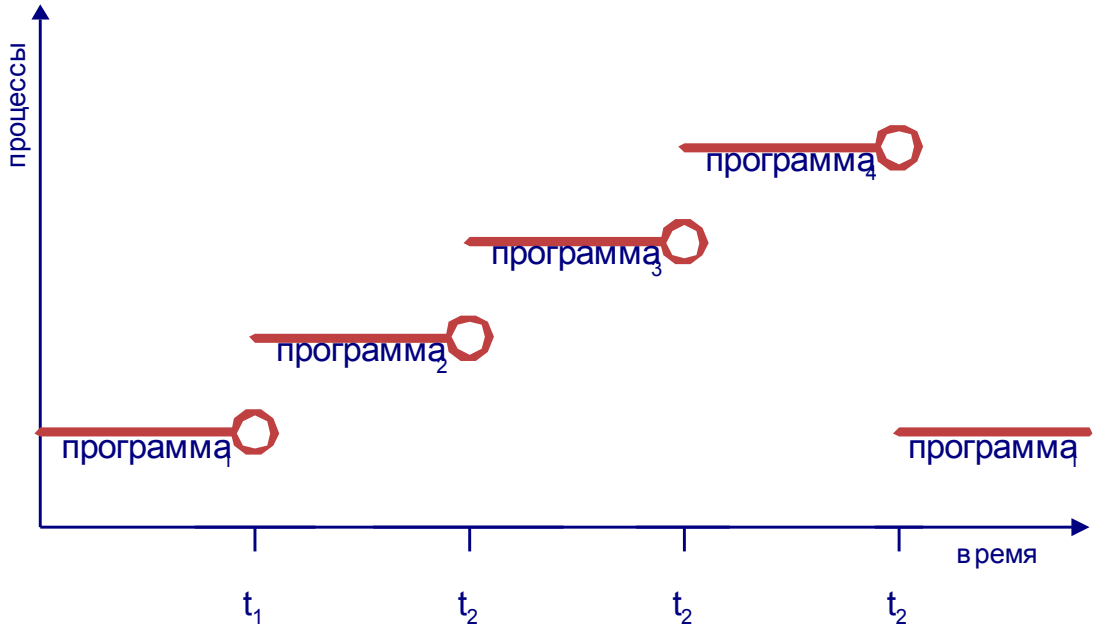
БИЛЕТ 12 Мультипрограммный режим

Итак, выше мы выяснили, что, несмотря на возможность асинхронной работы с ВУ, имеют место периоды ожидания программой завершения обмена. Если система обрабатывает единственную программу, то в это время ЦП не производит

никакой полезной работы, то есть простаивает (на самом деле термин простой достаточно условный, так как при этом работает операционная система).



Решением проблемы простоя ЦП в этом случае является использование ВС в ***мультипрограммном режиме***, в режиме при котором возможна организация переключения выполнения с одной программы на другую



На рисунке изображена подобная мультипрограммная система, обрабатывающая одновременно 4 программы (процесса). t_1 – момент времени в который программа$_1$ будет остановлена для ожидания завершения обмена (до момента времени t_4). В момент времени t_1 система запускает выполнение программы$_2$, которая выполняется до момента времени t_2. С t_2 программа$_2$ также начинает ждать завершения своего обмена и т.д.

Для корректной организации мультипрограммной обработки необходима аппаратная поддержка ЭВМ. Как минимум аппаратура ЭВМ должна поддерживать следующие функции.

1. ***Аппарат защиты памяти***. Аппаратная возможность ассоциирования некоторых областей ОЗУ с одним из выполняющихся процессов/программ. Настройка аппарата защиты памяти происходит аппаратно, то есть

назначение программе/процессу области памяти происходит программно (т.е., в общем случае операционная система устанавливает соответствующую информацию в специальных регистрах), а контроль за доступом – автоматически. При этом при попытке другим процессом/программой обратиться к этим областям ОЗУ происходит прерывание “Защита памяти”

2. Наличие специального режима ***операционной системы (привилегированный режимом или режим супервизора)*** ЦП. Суть заключается в следующем: все множество машинных команд разбивается на 2 группы. Первая группа – команды, которые могут исполняться всегда (пользовательские команды). Вторая группа – команды, которые могут исполняться только в том случае, если ЦП работает в режиме ОС. Если ЦП работает в режиме пользователя, то попытка выполнения специализированной команды вызовет прерывание – "Запрещенная команда”. Какова необходимость наличия такого режима выполнения команд? Простой пример – управление аппаратом защиты памяти. Для корректного функционирования этого аппарата необходимо обеспечить централизованный доступ к командам настройки аппарата защиты памяти. То есть эта возможность должна быть доступна не всем программам.
3. Необходимо наличие аппарата прерываний. Как минимум в машине должно быть прерывание по таймеру, что позволит избежать “зависания“ всей системы при зацикливании одной из программ.

БИЛЕТ 13

Организация регистровой памяти (регистровые окна, стек)

Регистровые окна. Компьютер поддерживает аппарат виртуальных регистров. Команды программы могут оперировать с регистрами общего назначения.

Одно из решений – регистровые окна. В компьютере есть фиксированный набор физических регистров с номерами 0, …, к-1. В программе доступна нумерация виртуальных регистров 0, …,k-1, где l<к.

Аппарат позволяет привязать регистровые окна к множеству физических регистров. Окна перемещаются дискретно. Окно состоит из трех частей: входные, выходные регистры; локальные (внутренние) регистры. Аппаратура обеспечивает существование фиксированного количества окон. Окна расположены циклическим образом. Этот аппарат позволяет активизировать вложенные программы.

В каждый момент времени программа может использовать множество регистров общего назначения. Каждое регистровое окно состоит из трех частей: двух номеров регистров для приема и передачи информации. Средняя часть – для локализации подпрограммы.

Обращение к подпрограмме: параметры, которые должны быть переданы, размещаются в третьей части окна. Осуществляется смена указателя на текущее окно.

Микропроцессоры SPARK используют такую архитектуру.

Два регистра: указатель текущего окна и указатель сохраненного окна. Возникает необходимость сохранения регистровых окон в память. Регистр SWP указывает на сохраненное окно.

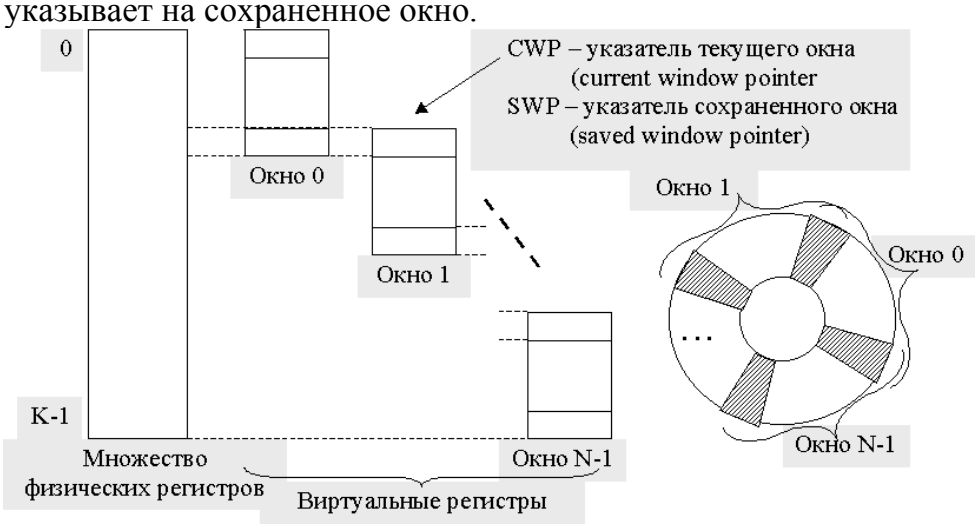


Схема работы:

При обращении к функции: программно увеличиваем указатель на текущее окно на 1 по модулю N. Проверяется содержимое этого указателя и указателя на сохраненное окно. Если они совпадают, то происходит прерывание. Если же совпадения нет, то мы работаем с новым окном.

При выходе из функции: уменьшаем указатель текущего окна, сравниваем с указателем сохраненного окна.

5.2. Модель организации регистровой памяти в Intel Itanium.

Эта модель более совершенная. Регистры, доступные для программы, представляют собой множество из 128 регистров. Первые 32 из них – статические регистры (общие для всех случаев, они никак не меняются). После статических регистров располагаются динамические регистры, которые располагаются с 32 регистра до 127 (всего 96 регистров). Есть возможность при обращении к подпрограммам изменять регистровые окна.

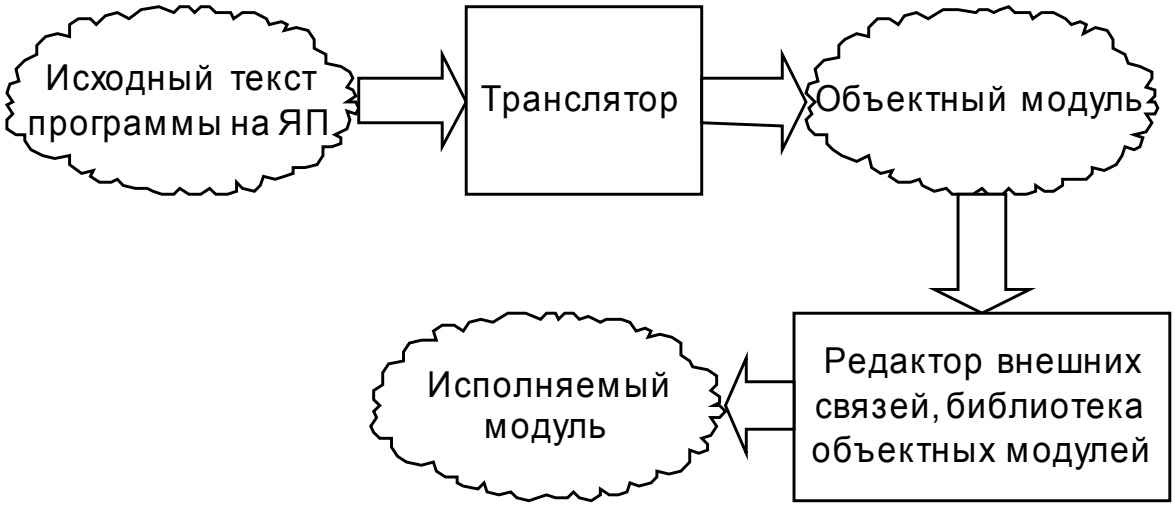
Отличие от предыдущей модели: размер окна при переключении окон может варьироваться от 96 до 1 регистра.

БИЛЕТ 14 Виртуальная оперативная память

Аппарат виртуальной памяти

Рассмотрим некоторые проблемы организации адресации в программах/процессах и связанные с ними проблемы использования ОЗУ в целом.

В общем случае схема получения исполняемого кода программы следующая:

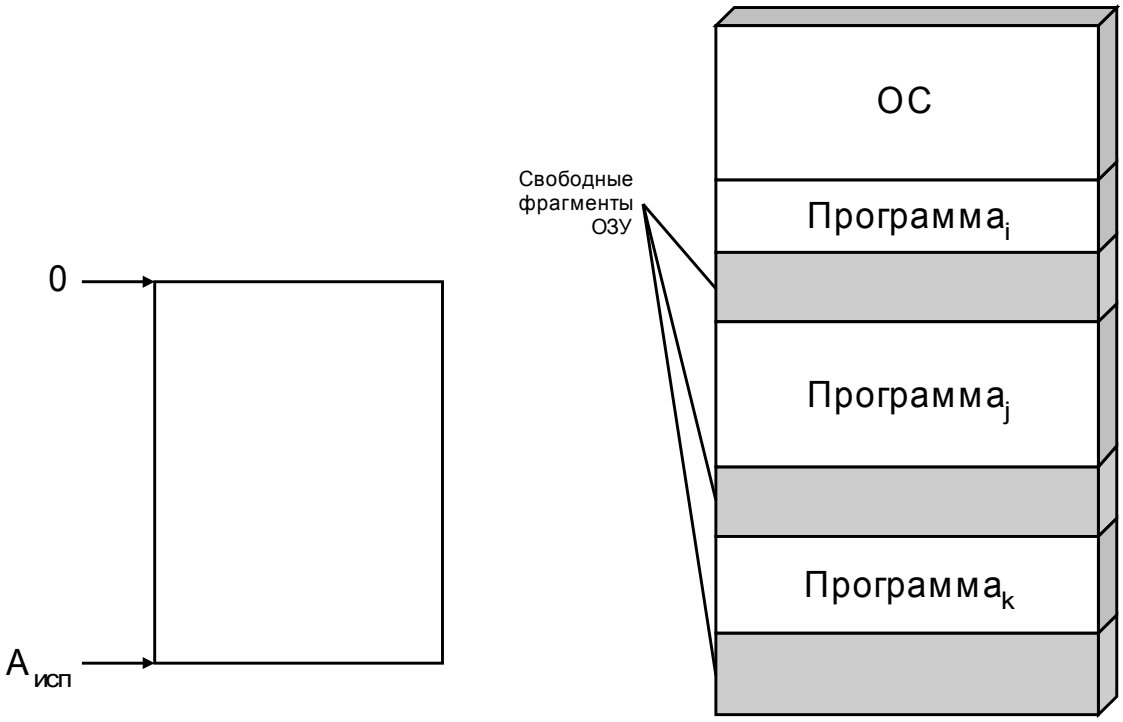


Данная схема достаточно очевидна, так как она связана с привычным для нас процессами трансляции. Остановимся подробней на исполняемом модуле. Данный модуль представляет собой готовую к выполнению программу в машинных кодах. При этом внутри программы к моменту образования исполняемого модуля используется модель организации адресного пространства программы (эта модель, в общем случае не связана с теми ресурсами ОЗУ, которые предполагается использовать позднее). Для простоты будем считать, что данная модель представляет собой непрерывный фрагмент адресного пространства в пределах которого размещены данные и команды программы. Будем называть подобную организацию адресации в ***программе программной адресацией или логической/виртуальной адресацией***.

Итак, повторяем, на уровне исполняемого кода имеется программа в машинных кодах, использующая адреса данных и команд. Эти адреса в общем случае не являются адресами конкретных физических ячеек памяти, в которых размещены эти данные, более того, в последствии мы увидим, что виртуальным (или программным) адресам могут ставиться в соответствие произвольные физические адреса памяти. То есть при реальном исполнении программы далеко не всегда виртуальная адресация, используемая в программе совпадает с физической адресацией, используемой ЦП при выполнении данной программы.

Элементарное программно-аппаратное решение – использование возможности ***базирования адресов***. Суть его состоит в следующем: пусть имеется исполняемый программный модуль. Виртуальное адресное пространство этого модуля лежит в диапазоне [0, $A_{кон}$]. В ЭВМ выделяется специальный регистр базирования $R_{баз.}$, который содержит физический адрес начала области памяти, в которой будет размещен код данного исполняемого модуля. При этом исполняемые адреса, используемые в модуле будут автоматически преобразовываться в адреса физического размещения данных путем их сложения с регистром $R_{баз.}$. Таким образом код используемого модуля может перемещаться по пространству

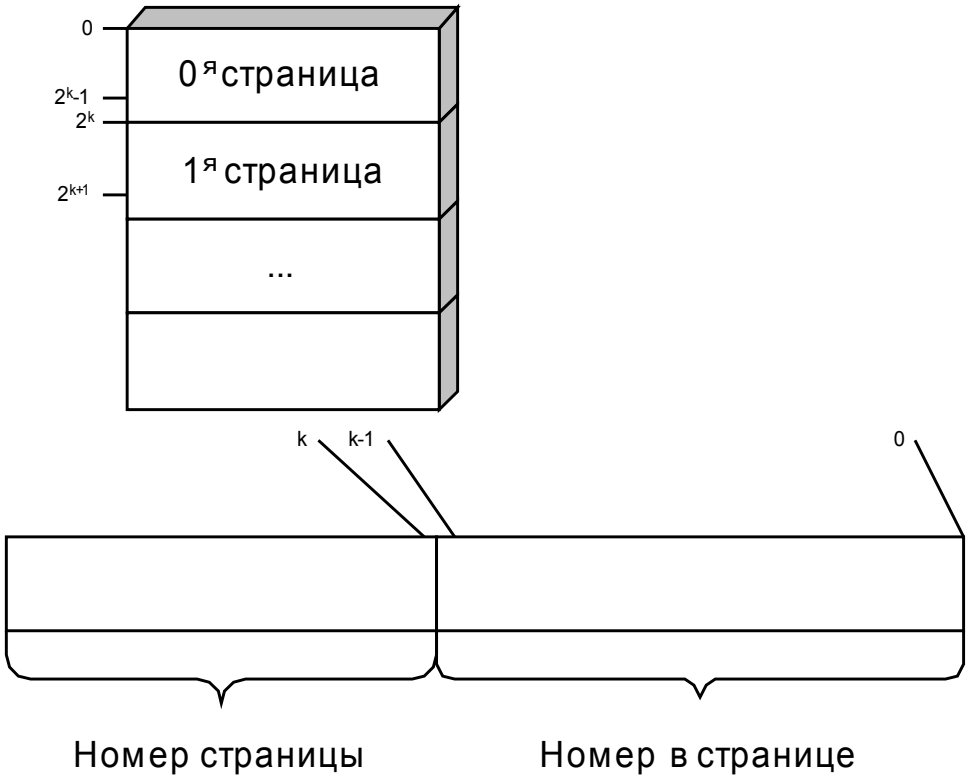
физического ОЗУ. Эта схема является элементарным решением организации простейшего ***аппарата виртуальной памяти***. То есть аппарата, позволяющего автоматически преобразовывать ***виртуальные адреса программы*** в адреса физической памяти.
Рассмотрим более сложные механизмы организации виртуальной памяти.



Пусть имеется вычислительная система, функционирующая в мультипрограммном режиме. То есть одновременно в системе обрабатываются несколько программ/процессов. Один из них занимает ресурсы ЦП. Другие ждут завершения операций обмена, третьи – готовы к исполнению и ожидают предоставления ресурсов ЦП. При этом происходит завершение выполнявшихся процессов и ввод новых, это приводит к возникновению проблемы фрагментации ОЗУ. Суть ее следующая. При размещении новых программ/процессов в ОЗУ ЭВМ (для их мультипрограммной обработки) образуются свободные фрагменты ОЗУ между программами/процессами. Суммарный объем свободных фрагментов может быть достаточно большим, но, в то же время, размер самого большого свободного фрагмента недостаточно для размещения в нем новой программы/процесса. В этой ситуации возможна деградация системы – в системе имеются незанятые ресурсы ОЗУ, но они не могут быть использованы. Путь решения этой проблемы – использование более развитых механизмов организации ОЗУ и виртуальной памяти, позволяющие отображать виртуальное адресное пространство программы/процесса не в одну непрерывную область физической памяти, а в некоторую совокупность областей.

Организация страничной памяти

Страничная организация памяти предполагает разделение всего пространства ОЗУ на блоки одинакового размера – страницы. Обычно размер страницы равен 2^k. В этом случае адрес, используемый в данной ЭВМ, будет иметь следующую структуру:


0
2k-1
2k
2k+1
0я страница
1я страница
...
k
k-1
0
Номер страницы
Номер в странице

БИЛЕТ 15

Пример организации страничной виртуальной памяти

Взять 14 билет.

Модельная (упрощенная) схема организации функционирования страничной памяти ЭВМ следующая: Пусть одна система команд ЭВМ позволяет адресовать и использовать m страниц размером 2^k каждая. То есть виртуальное адресное пространство программы/процесса может использовать для адресации команд и данных до m страниц.

Физическое адресное пространство, в общем случае может иметь произвольное число физических страниц (их может быть больше m, а может быть и меньше). Соответственно структура исполнительного физического адреса будет отличаться от структуры исполнительного виртуального адреса за счет размера поля ”номер страницы”.

В виртуальном адресе размер поля определяется максимальным числом виртуальных страниц – m.

В физическом адресе – максимально возможным количеством физических страниц, которые могут быть подключены к данной ЭВМ (это также фиксированная аппаратная характеристика ЭВМ).

В ЦП ЭВМ имеется аппаратная таблица страниц (иногда таблица приписки) следующей структуры:

0	α_0
1	α_1
2	α_2
3	α_3
...	
i	α_i
...	
m-1	α_{m-1}

Таблица содержит m строк. Содержимое таблицы определяет соответствие виртуальной памяти физической для выполняющейся в данный момент программы/процесса. Соответствие определяется следующим образом: i-я строка таблицы соответствует i-й виртуальной странице.

Содержимое строки α_i определяет, чему соответствует i-я виртуальная страница программы/процесса. Если $\alpha_i \geq 0$, то это означает, что α_i есть номер физической страницы, которая соответствует виртуальной странице программы/процесса. Если $\alpha_i = -1$, то это означает, что для i-й виртуальной страницы нет соответствия физической странице ОЗУ (обработка этой ситуации ниже).

Итак, рассмотрим последовательность действий при использовании аппарата виртуальной страничной памяти.

1. При выполнении очередной команды схемы управления ЦП вычисляют некоторый адрес операнда (операндов) $A_{исп}$. Это виртуальный исполнительный адрес.
2. Из $A_{исп.}$ Выделяются значимые поля номер страницы (номер виртуальной страницы). По этому значению происходит индексация и доступ к соответствующей строке таблицы страниц.

3. Если значение строки ≥ 0, то происходит замена содержимого поля номер страницы на соответствующее значение строки таблицы, таким образом, получается физический адрес. И далее ЦП осуществляет работу с физическим адресом.
4. Если значение строки таблицы равно –1 это означает, что полученный виртуальный адрес не размещен в ОЗУ. Причины такой ситуации? Их две. Первая – данная виртуальная страница отсутствует в перечне станиц, доступных для программы/процесса, то есть имеет место попытка обращения в “ чужую”, не легитимную память. Вторая ситуация, когда операционная система в целях оптимизации использования ОЗУ, откачала некоторые страницы программы/процесса в ВЗУ(свопинг, при действиях ОС при свопинге позднее). Что происходит в системе, если значение строки таблицы страниц –1, и мы обратились к этой строке? Происходит прерывание “ защита памяти”, управление передается операционной системе (по стандартной схеме обработки прерывания и далее происходит программная обработка ситуации (обращаем внимание, что все, что выполнялось до сих пор – пункт 1, 2, 3 и 4 – это действия аппаратуры, без какого-либо участия программного обеспечения).
   ОС по содержимому внутренних данных определяет конечную причину данного прерывания: или это действительно защита памяти, или мы пытались обратиться к странице ОЗУ, которая временно размещена во внешней памяти.

Таким образом, предложенная модель организации виртуальной памяти позволяет решить проблему фрагментации ОЗУ. На самом деле, некоторая фрагментация остается (если в странице занят хотя бы 1 байт, то занята вся страница), но она является контролируемой и не оказывает значительного влияния на производительность системы.

Далее, данная схема позволяет простыми средствами организовать защиту памяти, а также своппирование страниц.

Предложенная модель организации виртуальной памяти позволяет иметь отображение виртуального адресного пространства программы/процесса в произвольные физические адреса, также позволяет выполнять в системе программы/процессы, размещенные в ОЗУ частично (оставшаяся часть может быть размещена во внешней памяти).

Недостаток – необходимость наличия в ЦП аппаратной таблицы значительных размеров.

Итак мы рассмотрели модельный, упрощенный вариант организации виртуальной памяти. Реальные решения используемые в различных архитектурах ЭВМ могут быть гораздо сложнее, но основные идеи остаются неизменными.

Билет 16. ***Многомашинные, многопроцессорные ассоциации*****. Классификация. Примеры.**

Системы с распределенной памятью – MPP.

Примером системы с распределенной памятью может служить **массивно-параллельная архитектура** – **MPP**[1]. Массивно-параллельные системы состоят из однородных вычислительных узлов, каждый из которых включает в себя:

- один или несколько процессоров
- локальную память, прямой доступ к которой с других узлов невозможен
- коммуникационный процессор или сетевой адаптер
- устройства ввода/вывода

Схема MPP системы, где каждый вычислительный узел (ВУ) имеет один процессорный элемент (например, RISC-процессор, одинаковый для всех ВУ), память и коммуникационное устройство, изображена на рисунке.

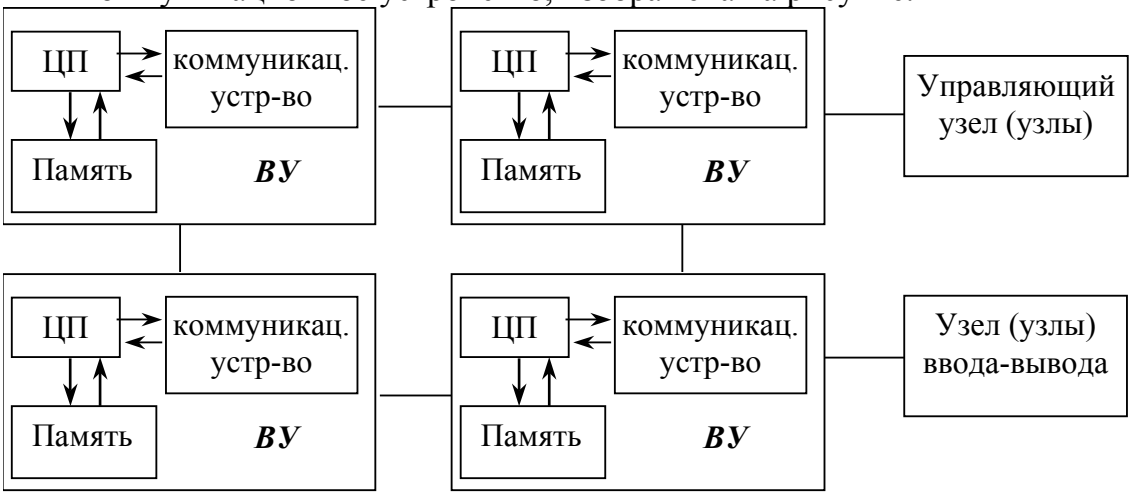


Рис. 1 Архитектура MPP.

Помимо вычислительных узлов, в систему могут входить специальные узлы ввода-вывода и управляющие узлы. Узлы связаны между собой посредством высокоскоростной среды передачи данных определенной топологии. Число процессоров в MPP-системах может достигать нескольких тысяч.

Поскольку в MPP-системе каждый узел представляет собой относительно самостоятельную единицу, то, как правило, управление массивно-параллельной системой в целом осуществляется одним из двух способов:

1. На каждом узле может работать полноценная операционная система, функционирующая отдельно от других узлов. При этом, разумеется, такая ОС должна поддерживать возможность коммуникации с другими узлами в соответствии с особенностями данной архитектуры.
2. «Полноценная» ОС работает только на управляющей машине, а на каждом из узлов MPP-системы работает некоторый сильно «урезанный» вариант ОС, обеспечивающий работу задач на данном узле.

В массивно-параллельной архитектуре отсутствует возможность осуществлять обмен данными между ВУ напрямую через память, поэтому

[1] Аббревиатура MPP представляет собой сокращение от «Massive Parallel Processing»

взаимодействие между процессорами реализуется с помощью аппаратно и программно поддерживаемого механизма передачи сообщений между ВУ. Соответственно, и программы для MPP-систем обычно создаются в рамках модели передачи сообщений.

Системы с общей памятью – SMP.

В качестве наиболее распространенного примера систем с общей памятью рассмотрим архитектуру **SMP**[2] – **симметричную мультипроцессорную систему**. SMP-системы состоят из нескольких **однородных** процессоров и массива общей памяти, который обычно состоит из нескольких независимых блоков. Слово «симметричный» в названии данной архитектуры указывает на то, что все процессоры имеют доступ напрямую (т.е. возможность адресации) к любой точке памяти, причем доступ любого процессора ко всем ячейкам памяти осуществляется с одинаковой скоростью. Общая схема SMP-архитектуры изображена на Рис. 2.

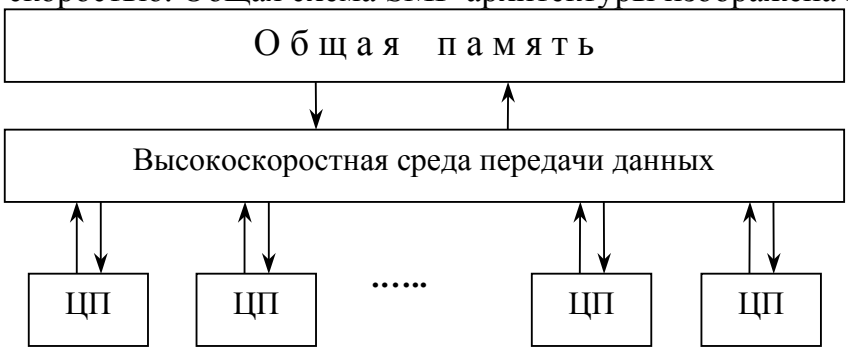


Рис. 2 Архитектура SMP

Процессоры подключены к памяти либо с помощью общей шины, либо с помощью коммутатора. Отметим, что в любой системе с общей памятью возникает проблема кэширования: так как к некоторой ячейке общей памяти имеет возможность обратиться каждый из процессоров, то вполне возможна ситуация, когда некоторое значение из этой ячейки памяти находится в кэше одного или нескольких процессоров, в то время как другой процессор изменяет значение по данному адресу. В этом случае, очевидно, значения, находящиеся в кэшах других процессоров, больше не могут быть использованы и должны быть обновлены. В SMP-архитектурах обычно согласованность данных в кэшах поддерживается аппаратно.

Очевидно, что наличие общей памяти в SMP-архитектурах позволяет эффективно организовать обмен данными между задачами, выполняющимися на разных процессорах, с использованием механизма разделяемой памяти. Однако сложность организации симметричного доступа к памяти и поддержания согласованности кэшей накладывает существенное ограничение на количество процессоров в таких системах – в реальности их число обычно не превышает 32 – в то время, как стоимость таких машин весьма велика. Некоторым компромиссом между масштабируемостью и однородностью доступа к памяти являются NUMA-архитектуры, которые мы рассмотрим далее.

[2] Аббревиатура SMP является сокращением фразы «Symmetric Multi Processing»

Системы с неоднородным доступом к памяти – NUMA.

Системы с неоднородным доступом к памяти (NUMA[3]) представляют собой промежуточный класс между системами с общей и распределенной памятью. Память в NUMA-системах является физически распределенной, но логически общедоступной. Это означает, что каждый процессор может адресовать как свою локальную память, так и память, находящуюся на других узлах, однако время доступа к удаленным ячейкам памяти будет в несколько раз больше, нежели время доступа к локальной памяти. Заметим, что единой адресное пространство и доступ к удаленной памяти поддерживаются аппаратно. Обычно аппаратно поддерживается и когерентность (согласованность) кэшей во всей системе

Системы с неоднородным доступом к памяти строятся из однородных базовых модулей, каждый из которых содержит небольшое число процессоров и блок памяти. Модули объединены между собой с помощью высокоскоростного коммутатора. Обычно вся система работает под управлением единой ОС. Поскольку логически программисту предоставляется абстракция общей памяти, то модель программирования, используемая в системах NUMA, обычно в известной степени аналогична той, что используется на симметричных мультипроцессорных системах, и организация межпроцессного взаимодействия опирается на использование разделяемой памяти.

Масштабируемость NUMA-систем ограничивается объемом адресного пространства, возможностями аппаратуры поддержки когерентности кэшей и возможностями операционной системы по управлению большим числом процессоров.

Кластерные системы.

Отдельным подклассом систем с распределенной памятью являются **кластерные системы,** которые представляют собой некоторый аналог массивно-параллельных систем, в котором в качестве ВУ выступают обычные рабочие станции общего назначения, причем иногда узлы кластера могут даже одновременно использоваться в качестве пользовательских рабочих станций. Кластер, объединяющий компьютеры разной мощности или разной архитектуры, называют **гетерогенным** (неоднородным). Для связи узлов используется одна из стандартных сетевых технологий, например, Fast Ethernet.

Главными преимуществами кластерных систем, благодаря которым они приобретают все большую популярность, являются их относительная дешевизна, возможность масштабирования и возможность использования при построении кластера тех вычислительных мощностей, которые уже имеются в распоряжении той или иной организации.

При программировании для кластерных систем, как и для других систем с распределенной памятью, используется модель передачи сообщений.

---

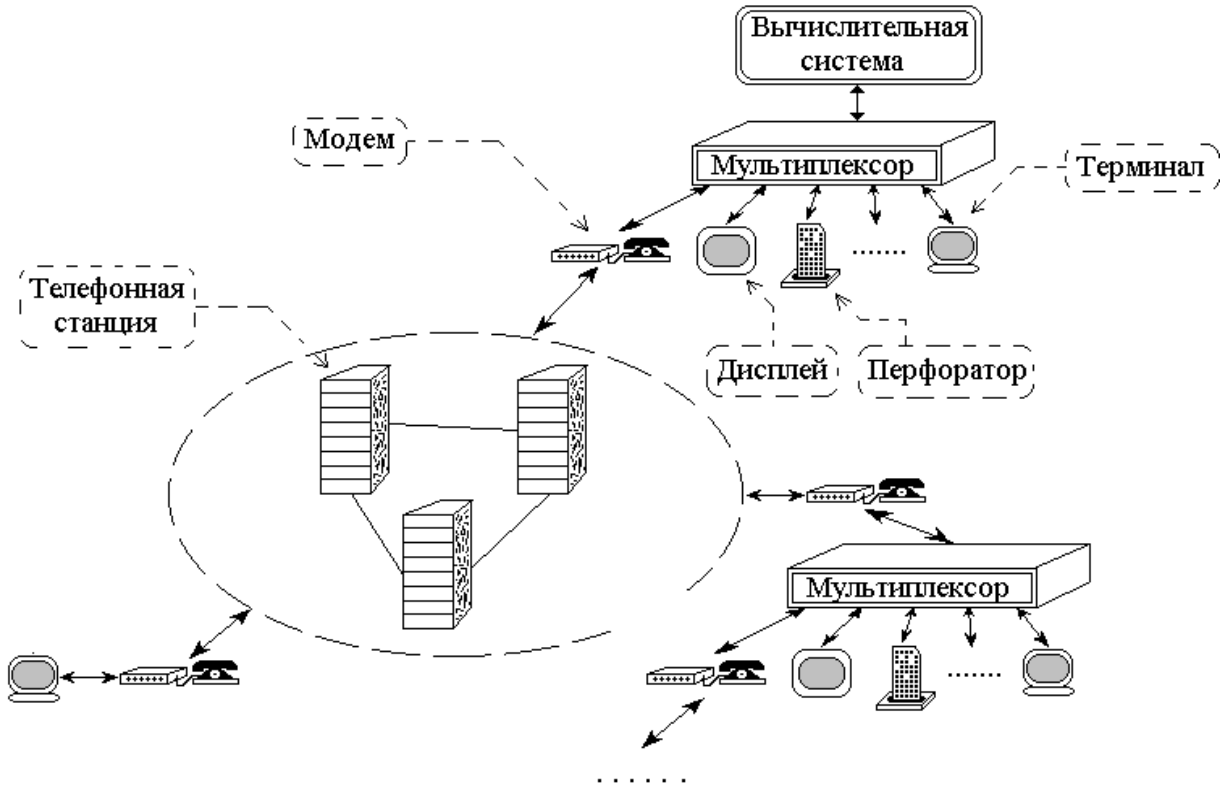
[3] Аббревиатура NUMA расшифровывается как «Non-Unified Memory Access», что в буквальном переводе и означает «неоднородный доступ к памяти». Часто используется также обозначение «ccNUMA», что означает «cache-coherent NUMA», или система с неоднородным доступом к памяти с поддержкой когерентности (т.е. согласованности) кэшей

БИЛЕТ 17. Терминальные комплексы. Компьютерные сети.

Терминальные комплексы.

Исторически, одним из первых примеров многомашинных ассоциаций являлись терминальные комплексы. ***Терминальный комплекс*** это многомашинная ассоциация предназначенная для организации массового доступа удаленных и локальных пользователей к ресурсам некоторой вычислительной системы. При этом, к примеру, возможно использование терминальных комплексов для сбора и централизованной обработки информации (например, обработка результатов переписи населения или выборов) или для массового доступа удаленных пользователей к информации, размещенной в вычислительной системе (например, доступ пользователей к электронной библиотеке или система бронирования и продажи авиа или железнодорожных билетов). Временем появления подобных задач является конец 50-х – начало 60-х годов 20 века.

Структуру терминального комплекса можно примерно изобразить следующим образом.



Терминальный комплекс может включать в свой состав:

- ***основную вычислительную систему*** – систему, массовый доступ к ресурсам которой обеспечивается терминальным комплексом;
- ***локальные мультиплексоры*** – аппаратные комплексы, предназначенные для осуществление связи и взаимодействия вычислительной системы с несколькими устройствами через один канал ввода/вывода, в общем случае возможна схема M x N, где M – число обслуживаемых мультиплексором устройств, N число используемых для организации работы каналов ввода/вывода (M > N);
- ***локальные терминалы*** – оконечные устройства, используемые для взаимодействия пользователей с вычислительной системой (это могут быть алфавитно-цифровые терминалы, графические

терминалы, устройства печати, вычислительные машины, эмулирующие работу терминалов и т.п.) и, подключаемые к вычислительной системе непосредственно через каналы ввода/вывода или через локальные мультиплексоры;

- ***модемы*** – устройства, предназначенные для организации взаимодействия вычислительной системы с удаленными терминалами с использованием телефонной сети. В функцию модема входит преобразование информации из дискретного, цифрового представления, используемого в вычислительной технике в аналоговое представление, используемое в телефонии и обратно (в общем случае модем это устройство, предназначенное для взаимного преобразования данных из различных форм представления, например, могут быть оптические модемы, преобразующие данные из цифрового формата в оптический, предназначенный для передачи по оптоволоконным линиям связи). Со стороны вычислительной системы модем подключается либо через канал ввода/вывода, либо через мультиплексор.
- ***удаленные терминалы*** – терминалы, имеющие доступ к вычислительной системе с использованием телефонных линий связи и модемов.
- ***удаленные мультиплексоры*** – мультиплексоры, подключенные к вычислительной системе с использованием телефонных линий связи и модемов.

Телефонная сеть состоит из набора телефонных станций, объединенных друг с другом линиями связи. Связь абонентов телефонной в том числе и связь удаленных терминалов с вычислительной системой осуществляется с использованием ***коммутируемого канала,*** либо по ***выделенным каналам***. Суть соединения через коммутируемый канал заключается в том, что при нескольких звонках к одному и тому же абоненту, раз от раза маршруты коммутации (т.е. набор проводов, по которым идет сообщение) отличаются друг от друга, за счет того, что каждый раз выбираются свободные каналы в телефонных станциях по пути соединения. После завершения сеанса связи между абонентами коммутируемый канал освобождается. При использовании выделенного канала маршрут коммутации между абонентами фиксируется на период аренды выделенного канала. Достоинства/недостатки использования коммутируемых и выделенных каналов очевидны.

Линия связи, которая связывает один удаленный терминал с компьютером, называется линией связи типа ***точка-точка***. Таким образом эта линия может быть либо выделенной (мы договариваемся с телефонными станциями и фиксируем коммутацию), либо коммутируемой.

Канал может быть ***многоточечным***. При этом на входе находится удаленный мультиплексор. Многоточечные каналы также могут быть либо выделенными, либо коммутируемыми.

С точки зрения организации потоков информации можно выделить следующие разновидности каналов.

1. ***Симплексные каналы*** - каналы, по которым передача информации ведется в одном направлении (например, телевизионный канал –

обеспечивает передачу информации только в одном направлении от передающей антенны к принимающей).

2. ***Дуплексные каналы*** - каналы, которые обеспечивают одновременную передачу информации в двух направлениях (например, телефонный разговор, мы одновременно можем и говорить и слушать).
3. ***Полудуплексные каналы*** - каналы, которые обеспечивают передачу информации в двух направлениях, но в каждый момент времени только в одну сторону (подобно рации).

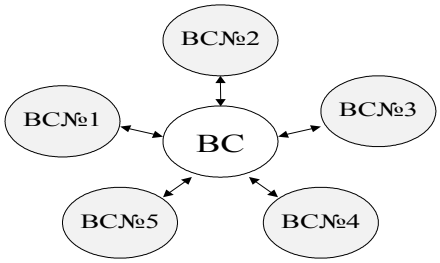
Одним из примеров терминального комплекса может быть система NASDAQ (National Association of Securities Dealers Automated Quatation), построенная в 60-70-х годах 20 века и предназначенная для сбора и передачи сообщений о курсах акций на бирже. Система была построена на использовании мощной, по тем временам, вычислительной машины Univac-1108 и значительного числа терминалов (несколько тысяч), установленных в биржевых конторах по всей территории США.

Многомашинные вычислительные комплексы

Многомашинные вычислительные комплексы (ММВК) - это программно аппаратное объединение группы вычислительных машин, в которых:

1. На каждой из машин работает своя операционная система (этот признак отличает ММВК от многопроцессорного вычислительного комплекса).
2. В ММВК имеются общие физические ресурсы, например ОЗУ, ВЗУ или общие каналы связи (а, следовательно, имеются проблемы синхронизации доступа).

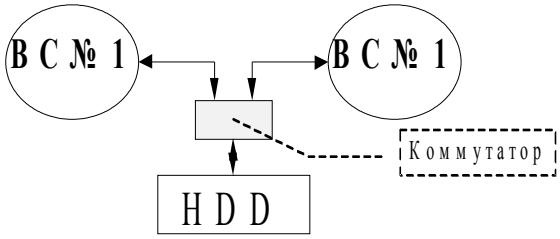
ММВК использовались в качестве систем сбора и обработки больших наборов данных, и для организации глобальных терминальных комплексов. ММВК появились в начале 60-х и сейчас продолжают успешно существовать. Одно из основных применений ММВК - это дублирование вычислительной мощи, примером таких систем может служить любая система управления важными технологическими процессами.



В ММВК общий ресурс является общим не только для всех ВС, но и для групп ВС, благодаря этому мы можем организовывать ММВК сложной структуры, необходимой для решения конкретной проблемы (Например, ММВК для продажи авиабилетов и ММВК для параллельного проведения какого-нибудь сложного научного расчета). Мы также обсудили тот факт, что в ММВК на каждой из машин работает своя операционная система. Отсюда вытекает, что все проблемы

взаимодействия должны решаться на уровне взаимодействия ОС. Система, аналогичная ММВК, но в которой работает одна ОС, - многопроцессорная ВС. Существуют задачи, для которых не хватает средств, предоставляемых терминальными комплексами. Это, например, проблема организации больших баз данных. В этом случае используют ММВК.

В ММВК имеется проблема синхронизации доступа к разделяемым ресурсом. Разделяемыми ресурсами могут быть устройства внешней памяти, ОЗУ, каналы связи, соединенные двумя или более компонентами вычислительного комплекса. Рассмотрим такой пример. У нас есть ММВК, состоящий из двух ВС. Разделяемый ресурс - жесткий диск. Проблема в данном случае явно формулируется так: «Нужно научить две ВС синхронизированно обмениваться с HDD.» Т. е. если программа одной ВС что-то пишет на HDD, то область данных, в которую она пишет или весь HDD должны быть заблокированы для другой ВС (Проблема напоминает проблему семафоров). Одно из решений - коммутатор HDD, некий контроллер, который имеет команду, блокирующую HDD. При начале обмена одной вычислительной системы доступ к HDD заблокирован для других ВС. А эта ВС в монопольном режиме использует HDD. Если другая ВС попытается начать обмен с HDD возможны два решения:



1) синхронное ожидание;
2) асинхронное ожидание (система не будет простаивать, она временно остановит процесс, подавший заказ на обмен и активизирует другой процесс).

На самом деле коммутаторы, конечно, более интеллектуальны. Они, например, устанавливают блокировку не на весь HDD, а только на некоторые его блоки.

Приведенное выше решение привлекает своей технической простотой как с аппаратной точки зрения, так и с точки зрения программной реализации (нет сложных взаимосвязей), но оно имеет существенный недостаток. ВС может заблокировать HDD и после этого зациклиться. Для борьбы с такими ситуациями можно использовать различные устройства, отличные от коммутатора HDD, позволяющие послать сигнал от одной ВС к другой. Это может быть, например, низкоскоростной канал связи (скорость передачи нам здесь не нужна).

БИЛЕТ 18

Базовые понятия, определения, структура

Операционная система – это комплекс программ, обеспечивающий контроль за существованием (некоторые из ресурсов ВС, как мы знаем, являются программными или логическими/виртуальными и создаются под контролем операционной системой), распределением и использованием ресурсов ВС.

Любая ОС оперирует некоторым набором базовых сущностей (понятий) на основе которых строится логика функционирования системы. Например, подобными базовыми понятиями могут быть задача, задание, процесс, набор данных, файл, объект.

Одним из наиболее распространенных базовых понятий ОС является ***процесс***.

Интуитивно определение процесса достаточно просто, но определить процесс строго, формально, достаточно сложно. Поэтому существует целый ряд определений процесса, многие из которых системно-ориентированы.

Процесс – это совокупность машинных команд и данных, исполняющаяся в рамках ВС и обладающая правами на владение некоторым набором ресурсов. Эти права могут быть эксклюзивными, когда ресурс принадлежит только этому процессу. Некоторые из ресурсов могут разделяться, т. е. одновременно принадлежать двум и более процессам, в этом случае мы говорим о ***разделяемых ресурсах***.

Возможно два варианта выделения ресурсов процессу:

- предварительная декларация использования тех или иных ресурсов (до начала выполнения процесса в систему передается перечень ресурсов, которые будут использованы процессом);
- Динамическое пополнение списка принадлежащих процессу ресурсов по ходу выполнения процесса при непосредственном обращении к ресурсу.

Реальная схема зависит от конкретной ОС. На практике возможно использование комбинации этих вариантов. Для простоты изложения будем считать, что модельная ОС имеет возможность предварительной декларации ресурсов, которые будут использованы процессом.

Любая ОС должна удовлетворять следующим свойствам:

- надежность
- защита
- эффективность
- предсказуемость

Типовая структура ОС.

Ядро – резидентная часть ОС, работающая в режиме супервизора. В ядре размещаются программы обработки прерываний и драйверы наиболее «ответственных» устройств. Это могут быть и физические, и виртуальные устройства. Например, в ядре могут располагаться драйверы файловой системы, ОЗУ. Обычно ядро работает в режиме физической адресации.

Следующие уровни структуры – динамически подгружаемые драйверы физических и виртуальных устройств. Это драйверы, добавление которых в систему возможно «на ходу» без перекомпоновки программ ОС. Они могут являться резидентными и нерезидентными, а также могут работать как в режиме супервизора, так и в пользовательском режиме.

Можно выделить следующие основные логические функции ОС:

- управление процессами;
- управление ОП;
- планирование;
- управление устройствами и ФС.

Билет 19 Типы операционных систем

Пакетная ОС

Пакет программ – совокупность программ, для выполнения каждого из которых требуется некоторое время работы процессора. Этот тип был на первых компьютерах. Пакет программ – стопка перфокарт.

Стратегия переключения с одного процесса на другой, если

а)выполняемый процесс завершен
б)возникло прерывание по обмену в выполняемом программе
в)зафиксировался факт зацикливания.

Системы разделения времени

Квант времени ЦП – некоторый фиксированный ОС промежуток времени работы ЦП
ЦП предоставляется процессу на один квант времени. Меняя размер кванта можно получить различные характеристики ОС. Большой квант времени удобен для отладки.
Если квант времени устремить к нулю, то у пользователя создается впечатление, что он работает один на этой ОС. Это происходит потому, что критерий эффективности с точки зрения человека – через сколько компьютер реагирует на действия человека.
Переключение выполнения процессов происходит только в одном из случаев:
- Исчерпался выделенный квант времени
- Выполнение процесса завершено
- Возникло прерывание
- Был фиксирован факт зацикливания процесса

Системы реального времени

являются специализированными системами в которых все функции планирования ориентированы на обработку некоторых событий за время, не превосходящее некоторого предельного значение

Критерий качества – обработка любого события за некоторый гарантированный промежуток времени (бортовой компьютер, автопилот…)

Реально (за исключением систем реального времени, которые могут быть разные по областям применения, важности серьезности и т.д.) используются комбинации пакетных и систем разделения времени друг в друге и с различными стратегиями

Сетевые, распределенные ОС

Сетевая ОС –

Мы имеем физическую сеть в которой подключенные компьютеры взаимодействуют с помощью протоколов, сетевая ОС предоставляет пользователям распределенные прикладные приложения.

Распределенная ОС –

Состоит из ядра, локализованного в рамках одного компьютера, и остальных функций распределенных по компьютерам сети.
Проблема распределения файловой системы

Билет 20 Модель организации взаимодействия в сети ISO/OSI

Необходима аппаратная стандартизация. Предложена модель семиуровневого взаимодействия в сетях.

Основные проблемы: 1. Стандартизация программного обеспечения, устройств и т.д. С развитием сетей та проблема увеличивалась. Сети создавались как корпоративные, локальные. Каждое решение было уникальным. (каналы связи, формат передаваемой информации, программный интерфейс), следовательно перенос сетевой программы с одного компьютера на другой был невозможен, либо сильно затруднен. Т.к. мир существует на объединении и разделении предприятий, это было очень неудобно. Возникла необходимость стандартизации.
OSI – системы открытых интерфейсов.
1..7 – все возможные уровни взаимодействия компьютеров в сети
Каждый уровень использует логически целостный набор действий и форматов данных, предназначенных для передачи информации между взаимодействующими в сети ВС. регламентации программных или аппаратных средств.
В каждом уровне модель ISO/OSI предполагает наличие некоторого количества протоколов, каждый из которых может осуществлять взаимодействие с одноименным протоколом на другой взаимодействующей машине (возможно виртуальной).

1.Физический уровень На этом уровне однозначно определяется физическая сфера передачи данных и форматы передаваемых сигналов. На этом уровне решаются вопросы взаимосвязи в терминах сигналов. Этот уровень однозначно определяется физической средой, используемой для передачи данных и отвечает за организацию физической связи между устройствами и передачи данных в сети.
2. Канальный уровень Обеспечивает управление доступом к физической среде передачи данных, в частности обеспечение синхронизации передачи данных. Формализуются правила передачи данных.Решаются задачи обнаружения и синхронизации ошибок.
3.Сетевой уровень Решается вопрос управления связью между взаимодействующими компьютерами. Решается задача
маршрутизации. и адресацией в сети.
4. Транспортный уровень (уровень логического канала) Решаются проблемы управления и передачи данных локализация и обработка ошибок, сервис передачи данных.
5.Сеансовый уровень Управление сеансами связи. Синхронизация отправки и приема данных.Управление подтверждением полномочий. Обработка внештатных ситуаций. прерывания/продолжения работы в тех или иных внештатных ситуациях, управление подтверждением полномочий (паролей).
6.Представительский уровень Разрешается проблема унификации кодировок. Уровень представления данных. На этом уровне находятся протоколы,

реализующие единые соглашения перевода из внутреннего представления данных конкретной машины в сетевое и обратно.

7. **Прикладной уровень** Осуществляет стандартизацию взаимодействия с прикладными системами.

Основные понятия

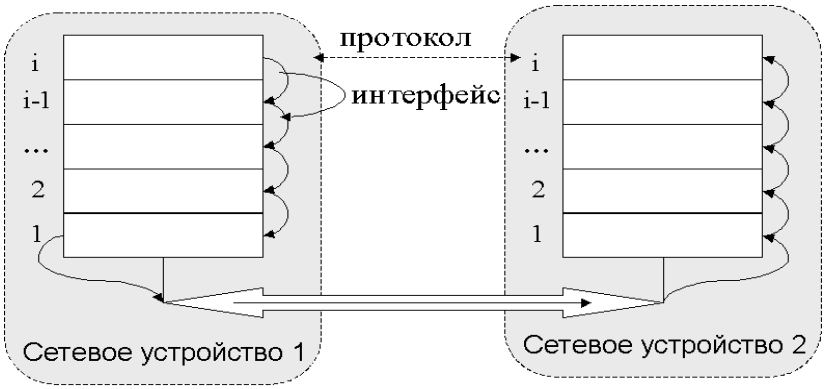
Протокол – формальное описание сообщений и правил, по которым сетевые устройства (вычислительные системы) осуществляют обмен информацией.
или
Правила взаимодействия одноименных уровней.

Интерфейс – правила взаимодействия вышестоящего уровня с нижестоящим.

Служба или сервис – набор операций, предоставляемых нижестоящим уровнем вышестоящему.

Стек протоколов – перечень разноуровневых протоколов, реализованных в системе

Логическое взаимодействие сетевых устройств по i-ому протоколу



Для организации взаимодействия при передаче сообщений от одного уровня к соседнему, существуют стандартизованные соглашения, которые называются **интерфейсами**.

Таким образом, данные от одной прикладной программы до другой прикладной программы в сети проходят путь от уровня протоколов прикладных программ до физического уровня на ВС, отправляющей данные, и далее на ВС, принимающей данные, они проходят этот путь обратном порядке.

Билет 21 Семейство протоколов TCP/IP

Соответствие модели ISO/OSI модели семейства протоколов TCP/IP

Уровень доступа к сети. Стандартизация доступа к сети. Состоит из подпрограмм доступа к физической сети. Модель TCP/IP не разделяет два уровня модели OSI – канальный и физический, а рассматривает их как единое целое.

Межсетевой уровень Работает с дейтаграммами, адресами, выполняет маршрутизацию и «прикрывает» транспортный уровень от общения с физической сетью. Однако, в отличие от сетевого уровня модели OSI, этот уровень не устанавливает соединений с другими машинами.

Протоколы уровня доступа к сети используют при передаче и приеме данных пакеты, называемые ***фреймами.*** На межсетевом уровне используются ***дейтаграммы***

3. Транспортный уровень. Обеспечивает доставку данных от компьютера к компьютеру, обеспечивает средства для поддержки логических соединений между прикладными программами. В отличие от транспортного уровня модели OSI, в функции транспортного уровня TCP/IP не всегда входят контроль за ошибками и их коррекция. TCP/IP предоставляет два разных сервиса передачи данных на этом уровне**.**

Уровень транспортных протоколов семейства представляется двумя протоколами TCP и UDP. Протокол TCP оперирует ***сегментами***. UDP – ***пакетами***. На уровне прикладных программ, системы построенные на использовании протокола TCP используют ***поток*** данных, а системы использующие UDP - ***сообщения***.

4. **Уровень прикладных программ** Состоит из прикладных программ и процессов, использующих сеть и доступных пользователю. В отличие от модели OSI, прикладные программы сами стандартизуют представление данных

Свойства протоколов семейства TCP/IP

•Открытые стандарты протоколов, которые поддерживаются почти всеми операционными средами и вычислительными платформами независимо от аппаратного обеспечения сети аппаратных данных.
• независимость от аппаратного обеспечения сети передачи данных
• обладает уникальным системным именованием сетевых устройств, что позволяет любому устройству единым образом адресоваться в этой сети.
• Стандартизованные протоколы прикладных программ

Взаимодействие между уровнями протоколов TCP/IP

Уровень доступа к сети. Протоколы на этом уровне обеспечивают систему средствами для передачи данных другим устройствам в сети. Они определяют, как использовать сеть для передачи дейтаграмм IP. В отличие от протоколов более высоких уровней, протоколы этого уровня должны знать детали физической сети (структуру пакетов, систему адресации и т.д.), чтобы правильно оформить передаваемые данные.

Межсетевой уровень. Протокол IP

•Функции протокола IP

- формирование дейтаграмм
- поддержание системы адресации
- обмен данными между транспортным уровнем и уровнем доступа к сети
- организация маршрутизации дейтаграмм
- разбиение и обратная сборка дейтаграмм

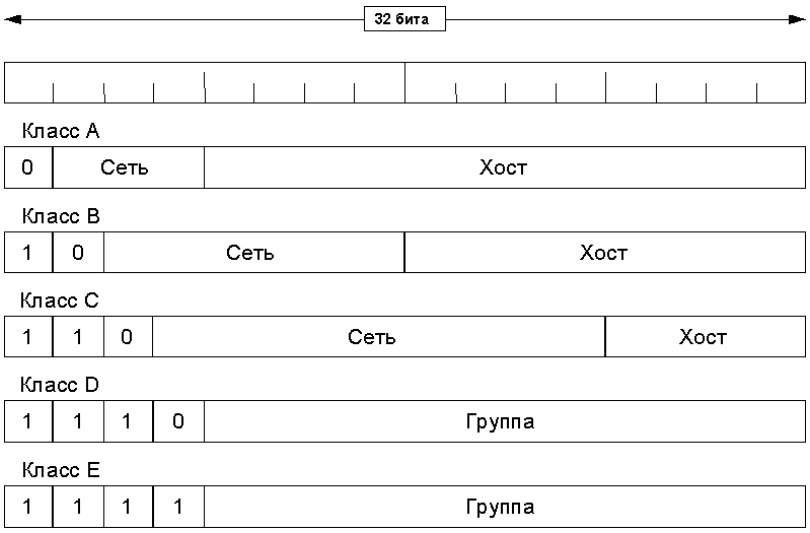
IP является протоколом ***без логического установления соединения***. Это значит, что он не обменивается контрольной информацией для установки соединения, перед началом передачи данных. IP оставляет другим протоколам право устанавливать соединения – этим занимается либо протокол TCP, либо сами прикладные программы.

Протокол IP **не обеспечивает обнаружение и исправление ошибок**

Одним из основных свойств протокола IP является система адресации, которая обеспечивает уникальное именование любого ***сетевого устройства.***

Устройство будем считать **сетевым**, если с ним ассоциирован некоторый стек протоколов)

Система адресации протокола IP



IP адрес представляется последовательностью четырех байтов. В адресе кодируется уникальный номер сети, а также номер компьютера (сетевого устройства в сети).

. Для представление содержимого IP адреса используется последовательность цифр:

N1.N2.N3.N4 ,

где Ni – десятичное представление содержимого i – го байта адреса.

Типы адресов

A номер сети <=126, уникальные сети, которые исторически принадлежат крупным мировым корпорациям.

C самые распространенные.

Некоторые из IP адресов являются зарезервированными, т.е. их интерпретация отличается от стандартной.

Протоколы TCP/IP были созданы для передачи данных через ARPANET, которая является сетью с коммутацией пакетов.

Пакет – это блок данных, который передаётся вместе с информацией, необходимой для его корректной доставки. Каждый пакет перемещается по сети независимо от остальных.

Дейтаграмма – это пакет протокола IP. Контрольная информация занимает первые пять или шесть 32-битных слов дейтаграммы. Это её заголовок (header). По умолчанию, его длина равна пяти словам, шестое является дополнительным. Для указания точной длины заголовка в нём есть специальное поле – *длина заголовка* (IHL, Internal Header Length).

Шлюз – устройство, передающее пакеты между различными сетями

Маршрутизация – процесс выбора шлюза или маршрутизатора

Шлюз – компьютер, который имеет >=2 сетевых адаптеров (каждый имеет свой IP адрес)

Компьютерные системы могут передавать данные только внутри той сети, к которой они подключены. Поэтому передача дейтаграмм из одной сети в другую идёт через шлюзы – от одного к другому. Внутри хоста данные проходят пути от уровня прикладных программ до уровня доступа к сети (и обратно). Дейтаграммы, которые переправляет шлюз, поднимаются только до межсетевого уровня. На этом уровне протокол IP, узнавая адрес получателя данных (на протяжении всего пути следования этот адрес не меняется – меняются промежуточные машины), принимает решение отправить дейтаграмму в одну из сетей, к которым подключен.

На рисунке выше показано, как используются шлюзы для ретрансляции пакетов.

Транспортный уровень

Протокол контроля передачи (TCP, Transmission Control Protocol) - обеспечивает надежную доставку данных с обнаружением и исправлением ошибок и с установлением логического соединения.

Протокол пользовательских дейтаграмм (UDP, User Datagram Protocol) - отправляет пакеты с данными, «не заботясь» об их доставке.

TCP Надежная передача данных. При отправке TCP пакета идет подтверждение о получении. Подтверждение должно прийти за некоторое детерминированное время. Если не пришло, то считается, что пакет потерялся. Обеспечивается порядок приема и передачи сообщений.

UDP Не требует подтверждения о доставки пакета.

TCP лучше, но за это мы платим содержательной скоростью и нагрузкой на сеть.

UDP быстрее, т.к. меньше мусора пересылается.

Выводы

UDP лучше для локальной сети, а TCP для межсетевого взаимодействия.

Уровень прикладных программ

На самой вершине архитектуры семейства протоколов TCP находится **уровень прикладных программ**.

Все процессы этого уровня пользуются протоколами транспортного уровня для обмена данными по сети.

Протоколы, опирающиеся на TCP

TELNET (Network Terminal Protocol), Протокол сетевого терминала, разработан для удаленного доступа к компьютерам сети (remote login).

FTP (File Transfer Protocol), Протокол передачи файлов, используется для интерактивной передачи файлов между компьютерами сети.

SMTP (Simple Mail Transfer Protocol), Простой протокол передачи почты. Основной протокол для обмена почтой, использующийся в Internet.

Протоколы, опирающиеся на UDP

DNS (Domain Name Service), Служба имен доменов, или просто Служба именования. С помощью этого протокола устанавливается взаимно однозначное соответствие между IP-адресами сетевых устройств и их именами.

RIP (Routing Information Protocol), Протокол информации о маршрутизации. Маршрутизация данных (поиск путей их передачи от хоста-отправителя к хосту-получателю) в Internet является одной из важнейших функций семейства протоколов

TCP/IP. RIP используется сетевыми устройствами для обмена информацией о маршрутизации.

NFS (Network File System), сетевая файловая система. С помощью этого протокола компьютеры могут совместно использовать файлы, разбросанные по сети.

1. БИЛЕТ 22 *Управление процессами*. Определение процесса, типы. Жизненный цикл, состояния процесса. Свопинг. Модели жизненного цикла процесса. Контекст процесса.

Рассмотрим типовые этапы обработки процесса в системе, совокупность этих этапов будем назвать ***жизненным циклом процесса*** в системе. Традиционно, жизненный цикл процесса содержит этапы:

- образование (порождение) процесса;
- обработка (выполнение) процесса;
- ожидание (по тем или иным причинам) постановки на выполнение;
- завершение процесса.

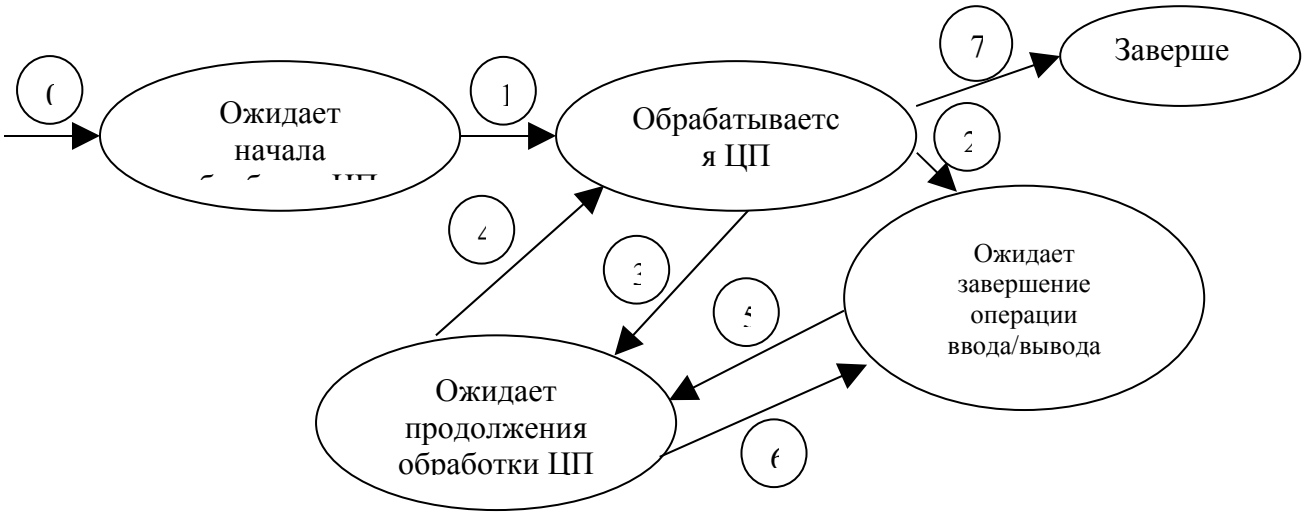
Однако жизненные циклы процессов в реальных системах могут иметь свою, системно-ориентированную совокупность этапов.

Рассмотрим модельную ОС.

Пусть имеется специальный ***буфер ввода процессов*** (БВП) – пространство, в котором размещаются и хранятся сформированные процессы от момента их образования, до момента начала выполнения. На данном этапе происходит формирование всех необходимых структур данных, соответствующих процессу. В частности, на данном этапе ОС формирует информацию о предварительно заказанных данным процессом ресурсах. Основная задача БВП – «подпитка» системы новыми процессами, готовыми к исполнению.

После начала выполнения процесса он попадает в ***буфер обрабатываемых процессов*** (БОП). В данном буфере размещаются все процессы, находящиеся в системе в мультипрограммной обработке.

Обобщенный жизненный цикл процесса можно представить в этом случае ***графом состояний***. Рассмотрим, кратко, переходы процесса из состояния в состояние.



0. После формирования процесс поступает в очередь на начало обработки ЦП (попадает в БВП).
1. В БВП выбирается наиболее приоритетный процесс для начала обработки ЦП (попадает в БОП).
2. Процесс прекращает обработку ЦП по причине ожидания операции в/в, поступает в очередь завершения операции обмена (БОП).
3. Процесс прекращает обработку ЦП, но в любой момент может быть продолжен (например, истек квант времени ЦП, выделенный процессу). Поступает в очередь процессов, ожидающих продолжения выполнения центральным процессором (БОП).
4. Наиболее приоритетный процесс продолжает выполнение ЦП (БОП).
5. Операция обмена завершена и процесс поступает в очередь ожидания продолжения выполнения ЦП (БОП).
6. Переход из очереди готовых к продолжению процессов в очередь процессов, ожидающих завершения обмена (например, ОС откачала содержимое адресного пространства процесса из ОЗУ во внешнюю память) (БОП).
7. Завершение процесса, освобождение системных ресурсов. Корректное завершение работы процесса, разгрузка информационных буферов, освобождение ресурсов (например, реальный вывод информации на устройство печати).

Текущее состояние любого процесса из БОП изменяется во времени в зависимости от самого процесса и состояния ОС. С каждым из процессов из БОП система ассоциирует совокупность данных, характеризующих актуальное состояние процесса – ***контекст процесса***. (в общем случае контекст процесса содержит информацию о текущем состоянии процесса, включая информацию о режимах работы процессора, содержимом регистровой памяти, используемой процессом, системной информации ОС, ассоциированной с данным процессом).

Процессы, находящиеся в одном из состояний ожидания в своих контекстах содержат всю информацию, необходимую для продолжения выполнения - состояние процесса в момент прерывания (копии регистров, режимы ОП, настройки аппарата виртуальной памяти и т. д.). Соответственно при смене выполняемого процесса ОС осуществляет «перенастройку» внутренних ресурсов ЦП, происходит смена контекстов выполняемых процессов.

На этапе выполнения процесса ОС обеспечивает возможность корректного взаимодействия процессов от передачи сигнальных воздействий от процесса к процессу до организации корректной работы с разделяемыми ресурсами.

Итак, контекст процесса может состоять из:

- пользовательской составляющей – состояние программы, как совокупности машинных команд и данных, размещенных в ОЗУ;

системной составляющей – содержимое регистров и режимов работы процессора, настройки аппарата защиты памяти, виртуальной памяти, принадлежащие процессу ресурсы (как физические, так и виртуальные).

Типы процессов

В различных системах используются различные трактовки определения термина ***процесс***. Рассмотрим уточнение понятия процесса.

Полновесные процессы - это процессы, выполняющиеся внутри защищенных участков памяти операционной системы, то есть имеющие собственные виртуальные адресные пространства для статических и динамических данных. В мультипрограммной среде управление такими процессами тесно связано с управлением и защитой памяти, поэтому переключение процессора с выполнения одного процесса на выполнение другого является достаточно дорогой операцией. В дальнейшем, используя термин ***процесс*** будем подразумевать ***полновесный процесс***.

Легковесные процессы, называемые еще как ***нити*** или ***сопрограммы***, не имеют собственных защищенных областей памяти. Они работают в мультипрограммном режиме одновременно с активировавшей их задачей и используют ее виртуальное адресное пространство, в котором им при создании выделяется участок памяти под динамические данные (стек), то есть они могут обладать собственными локальными данными. Нить описывается как обычная функция, которая может использовать статические данные программы. Для одних операционных систем можно сказать, что нити являются некоторым аналогом процесса, а в других нити представляют собой части процессов. Таким образом, обобщая можно сказать – в любой операционной системе понятие «процесс» включает в себя следующее:

- исполняемый код;
- собственное адресное пространство, которое представляет собой совокупность виртуальных адресов, которые может использовать процесс;
- ресурсы системы, которые назначены процессу ОС;
- хотя бы одну выполняемую нить.

При этом подчеркнем – понятие процесса может включать в себя понятие исполняемой нити, т. е. однонитевую организацию – «один процесс – одна нить». В данном случае понятие процесса жестко связано с понятием отдельной и недоступной для других процессов виртуальной памяти. С другой стороны, в процессе может несколько нитей, т. е. процесс может представлять собой многонитевую организацию.

Нить также имеет понятие контекста – это информация, которая необходима ОС для того, чтобы продолжить выполнение прерванной нити. Контекст нити содержит текущее состояние регистров, стеков и индивидуальной области памяти, которая используется подсистемами и библиотеками. Как видно, в данном случае характеристики нити во многом аналогичны характеристикам процесса. С точки зрения процесса, нить можно определить как независимый поток управления, выполняемый в контексте процесса. При этом каждая нить, в свою очередь, имеет свой собственный контекст.

Принципы организации свопинга.

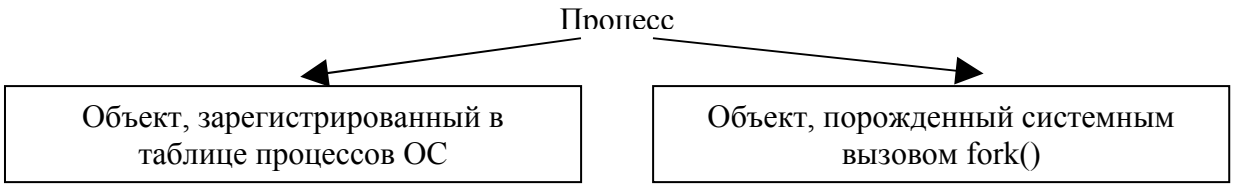
Проблема планирования определяет эксплуатационные качества системы. Это планирование времени, памяти. Это организация стратегий очередей. Планирование дает тип ОС. Рассмотрим ОС разделения времени. ОС разделения времени может быть с разными квантами времени, от чего меняются качества работы ОС. Большой квант времени – выполняются много мелких программ; маленький квант времени – многопользовательский режим работы. В ОС реального времени должна быть гарантия обработки времени за некоторое время.
Проблема организации свопинга. Мультипрограммные ОС используют свопинг. Элементом планирования является планирование свопинга. Пример: простейшая операционная система типа UNIX: полная откачка процесса в свопинговую память. Сейчас все делается по частям. Правила откачки: в контексте процесса была переменная p_time – время непрерывного размещения процесса в область свопинга и обратно. Сначала p_time обнулилась и считала время, сколько процесс находится в этой области. Затем по определению расписания в ОС запускалась функция обработки, которая анализировала область свопинга и выбирала процесс с максимальным p_time, затем система анализировала наличие свободной оперативной памяти. Если ее достаточно, то процесс в нее загружался, иначе операционная система смотрела процесс, который закрыт по обмену и имеет максимальный p_time, и его откачивала в область свопинга. Затем вновь анализ оперативной памяти, если недостаточно, то далее анализировались процессы в оперативной памяти, затем отыскивался процесс с максимальным p_time и освобождал память. Он мешал в последнюю очередь тем процессам, которые нормально читались, однако мешал процессу в обмене. Это лишает ОС возможности сразу закончить обмен. Это простейшая схема.

БИЛЕТ 23 Процесс в Unix

Определение процесса. Контекст

В любой системе, оперирующей понятием процесс, существует системно-ориентированное определение процесса (определение, учитывающее конкретные особенности данной ОС).

С точки зрения Unix системно-ориентированное определение процесса:



Рассмотрим данные определения процесса Unix.

Процесс в ОС Unix – объект (не надо путать с объектом ООП!!!), зарегистрированный в таблице процессов Unix.

Таблица процессов

Каждый процесс характеризуется уникальным именем – ***идентификатором процесса*** (PID). PID – целое число от 0 до некоторого предельного значения, определяющего максимальное число процессов, существующих в системе одновременно.

Будем использовать термины 0й процесс, 1й процесс, 125й процесс, это означает, что речь идет о процессах с PID = 0, 1, 125. 0й процесс в системе ассоциируется с работой ядра Unix. С точки зрения организации данных PID – номер строки в таблице, в которой размещена запись о процессе.

Контекст процесса

Содержимое записи таблицы процессов позволяет получить ***контекст процесса*** (часть данных контекста размещается непосредственно в записи таблицы процессов, на оставшуюся часть контекста имеются прямые или косвенные ссылки, также размещенные в записи таблицы процессов).

С точки зрения логической структуры контекст процесса Unix состоит из:

- ***пользовательской составляющей*** или ***тела процесса*** (иногда используется пользовательский контекст)
- ***аппаратной составляющей*** (иногда используется аппаратный контекст)
- ***системной составляющей*** ОС Unix (иногда – системный контекст)

Иногда два последних компонента объединяют, в этом случае используется термин ***общесистемная составляющая контекста***.

Тело процесса состоит из ***сегмента кода*** и ***сегмента данных***.

Сегмент кода содержит машинные команды и неизменяемые константы соответствующей процессу программы.

Сегмент данных – содержит данные, динамически изменяемые в ходе выполнения кода процесса. Сегмент данных содержит область статических переменных, область разделяемой с другими процессами памяти, а также область стека (обычно эта область служит основой для организации автоматических переменных, передачи параметров в функции, организацию динамической памяти).

Некоторые современные ОС имеют возможность разделения единого сегмента кода между разными процессами. Тем самым достигается экономия памяти в случаях одновременного выполнения идентичных процессов.

Например, при функционировании терминального класса одновременно могут быть сформированы несколько копий текстового редактора. В этом случае сегмент кода у всех процессов, соответствующих редакторам, будет единый, а сегменты данных будут у каждого процесса свои.
Следует отметить, что при использовании динамически загружаемых библиотек возможно разделение сегмента кода на неизменную часть, которая может разделяться между процессами и часть, соответствующую изменяемому в динамике коду подгружаемых программ.
Аппаратная составляющая содержит все регистры и аппаратные таблицы ЦП, используемые активным или исполняемым процессом (счетчик команд, регистр состояния процессора, аппарат виртуальной памяти, регистры общего назначения и т. д.).

Обращаем внимание, что аппаратная составляющая имеет смысл только для процессов, находящихся в состоянии ***выполнения***. Для процессов, находящихся в других состояниях содержимое составляющей не определено.

Системная составляющая.

В системной составляющей контекста процесса содержатся различные атрибуты процесса, такие как:

- идентификатор родительского процесса;
- текущее состояние процесса;
- приоритет процесса;
- реальный идентификатор пользователя-владельца (идентификатор пользователя, сформировавшего процесс);
- эффективный идентификатор пользователя-владельца (идентификатор пользователя, по которому определяются права доступа процесса к файловой системе);
- реальный идентификатор группы, к которой принадлежит владелец (идентификатор группы к которой принадлежит пользователь, сформировавший процесс);
- эффективный идентификатор группы, к которой принадлежит владелец (идентификатор группы «эффективного» пользователя, по которому определяются права доступа процесса к файловой системе);
- список областей памяти;
- таблица открытых файлов процесса;
- информация о том, какая реакция установлена на тот или иной сигнал (аппарат сигналов позволяет передавать воздействия от ядра системы процессу и от процесса к процессу);
- информация о сигналах, ожидающих доставки в данный процесс;
- сохраненные значения аппаратной составляющей (когда выполнение процесса приостановлено).

Рассмотрим второе определение процесса Unix.

Процесс в ОС Unix – это объект, порожденный системным вызовом ***fork()***. Данный системный вызов является единственным стандартным средством порождения процессов в системе Unix. Ниже рассмотрим возможности данного системного вызова подробнее.

Аппарат системных вызов в ОС UNIX.

Привилегированный и обычный режим(есть набор инструкций, доступный только из привил.)Чтобы работать в с ресурсами ВС – переход в привел. Системные вызовы, предоставляемые ОС UNIX. К интересующим нас вызовам относятся вызовы

- для создания процесса;
- для организации ввода вывода;
- для решения задач управления;
- для операции координации процессов;
- для установки параметров системы.

Отметим некоторые общие моменты, связанные с работой системных вызовов.

Большая часть системных вызовов определены как функции, возвращающие целое значение, при этом при нормальном завершении системный вызов возвращает 0, а при неудачном завершении -1[4]. При этом код ошибки можно выяснить, анализируя значение внешней переменной **errno**, определенной в заголовочном файле **<errno.h>**.

В случае, если выполнение системного вызова прервано сигналом, поведение ОС зависит от конкретной реализации. Например, в BSD UNIX ядро автоматически перезапускает системный вызов после его прерывания сигналом, и таким образом, внешне никакого различия с нормальным выполнением системного вызова нет. Стандарт POSIX допускает и вариант, когда системный вызов не перезапускается, при этом системный вызов вернет –1, а в переменной **errno** устанавливается значение **EINTR**, сигнализирующее о данной ситуации.

4

БИЛЕТ 24

Базовые средства организации и управления процессами

Для порождения новых процессов в UNIX существует единая схема, с помощью которой создаются все процессы, существующие в работающем экземпляре ОС UNIX, за исключением первых двух процессов (0-го и 1-го).

Для создания нового процесса в операционной системе UNIX используется системный вызов **fork(),** в результате в таблицу процессов заносится новая запись, и порожденный процесс получает свой уникальный идентификатор. Для нового процесса создается контекст, большая часть содержимого которого идентична контексту родительского процесса, в частности, тело порожденного процесса содержит копии сегментов кода и данных его родителя. Сыновний процесс наследует от родительского процесса:

- **окружение** - при формировании процесса ему передается некоторый набор параметров-переменных, используя которые, процесс может взаимодействовать с операционным окружением (интерпретатором команд и т.д.);
- **файлы, открытые в процессе-отце**, за исключением тех, которым было запрещено передаваться процессам-потомкам с помощью задания специального параметра при открытии. (Речь идет о том, что в системе при открытии файла с файлом ассоциируется некоторый атрибут, который определяет правила передачи этого открытого файла сыновним процессам. По умолчанию открытые в «отце» файлы можно передавать «потомкам», но можно изменить значение этого параметра и блокировать передачу открытых в процессе-отце файлов.);
- **способы обработки сигналов**;
- **разрешение переустановки эффективного идентификатора пользователя**;
- **разделяемые ресурсы** процесса-отца;
- **текущий рабочий каталог и домашний каталоги**
- и т.д.

По завершении системного вызова **fork()** каждый из процессов – родительский и порожденный – получив управление, продолжат выполнение с одной и той же инструкции одной и той же программы, а именно с той точки, где происходит возврат из системного вызова **fork().** Вызов **fork()** в случае удачного завершения возвращает сыновнему процессу значение **0**, а родительскому **PID** порожденного процесса. Это принципиально важно для различения сыновнего и родительского процессов, так как сегменты кода у них идентичны. Таким образом, у программиста имеется возможность разделить путь выполнения инструкций в этих процессах.

В случае неудачного завершения, т.е. если сыновний процесс не был порожден, системный вызов **fork()** возвращает –1, код ошибки устанавливается в переменной **errno**.

Пример.

Программа создает два процесса – процесс-предок распечатывает заглавные буквы, а процесс-потомок строчные.

```
int main(int argc, char **argv)
{
     char ch, first, last;
     int pid;
     if((pid=fork())>0)
     {
          /*процесс-предок*/
          first =’A’;
          last =’Z’;
     }
     else
         {
          /*процесс-потомок*/
          first =’a’;
          last =’z’;
         }

     for (ch = first; ch <= last; ch++)
     {
               write(1,&ch,1);
     }
          _exit(0);
}
```

Механизм замены тела процесса.

Семейство системных вызовов **exec()** производит замену тела вызывающего процесса, после чего данный процесс начинает выполнять другую программу, передавая управление на точку ее входа. Возврат к первоначальной программе происходит только в случае ошибки при обращении к **exec()** , т.е. если фактической замены тела процесса не произошло.

Заметим, что выполнение “нового” тела происходит в рамках уже существующего процесса, т.е. после вызова **exec()** сохраняется идентификатор процесса, и идентификатор родительского процесса, таблица дескрипторов файлов, приоритет, и большая часть других атрибутов процесса. Фактически происходит замена сегмента кода и сегмента данных. Изменяются следующие атрибуты процесса:

- режимы обработки сигналов: для сигналов, которые перехватывались, после замены тела процесса будет установлена обработка по умолчанию, т.к. в новой программе могут отсутствовать указанные функции-обработчики сигналов;
- эффективные идентификаторы владельца и группы могут измениться, если для новой выполняемой программы установлен s-бит
- перед началом выполнения новой программы могут быть закрыты некоторые файлы, ранее открытые в процессе. Это касается тех файлов, для которых при помощи системного вызова **fcntl()** был установлен флаг

close-on-exec. Соответствующие файловые дескрипторы будут помечены как свободные.

Ниже представлены прототипы функций семейства **exec()**:

```
#include <unistd.h>
int execl(const char *path, char *arg0,…);
int execlp(const char *file, char *arg0,…);
int  execle(const  char  *path,  char  *arg0,…,  const  char
**env);
int execv(const char *path, const char **arg);
int execvp(const char *file, const char **arg);
int  execve(const  char  *path,  const  char  **arg,  const  char
**env);
```

Первый параметр во всех вызовах задает имя файла программы, подлежащей исполнению. Этот файл должен быть исполняемым файлом и пользователь-владелец процесса должен иметь право на исполнение данного файла. Для функций с суффиксом «p» в названии имя файла может быть кратким, при этом при поиске нужного файла будет использоваться переменная окружения PATH. Далее передаются аргументы командной строки для вновь запускаемой программы, которые отобразятся в ее массив **argv** – в виде списка аргументов переменной длины для функций с суффиксом «l» либо в виде вектора строк для функций с суффиксом «v». В любом случае, в списке аргументов должно присутствовать как минимум 2 аргумента: имя программы, которое отобразится в элемент **argv[0]**, и значение NULL, завершающее список.

В функциях с суффиксом «e» имеется также дополнительный аргумент, описывающий переменные окружения для вновь запускаемой программы – это массив строк вида `name=value`, завершенный значением NULL.

Пример.

```
#include <unistd.h>
int main(int argc, char **argv)
{
     …
     /*тело программы*/
     …
     execl(“/bin/ls”,”ls”,”-l”,(char*)0);
     /*  или execlp(“ls”,”ls”, ”-l”,(char*)0);*/
     printf(“это напечатается в случае неудачного обращения
     к предыдущей функции, к примеру, если не был найден
     файл ls \n”);
     …
}
```

В данном случае второй параметр – вектор из указателей на параметры строки, которые будут переданы в вызываемую программу. Как и ранее первый указатель – имя программы, последний – нулевой указатель. Эти вызовы удобны, когда заранее неизвестно число аргументов вызываемой программы.

Чрезвычайно полезным является использование **fork()** совместно с системным вызовом **exec().** Как отмечалось выше системный вызов **exec()** используется для запуска исполняемого файла в рамках существующего процесса. Ниже приведена общая схема использования связки **fork() - exec().**

Завершение процесса.

Для завершения выполнения процесса предназначен системный вызов **_exit()**

```
void _exit(int exitcode);
```

Кроме обращения к вызову **_exit()**, другими причинами завершения процесса могут быть:

- оператора **return**, входящего в состав функции **main()**
- получение некоторых сигналов (об этом речь пойдет чуть ниже)

В любом из этих случаев происходит следующее:

- освобождаются сегмент кода и сегмент данных процесса
- закрываются все открытые дескрипторы файлов
- если у процесса имеются потомки, их предком назначается процесс с идентификатором 1
- освобождается большая часть контекста процесса, однако сохраняется запись в таблице процессов и та часть контекста, в которой хранится статус завершения процесса и статистика его выполнения
- процессу-предку завершаемого процесса посылается сигнал **SIGCHLD**

Состояние, в которое при этом переходит завершаемый процесс, в литературе часто называют состоянием “зомби”.

Процесс-предок имеет возможность получить информацию о завершении своего потомка. Для этого служит системный вызов **wait()**:

```
pid_t wait(int *status);
```

При обращении к этому вызову выполнение родительского процесса приостанавливается до тех пор, пока один из его потомков не завершится либо не будет остановлен. Если у процесса имеется несколько потомков, процесс будет ожидать завершения любого из них (т.е., если процесс хочет получить информацию о завершении каждого из своих потомков, он должен несколько раз обратиться к вызову **wait()**).

Возвращаемым значением **wait()** будет идентификатор завершенного процесса, а через параметр **status** будет возвращена информация о причине завершения процесса (путем вызова **_exit()** либо прерван сигналом) и коде возврата. Если процесс не интересуется это информацией, он может передать в качестве аргумента вызову **wait()** NULL-указатель.

Если к моменту вызова **wait()** один из потомков данного процесса уже завершился, перейдя в состояние зомби, то выполнение родительского процесса не блокируется, и **wait()** сразу же возвращает информацию об этом завершенном процессе. Если же к моменту вызова **wait()** у процесса нет потомков, системный вызов сразу же вернет –1. Также возможен аналогичный возврат из этого вызова, если его выполнение будет прервано поступившим сигналом.

Жизненный цикл процессов

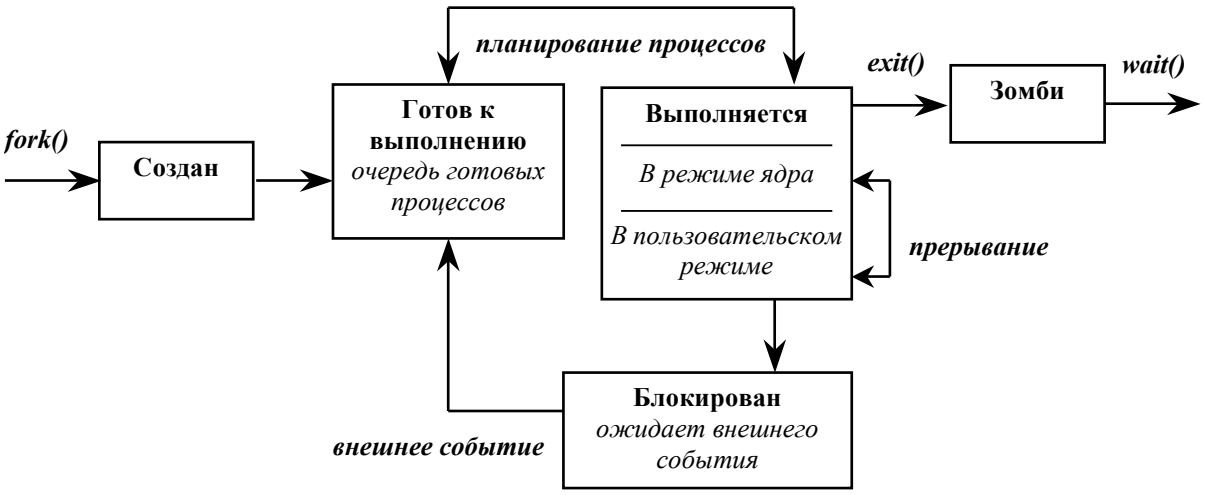


Рис. 3 Жизненный цикл процесса.

Формирование процессов 0 и 1

Рассмотрим подробнее, что происходит в момент начальной загрузки ОС UNIX. Начальная загрузка – это загрузка ядра системы в основную память и ее запуск. Нулевой блок каждой файловой системы предназначен для записи короткой программы, выполняющей начальную загрузку. Начальная загрузка выполняется в несколько этапов.

1.Аппаратный загрузчик читает нулевой блок системного устройства.

2.После чтения этой программы она выполняется, т.е. ищется и считывается в память файл **/unix**, расположенный в корневом каталоге и который содержит код ядра системы.

3.Запускается на исполнение этот файл.

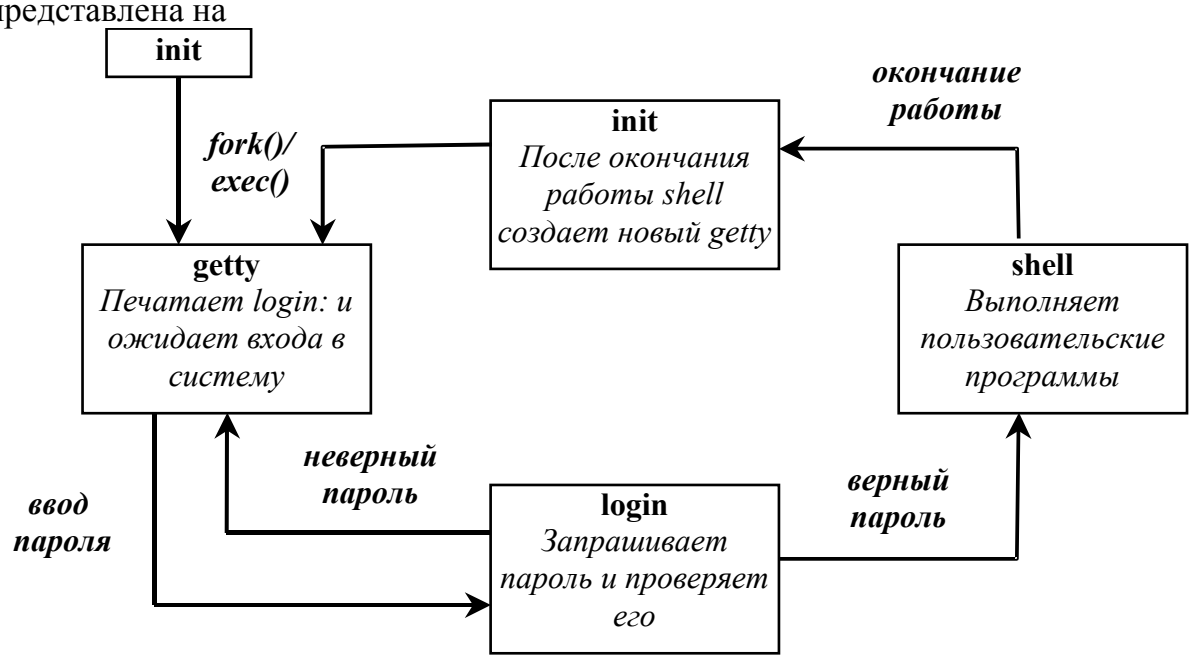
В самом начале ядром выполняются определенные действия по инициализации системы, а именно, устанавливаются системные часы (для генерации прерываний), формируется диспетчер памяти, формируются значения некоторых структур данных (наборы буферов блоков, буфера индексных дескрипторов) и ряд других. По окончании этих действий происходит инициализация процесса с номером **"0".** По понятным причинам для этого невозможно использовать методы порождения процессов, изложенные выше, т.е. с использованием функций **fork()** и **exec().** При инициализации этого процесса резервируется память под его контекст и формируется нулевая запись в таблице процессов. Основными отличиями нулевого процесса являются следующие моменты

1.Данный процесс не имеет кодового сегмента, это просто структура данных, используемая ядром, и процессом его называют потому, что он каталогизирован в таблице процессов.

2. Он существует в течении всего времени работы системы (чисто системный процесс) и считается, что он активен, когда работает ядро ОС.

Далее ядро копирует **"0"** процесс и создает **"1"** процесс. Алгоритм создания этого процесса напоминает стандартную процедуру, хотя и носит упрощенный характер. Сначала процесс **"1"** представляет собой полную копию процесса **"0"** , т.е. у него нет области кода. Далее происходит увеличение его размера и во вновь

созданную кодовую область копируется программа, реализующая системный вызов **exec()** , необходимый для выполнения программы **/etc/init.** На этом завершается подготовка первых двух процессов. Первый из них представляет собой структуру данных, при помощи которой ядро организует мультипрограммный режим и управление процессами. Второй – это уже подобие реального процесса. Далее ОС переходит к выполнению программ диспетчера. Диспетчер наделен обычными функциями и на первом этапе у него нет выбора – он запускает **exec() ,** который заменит команды процесса **"1"** кодом, содержащимся в файле **/etc/init.** Получившийся процесс, называемый **init**, призван настраивать структуры процессов системы. Далее он подключает интерпретатор команд к системной консоли. Так возникает однопользовательский режим, так как консоль регистрируется с корневыми привилегиями и доступ по каким-либо другим линиям связи невозможен. При выходе из однопользовательского режима **init** создает многопользовательскую среду. С этой целью **init** организует процесс **getty** для каждого активного канала связи, т.е. каждого терминала Это программа ожидает входа кого-либо по каналу связи. Далее, используя системный вызов **exec(), getty** передает управление программе **login,** проверяющей пароль. Во время работы ОС процесс **init** ожидает завершения одного из порожденных им процессов, после чего он активизируется и создает новую программу **getty** для соответствующего терминала. Таким образом процесс **init** поддерживает многопользовательскую структуру во время функционирования системы. Схема описанного “круговорота” представлена на



БИЛЕТ 25

Основные задачи планирования

- Планирование очереди процессов на начало обработки
- Планирование распределения времени ЦП между процессами
- Планирование свопинга
- Планирование обработки прерываний
- Планирование очереди запросов на обмен

Планирование распределения времени ЦП между процессами

Квант времени – непрерывный период процессорного времени.
Приоритет процесса – числовое значение, показывающее степень привилегированности процесса при использовании ресурсов ВС (в частности, времени ЦП).

Для грамотного планирования надо решить две задачи:
– определить величину кванта
– определить стратегию обслуживания очереди готовых к выполнению процессов

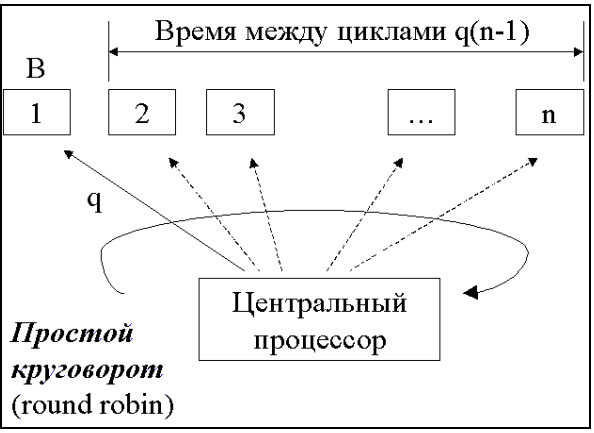
Если величина кванта не ограничена – ***невытесняющая стратегия*** планирования времени ЦП (применяется в пакетных системах). Никто принудительно не скидывает процесс с ЦП. Разработчики берут на себя функции диспетчера. Например, программа что-то долго считает => сама периодически снимает себя с ЦП, что могли выполнится задачи требующие меньшее количество времени для выполнения и не могущие долго ждать.

Вытесняющая стратегия **-** величина кванта ограничена.

Может существовать несколько очередей на обработку на ЦП. Первыми берутся процессы из первой очереди. Вторая очередь подпитывает первую, третья – вторую, и т.д.. Если первая очередь пуста берется из второй, вторая пуста – из третей и т. д. Планировщик определяет в какую очередь скинуть процесс.

Рассмотрим, как решается проблема с определения кванта времени.

8.3.1 Кванты постоянной длины.



•Время ожидания кванта процессом ~ q(n-1)
•Параметры: длина очереди и величина кванта.
•Дисциплина обслуживания очереди, например, FIFO.
•Переключение процессов – операция, требующая времени.
Проблема: как определить длину кванта. Слишком маленький – не хватит времени на переключение, большой - некоторые успеют выполниться полностью.

8.3.2 Кванты переменной длины

Величина кванта может меняться со временем

- Вначале «большой» квант q=A,на следующем шаге q=A-t, q=A-2t,…, до q=B (B<A). Преимущество для коротких задач.
- Вначале q=B, далее q=B+t,…, до q=A. Уменьшение накладных расходов на переключение задач, когда несколько задач выполняют длительные вычисления.

Если процесс интенсивно пользуется операциями ввода/вывода, то он может использовать выделенный квант не до конца. В качестве компенсации ему могут предоставляться привилегии при дальнейшем обслуживании.

БИЛЕТ 26 Стратегии планирования времени ЦП. Алгоритмы, основанные на приоритетах.

Алгоритмы, основанные на приоритетах

Вычисление приоритета основывается на статических и динамических характеристиках. Изменение приоритета может происходить по инициативе процесса, пользователя, ОС. Правила назначения приоритета процессов определяют эффективность работы системы.

8.4.1 Планирование по наивысшему приоритету (highest priority first - HPF).

При появлении в очереди готовых процессов процесса с более высоким приоритетом, чем у текущего наступает момент смены процесса.
Возможно два варианта:

- относительный приоритет (ожидание исчерпания кванта у текущего процесса)
- абсолютный приоритет (немедленная смена текущего процесса)

Задача выбора/постановки процесса с наивысшим приоритетом зависит от организации очереди (упорядочена/неупорядочена).
Возможно наличие очередей с одинаковым приоритетом.
Пример использования стратегии HPF.
Выбор самого короткого задания (shortest job first - SJF).

Время выполнения – характеристика, на которой основан приоритет. Приоритет обратно пропорционален ожидаемому времени обработки.
Этот вариант
удобен для “коротких” процессов.

8.4.2 Класс подходов, использующих линейно возрастающий приоритет.

Процесс при входе в систему получает некий приоритет, который возрастает с коэффициентом A во время ожидания в очереди готовых процессов, и с коэффициентом B во время выполнения.
Из выбора A и B - разные правила планирования:

- Если 0<A<=B обслуживание очереди по дисциплине FIFO
- Если 0>B>=A обслуживание очереди по дисциплине LIFO

8.4.3 Нелинейные функции изменения приоритета

Например, приоритет убывает по линейному закону с течением времени. Когда достигается некое максимальное время, приоритет скачком возрастает до некоторой большой величины. Это благоприятствует коротким процессам, и при этом соблюдается условие, что ни одному процессу не придется ждать обслуживания слишком долго.
В частности, метод SJF можно модифицировать, добавляя приоритет длинным процессам после некоторого времени ожидания.

8.5 Разновидности круговорота.

Простой круговорот (RR – round robin) не использует никакой статистической или динамической информации о приоритетах. (см. рисунок выше)

При ***круговороте со смещением*** каждому процессу соответствует своя длина кванта, пропорциональная его приоритету.

«Эгоистический» круговорот. Если параметры A и B : 0<=B<A.
Процесс, войдя в систему ждет пока его приоритет не достигнет приоритета работающих процессов, а далее выполняется в круговороте.
Приоритет выполняемых процессов увеличивается с коэффициентом B<A, следовательно, ожидающие процессы их догонят.
При B=0 «эгоистический» круговорот практически сводится к простому.

8.6 Очереди с обратной связью (feedback – FB).

Используется N очередей. Новый процесс ставится в первую очередь, после получения кванта он переносится во вторую и так далее. Процессор обслуживает непустую очередь с наименьшим номером.
В FB поступивший процесс **неявно** получает наивысший приоритет и выполняется подряд в течение нескольких квантов до прихода следующего, но не более чем успел проработать предыдущий.
«-» Работа с несколькими очередями – издержки.
«+» Удобны для коротких заданий: не требуется предварительная информация о времени выполнения процессов.

БИЛЕТ 27

Смешанные алгоритмы планирования

На практике концепции квантования и приоритетов часто используются совместно.
К примеру, в основе – концепция квантования, а определение кванта и/или дисциплина обслуживания очередей базируется на приоритетах.

БИЛЕТ 28 . Планирование свопинга в ОС UNIX.

Планирование процесса, которому предстоит занять время центрального процессора, основывается на понятии приоритета. Каждому процессу сопоставляется некоторое целое числовое значение его приоритета (в т.ч., возможно, и отрицательное). Общее правило таково: чем больше числовое значение приоритета процесса, тем меньше его приоритет, т.е. наибольшие шансы занять время ЦП будут у того процесса, у которого числовое значение приоритета минимально.

Итак, числовое значение приоритета, или просто приоритет процесса - это параметр, который размещен в таблице процессов, и по значению этого параметра осуществляется выбор очередного процесса для продолжения работы и принимается решение о приостановке работающего процесса. Приоритеты системных и пользовательских процессов вычисляются по-разному. Рассмотрим, как это происходит для пользовательского процесса.

В вычислении приоритета **P_PRI** используются две изменяемые составляющие - **P_NICE** и **P_CPU. P_NICE** - это пользовательская составляющая приоритета. Его начальное значение полагается равным системной константе **NZERO**, в процессе выполнения процесса **P_NICE** может модифицироваться системным вызовом **nice().** Аргументом этого системного вызова является добавка к текущему значению (для обычного – непривилегированного - процесса эти добавки представляют собой неотрицательные числа). Значение **P_NICE** наследуется при порождении процессов, и таким образом, значение приоритета не может быть понижено при наследовании. Заметим, что изменяться **P_NICE** может только в сторону увеличения значения (до некоторого предельного значения), таким образом пользователь может снижать приоритет своих процессов.

P_CPU - это системная составляющая. Она формируется системой следующим образом: при прерывании по таймеру через предопределенные периоды времени для процесса, занимающего процессор в текущий момент, **P_CPU** увеличивается на единицу. Также, как и **P_NICE**, **P_CPU** имеет некоторое предельное значение. Если процесс будет находиться в состоянии выполнения так долго, что составляющая **P_CPU** достигнет своего верхнего предела, то значение **P_CPU** будет сброшено в нуль, а затем снова начнет расти. Отметим, однако, что такая ситуация весьма маловероятна, т.к. скорее всего, этот процесс будет выгружен и заменен другим еще до того момента, как **P_CPU** достигнет максимума.

В общем случае, приоритет процесса есть функция:

P_PRI = f(P_NICE, P_CPU)

Константа **P_USER** представляет собой нижний порог приоритета для пользовательских процессов. Пользовательская составляющая, как правило, учитывается в виде разности **_NICE – NZERO**, что позволяет принимать в расчет только добавку, введенную посредством системного вызова **nice().** Системная составляющая учитывается с некоторым коэффициентом. Поскольку неизвестно, проработал ли до момента прерывания процесс на процессоре полный интервал между прерываниями, то берется некоторое усреднение. Суммарно получается следующая формула для вычисления приоритета

P_PRI = P_USER + P_NICE – NZERO + P_CPU/a.

Заметим, что, если приоритет процесса не изменялся при помощи **nice()**, то единственной изменяемой составляющей приоритета будет **P_CPU**, причем эта составляющая растет только для того процесса, который находится в состоянии выполнения. В тот момент, когда значение ее станет таково, что в очереди готовых к выполнению процессов найдется процесс с меньшим значением приоритета, выполняемый процесс будет приостановлен и заменен процессом с меньшим значением приоритета. При этом значение составляющей **P_CPU** для выгруженного процесса сбрасывается в нуль.

Пример. Рассмотрим два активных процесса, разделяющих процессор, причем таких, что ни их процессы-предки, ни они сами не меняли составляющую **P_NICE** системным вызовом **nice() .** Тогда **P_NICE =NZERO и** оба процесса имеют начальное значение приоритета **P_PRI = P_USER**, так как **P_CPU=0.** Пусть значение **P_CPU** увеличивается на единицу через **N** единиц времени (частота прерывания по таймеру), а в вычисление приоритета она входит с коэффициентом **1/A.** Таким образом, дополнительная единица в приоритете процесса, занимающего процессор, «набежит» через **A** таймерных интервалов. Значение **P_CPU** второго процесса остается неизменным, и его приоритет остается постоянным. Через **NA** единиц времени разница приоритетов составит единицу в пользу второго процесса и произойдет смена процессов на процессоре.
Принципы организация своппинга.

В системе определенным образом выделяется пространство для области свопинга. Есть пространство оперативной памяти, в котором находятся процессы, обрабатываемые системой в режиме мультипрограммирования. Есть область на ВЗУ, предназначенная для откачки этих процессов по мере необходимости. Упрощенная схема планирования подкачки основывается на использовании некоторого приоритета, который называется **P_TIME** и также находится в контексте процесса. В этом параметре аккумулируется время пребывания процесса в состоянии мультипрограммной обработки, или в области свопинга. В поле **P_TIME** существует счётчик выгрузки (**outage**) и счётчик загрузки (**inage**).
При перемещении процесса из оперативной памяти в область свопинга или обратно система обнуляет значение параметра **P_TIME**. Для загрузки процесса в память из области свопинга выбирается процесс с максимальным значением **P_TIME**. Если для загрузки этого процесса нет свободного пространства оперативной памяти, то система ищет среди процессов в оперативной памяти процесс, ожидающий ввода/вывода (сравнительно медленных операций, процессы у которых приоритет выше значения **P_ZERO**) и имеющий максимальное значение **P_TIME** (т.е. тот, который находился в оперативной памяти дольше всех). Если такого процесса нет, то выбирается просто процесс с максимальным значением **P_TIME**.
Windows NT
Определено 32 уровня приоритета.
2 класса нитей:

- нити с переменными приоритетами (1-15);
- нити “реального времени” (16-32);

Изначально процессу присваивается базовый приоритет. Нить получает значение. Из базовых приоритетов операционная система может менять базовый приоритет.

Планирование проводится по высшему приоритету. Поддерживается группа очередей (по одной для каждого приоритета). Система просматривает очереди, начиная с высшего приоритета. Если квант закончился, то процесс получает приоритет 1 и отправляется в конец. Повышается значение приоритета при выходе из состояния ввода-вывода.

БИЛЕТ 29 Планирование в системах реального времени

Системы реального времени являются специализированными системами в которых все функции планирования ориентированы на обработку некоторых событий за время, не превосходящее некоторого предельного значение.

Системы реального времени бывают “Жесткие” и ”мягкие ”.

В первом случае время завершения выполнения каждого из процессов должно быть ***гарантировано*** для всех сценариев функционирования системы.
Это может быть обеспечено за счет :
- полного тестирования всевозможных сценариев

- построения статического расписания

- выбора математически просчитанного алгоритма динамического планирования

Периодические запросы – все моменты запроса периодического процесса можно определить заранее.
Пусть $\{T_i\}$ набор периодических процессов с периодами – p_i , предельными сроками выполнения d_i и требованиями ко времени выполнения c_i.
Для проверки возможного составления расписания анализируется расписание на отрезке времени равному наименьшему общему множителю периодов этих процессов.
Необходимое условие наличия расписания:
Сумма коэффициентов использования $\mu=\Sigma\ c_i / p_i <= k$, где k - количество доступных процессоров.

Классический алгоритм для жестких систем реального времени с одним процессором
Используются периодические запросы на выполнение процессов,
срок выполнения каждого процесса равен его периоду $\mathbf{p_i}$, все процессы независимы максимальное время выполнения каждого процесса $\mathbf{c_i}$ известно и постоянно, игнорируется время переключения контекста, вводится ограничение на суммарный коэффициент загрузки процессора $\mathbf{S\ c_i / p_i}$, при существовании **n** задач не превосходит $\mathbf{n(2^{1/n}-1)}$. Эта величина при $\mathbf{n\to\infty}$ равна **ln 2,** то есть **0.7**
Используются вытеснения и статические приоритеты.

Суть алгоритма:
Процессы получают статические приоритеты в соответствии с величиной их периодов выполнения, при этом самый высший приоритет получает самая короткая задача.
Соблюдение приведенных ограничений гарантирует выполнение временных ограничений для всех процессов во всевозможных ситуациях.

Алгоритмы с динамическим изменением приоритетов.
Параметр ***deadline*** – конечный срок выполнения.
Выбор процесса на выполнение по правилу:
выбирается процесс, у которого текущее значение разницы между конечным сроком выполнения и временем, необходимым для его непрерывного выполнения, является наименьшим.

БИЛЕТ 30 Организация планирования обработки прерываний в ОС WINDOWS NT.

ОС должна обеспечивать контроль над ходом выполнения системных процедур, вызываемых по прерываниям . Это необходимое условие для правильного планирования пользовательских процессов.

Рассмотрим пример, в котором обработчик прерываний принтера блокирует на длительное время обработку прерываний от таймера, в результате чего системное время на некоторое время «замирает», и один из процессов (2), критически важный для пользователя, не получат управление в запланированное время.

Упорядоченное планирование прерываний

Механизм прерываний поддерживает **приоритезацию** и **маскирование** прерываний.
Источники прерываний делятся на классы – каждому классу свой уровень приоритета запроса на прерывание.
Дисциплина обслуживания приоритетов
- относительная (выбор по наивысшему приоритету, но далее обработка не может быть отложена)
- абсолютная (происходит переход к обработке более приоритетного с откладыванием текущего)

Механизм маскирования запросов.
В схеме с абсолютными приоритетами заложено маскирование, так как запрещаются запросы с равными или более низкими приоритетами.
В общем случае - возможность маскирования прерываний любого класса и любого приоритета на некоторое время.

Упорядочивание работы обработчиков прерываний – механизм приоритетных очередей.
Наличие в ОС программного модуля – диспетчера прерываний.
При возникновении прерывания – вызов диспетчера.
Он блокирует все прерывания на некоторое время, устанавливает причину прерывания, сравнивает назначенный данному источнику прерывания приоритет с текущим приоритетом. В случае если у нового запроса на прерывание приоритет выше чем у текущего, то выполнение текущего приостанавливается и он помещается в соответствующую очередь. Иначе в соответствующую очередь помещается поступивший обработчик.

Планирование обработки прерываний в Windows NT

Все источники прерываний делятся на несколько классов, и каждому уровню присваивается уровень запроса прерывания – Interrupt Request Level (IRQL). Этот уровень и представляет приоритет данного класса.
Поступление запроса на прерывание/исключение – вызов диспетчера прерываний, который
-Запоминает информацию об источнике прерывания
-Анализирует его приоритет
Если приоритет <= IRQL прерванного, **то** отложить в очередь,
иначе текущий обработчик – в очередь, управление - новому

Одна из важных задач планирования – обеспечение занятости внешних устройств
Для этого можно присваивать процессам высокий приоритет в периоды, когда они интенсивно используют ввод/ вывод
Эти периоды легко прослеживаются: - процесс блокируется про обращении к вводу/выводу. - операции ввода/вывода обычно бывают сконцентрированы в отдельных частях программ.
Применяется стратегия HPF.

БИЛЕТ 31 . Разделяемые ресурсы. Критические секции. Взаимное исключение. Тупики.

В однопроцессорных системах имеет место так называемый **псевдопараллелизм** – хотя в каждый конкретный момент времени процессор занят обработкой одной конкретной задачи, благодаря постоянному переключению с исполнения одной задачи на другую, достигается иллюзия параллельного исполнения нескольких задач. Во многопроцессорных системах задача максимально эффективного использования каждого конкретного процессора также решается путем переключения между процессами, однако тут, наряду с псевдопараллелизмом, имеет место и **действительный параллелизм**, когда на разных процессорах в один и тот же момент времени исполняются разные процессы.
Процессы, выполнение которых хотя бы частично перекрывается по времени, называются параллельными. Они могут быть независимыми и взаимодействующими. Независимые процессы – процессы, использующие независимое множество ресурсов; на результат работы такого процесса не должна влиять работа другого независимого процесса. Наоборот – взаимодействующие процессы совместно используют ресурсы и выполнение одного процесса может оказывать влияние на результат другого. Совместное использование ресурса двумя процессами, когда каждый из процессов полностью владеет ресурсом некоторое время, называют **разделением ресурса**. Разделению подлежат как аппаратные, так программные ресурсы. Разделяемые ресурсы, которые должны быть доступны в текущий момент времени только одному процессу – это так называемые **критические ресурсы**. Таковыми ресурсами могут быть как внешнее устройство, так и некая переменная, значение которой может изменяться разными процессами.
Процессы могут быть связаны некоторыми соотношениями (например, когда один процесс является прямым потомком другого), а могут быть не связанными друг с другом. Кроме того, процессы могут выполняться в разных узлах сети. Эти обстоятельства влияют на способ их взаимодействия, а именно – на возможность совместно использовать ресурсы, обмениваться информацией, оповещать друг друга о наступлении некоторых событий, а также определяют возможность одного процесса влиять на выполнение другого.

Таким образом, необходимо уметь решать две важнейшие задачи:

1. Распределение ресурсов между процессами.
2. Организация защиты ресурсов, выделенных определенному процессу, от неконтролируемого доступа со стороны других процессов.

Важнейшим требованием мультипрограммирования с точки зрения распределения ресурсов является следующее: результат выполнения процессов не должен зависеть от порядка переключения выполнения между процессами, т.е. от

соотношения скорости выполнения данного процесса со скоростями выполнения других процессов.

В качестве примера ситуации, когда это правило нарушается, рассмотрим следующую. Пусть имеется некоторая простая функция, которая считывает символ, введенный с клавиатуры, и выводит его на экран:

```
void echo()
     {char in;input(in);output(in);}
```

В данном примере мы используем некоторые условные функции **input()** и **output()**, так как в данный момент для нас неважно, как конкретно реализован ввод/вывод в данной системе. Поскольку такой кусок кода будет использоваться практически в любой программе, его удобно сделать разделяемым, когда ОС загружает в некоторую область памяти, доступную всем процессам, одну-единственную копию данной программы, и все процессы используют эту копию совместно. Заметим, что в этом случае переменная **in** является разделяемой. Представим теперь ситуацию, изображенную на Рис. 4:

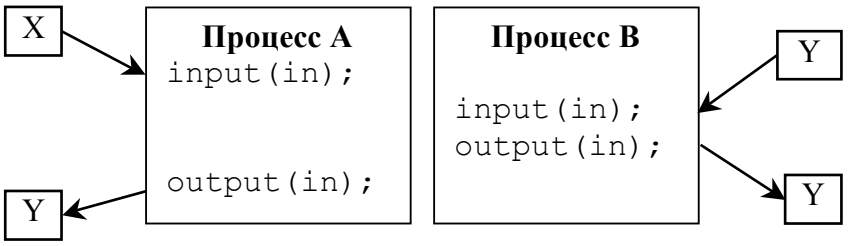


Рис. 4 Конкуренция процессов за ресурс.

1. Процесс **A** вызывает функцию **echo(),** однако в тот момент, когда входной символ был считан в переменную **in**, но до того, как он был выведен на экран, выполнение процесса прерывается и на выполнение загружается процесс **B.**
2. Процесс **B** тоже вызывает функцию **echo().** После выполнения функции **echo()** переменная **in** содержит уж новое значение, считанное с клавиатуры.
3. Процесс **A** возобновляет свою работу в той точке, в которой он был прерван, и выводит на экран символ, находящийся в переменной **in.**

В рассмотренном случае символ, считанный процессом **A**, был потерян, а символ, считанный процессом B, был выведен дважды. Результат выполнения процессов здесь зависит от того, в какой момент осуществляется переключение процессов, и от того, какой конкретно процесс будет выбран для выполнения следующим. Такие ситуации называются **гонками** (в англоязычной литературе - **race conditions**) между процессами, а процессы – конкурирующими. Единственный способ избежать гонок при использовании разделяемых ресурсов – контролировать доступ к любым разделяемым ресурсам в системе. При этом необходимо организовать **взаимное исключение** – т.е. такой способ работы с разделяемым ресурсом, при котором постулируется, что в тот момент, когда один из процессов работает с разделяемым ресурсом, все остальные процессы не могут иметь к нему доступ.

Проблему организации взаимного исключения можно сформулировать в более общем виде. Часть программы (фактически набор операций), в которой осуществляется работа с критическим ресурсом, называется **критической секцией**, или **критическим интервалом**. Задача взаимного исключения в этом случае сводится к тому, чтобы не допускать ситуации, когда два процесса одновременно находятся в критических секциях, связанных с одним и тем же ресурсом.

Заметим, что вопрос организации взаимного исключения актуален не только для взаимосвязанных процессов, совместно использующих определенные ресурсы для обмена информацией. Выше отмечалось, что возможна ситуация, когда процессы, не подозревающие о существовании друг друга, используют глобальные ресурсы системы, такие как, например, устройства ввода/вывода. В с этом случае имеет место конкуренция за ресурсы, доступ к которым также должен быть организован по принципу взаимного исключения.

Важно отметить, что при организации взаимного исключения могут возникнуть две неприятные проблемы:

1. Возникновение так называемых **тупиков (deadlocks**). Рассмотрим следующую ситуацию (см. Рис. 5): имеются процессы **A** и **B**, каждому из которых в некоторый момент требуется иметь доступ к двум ресурсам $\mathbf{R_1}$ и $\mathbf{R_2}$**.** Процесс **A** получил доступ к ресурсу $\mathbf{R_1}$, и следовательно, никакой другой процесс не может иметь к нему доступ, пока процесс **A** не закончит с ним работать. Одновременно процесс **B** завладел ресурсом $\mathbf{R_2}$**.** В этой ситуации каждый из процессов ожидает освобождения недостающего ресурса, но оба ресурса никогда не будут освобождены, и процессы никогда не смогут выполнить необходимые действия.

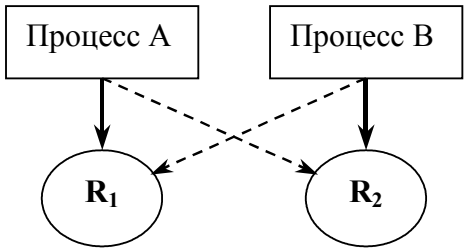


Рис. 5 Возникновение тупиковой ситуации.

2. Ситуация **блокирования** (дискриминации) одного из процессов, когда один из процессов будет бесконечно находиться в ожидании доступа к разделяемому ресурсу, в то время как каждый раз при его освобождении доступ к нему получает какой-то другой процесс.

Таким образом, любые средства организации взаимного исключения должны обеспечивать разрешение этих двух проблем. Помимо этого, к организации взаимного исключения выдвигаются следующие требования:

1. Не должно возникать ситуации, при которой процесс, находящийся *вне* своей критической секции, блокирует исполнение другого процесса.

2. Не должно делаться никаких предположений относительно взаимных скоростей исполнения процессов, а также относительно количества и скоростей работы процессоров в системе.

Далее мы рассмотрим различные механизмы организации взаимного исключения для синхронизации доступа к разделяемым ресурсам и обсудим достоинства, недостатки и области применения этих подходов.

БИЛЕТ 32 Некоторые способы реализации взаимного исключения: семафоры Дейкстры, мониторы, обмен сообщениями.

Семафоры.

Первый из таких подходов был предложен Дейкстрой в 1965 г. Дейкстра предложил новый тип данных, именуемый **семафором**. Семафор представляет собой переменную целого типа, над которой определены две операции: **down(P)** и **up(V).** Операция **down** проверяет значение семафора, и если оно больше нуля, то уменьшает его на **1**. Если же это не так, процесс блокируется, причем операция **down** считается незавершенной. Важно отметить, что вся операция является *неделимой***,** т.е. проверка значения, его уменьшение и, возможно, блокирование процесса производятся как одно *атомарное* действие, которое не может быть прервано. Операция **up** увеличивает значение семафора на **1**. При этом, если в системе присутствуют процессы, блокированные ранее при выполнении **down** на этом семафоре, ОС разблокирует один из них с тем, чтобы он завершил выполнение операции **down**, т.е. вновь уменьшил значение семафора. При этом также постулируется, что увеличение значения семафора и, возможно, разблокирование одного из процессов и уменьшение значения являются атомарной неделимой операцией.

Чтобы прояснить смысл использования семафоров для синхронизации, можно привести простую аналогию из повседневной жизни. Представим себе супермаркет, посетители которого, прежде чем войти в торговый зал, должны обязательно взять себе инвентарную тележку. В момент открытия магазина на входе имеется **N** свободных тележек – это начальное значение семафора. Каждый посетитель забирает одну из тележек (уменьшая тем самым количество оставшихся на **1**) и проходит в торговый зал – это аналог операции **down**. При выходе посетитель возвращает тележку на место, увеличивая количество тележек на **1** – это аналог операции **up**. Теперь представим себе, что очередной посетитель обнаруживает, что свободных тележек нет – он вынужден *блокироваться* на входе в ожидании появления тележки. Когда один из посетителей, находящихся в торговом зале, покидает его, посетитель, ожидающий тележку, *разблокируется***,** забирает тележку и проходит в зал. Таким образом, наш семафор в виде тележек позволяет находиться в торговом зале **(***аналоге критической секции***)** не более чем **N** посетителям одновременно. Положив **N = 1**, получим реализацию взаимного исключения. Семафор, начальное (и максимальное) значение которого равно **1**, называется **двоичным семафором** (так как имеет только **2** состояния: **0** и **1**). Использование двоичного семафора для организации взаимного исключения проиллюстрировано на Рис. 6.

процесс 1

```
int semaphore;
…
down(semaphore);
/*    критическая секция
      процесса 1 */
…
up(semaphore);
…
```

процесс 2

```
int semaphore;
…
down(semaphore);
/*    критическая секция
      процесса 2 */
…
up(semaphore);
…
```

Рис. 6 Взаимное исключение с использованием семафора

Семафоры представляют собой мощное средство синхронизации, однако программирование с использованием семафоров является достаточно тяжелой задачей, причем незаметная на первый взгляд логическая ошибка может привести к образованию тупиковых ситуаций или нарушению условий синхронизации.
С целью облегчить написание корректных программ были предложены более высокоуровневые средства синхронизации, которые мы рассмотрим далее.

Мониторы.

Идея **монитора** была впервые сформулирована в 1974 г. Хоаром. В отличие от других средств, монитор представляет собой *языковую* конструкцию, т.е. некоторое средство, предоставляемое языком программирования и поддерживаемое компилятором. Монитор представляет собой совокупность процедур и структур данных, объединенных в программный модуль специального типа. Постулируются три основных свойства монитора:

1. Структуры данных, входящие в монитор, могут быть доступны только для процедур, входящих в этот монитор (таким образом, монитор представляет собой некоторый аналог объекта в объектно-ориентированных языках и реализует инкапсуляцию данных)
2. Процесс «входит» в монитор путем вызова одной из его процедур
3. В любой момент времени внутри монитора может находиться не более одного процесса. Если процесс пытается попасть в монитор, в котором уже находится другой процесс, он блокируется. Таким образом, чтобы защитить разделяемые структуры данных, их достаточно поместить внутрь монитора вместе с процедурами, представляющими критические секции для их обработки.

Подчеркнем, что монитор представляет собой конструкцию языка программирования, и следовательно, компилятору известно о том, что входящие в него процедуры и данные имеют особую семантику, поэтому первое условие может проверяться еще на этапе компиляции. Кроме того, код для процедур монитора тоже может генерироваться особым образом, чтобы удовлетворялось третье условие. Поскольку организация взаимного исключения в данном случае возлагается на компилятор, количество программных ошибок, связанных с организацией взаимного исключения, сводится к минимуму.

Дополнительная синхронизация: переменные-условия.

Помимо обычных структур данных, мониторы могут включать в себя специальные **переменные-условия**, на которых определены операции **`wait`** и **`signal`.** Они используются для синхронизации. Если процесс, находящийся внутри монитора

(т.е. исполняющий одну из его процедур), обнаруживает, что логически он не может продолжать выполнение, пока не выполнится определенное условие (например, буфер для записи данных переполнился), он вызывает операцию **`wait`** над определенной переменной-условием. При этом его дальнейшее выполнение блокируется, и это позволяет другому процессу, ожидающему входа в монитор, попасть в него. В дальнейшем, если этот другой процесс произведет некоторые действия, которые приведут к изменению обстоятельств (в нашем примере – считает часть данных из буфера), он должен вызвать для соответствующей переменной-условия операцию **`signal`**, что позволит разблокировать ожидающий процесс. Тонкость заключается в том, что разблокированный процесс, как и тот, кто его разблокировал, должен оказаться внутри монитора, но нахождение двух процессов внутри монитора одновременно невозможно по определению. Хоар постулировал, что в этом случае процесс, вызвавший **`signal`**, приостанавливается. Хансен в своей модификации мониторов в 1975 г. предложил более простое дополнительное условие: вызов **`signal`** должен быть самым последним внутри процедуры монитора, чтобы процесс немедленно после его выполнения покинул монитор**.** Заметим, что переменные-условия используются в мониторах не для организации взаимного исключения (оно постулируется самим определением монитора), а для дополнительной синхронизации процессов. В нашем примере разделяемый ресурс – буфер для чтения/записи охраняется от одновременного доступа по чтению и по записи самим монитором, а переменная-условие предохраняет пишущий процесс от затирания ранее записанных данных.

Несомненным достоинством мониторов является то, что взаимное исключение здесь организуется автоматически, что существенно упрощает программирование и снижает вероятность ошибок. Недостатком же является то, что, как уже говорилось, монитор – это языковая конструкция. Следовательно, если язык программирования не содержит таких конструкций (а для большинства распространенных языком это так и есть), программист не может ею воспользоваться. В то же время семафоры, например, являются средством ОС, и если соответствующая ОС поддерживает семафоры, программист может их использовать независимо от того, на каком языке он пишет программы. Мониторы реализованы в некоторых языках программирования, таких как Concurrent Euclid, Concurrent Pascal, Modula-2, Modula-3, однако эти языки не слишком распространены.

Обмен сообщениями.

Общей проблемой и для мониторов, и для семафоров является то, что их реализация существенно опирается на предположение, что мы имеем дело либо с однопроцессорной системой, либо с многопроцессорной системой, где все процессоры имеют доступ к общей памяти. Однако в случае распределенной системы, где каждый процессор имеет прямой доступ только к своей памяти, такие средства не подходят. Более общим средством, решающим проблему синхронизации как для однопроцессорных систем и систем с общей памятью, так и для распределенных, является **обмен сообщениями**.

Обмен сообщениями представляет собой средство, которое может быть использовано как для синхронизации, в частности для организации взаимного исключения, так и для обмена информацией между взаимосвязанными процессами, выполняющими общую работу. Рассмотрим общую концепцию обмена

сообщениями. Основная функциональность реализуется двумя примитивами, реализующими, соответственно, посылку и прием сообщения:

```
send(destination, message)
receive(source, message)
```

Как и семафоры, и в отличие от мониторов, эти примитивы являются системными вызовами, а не конструкциями языка.

Рассмотрим основные особенности, которыми может обладать та или иная система обмена сообщениями.

Синхронизация.

Сам смысл обмена сообщениями предполагает определенную синхронизацию между процессом-отправителем и процессом-получателем, так как сообщение не может быть получено до того, как оно послано. Возникает вопрос, что происходит, если один процесс хочет получить сообщение, а другой его не отослал, и наоборот, если один процесс отсылает сообщение, а другой не собирается его получать. Здесь есть две возможности. Как операция посылки сообщения, так операция приема могут быть **блокирующими** и **неблокирующими.** Для операции **`send`** это означает, что либо процесс-отправитель может блокироваться до тех пор, пока получатель не вызовет **`receive`**, либо выполнение процесса может продолжаться далее независимо от наличия получателя. Для операции **`receive`** подобная ситуация возникает, когда эта операция вызвана раньше, чем сообщение было послано – в этом случае она может либо блокироваться до получения сообщения, либо возвращать управление сразу же.

В зависимости от целей использования механизма сообщений могут быть полезны различные комбинации этих условий:

- Блокирующий **`send`** и блокирующий **`receive`** – эта схема известна под названием «схемы рандеву». Она не требует буферизации сообщений и часто используется для синхронизации процессов
- Неблокирующий **`send`** и блокирующий **`receive`** – такая схема очень распространена в системах клиент/сервер: серверный процесс блокируется в ожидании очередного запроса для обработки, в то время как клиент, пославший запрос серверу, может продолжать выполняться, не ожидая окончания обработки своего запроса
- Также весьма распространена схема, когда обе операции являются неблокирующими – в этом случае оба процесса могут продолжать выполнение, не дожидаясь окончания коммуникации

Важно понимать, что в случае, если **`send`** является неблокирующим, процесс-отправитель не может знать, получено ли его сообщение. В этом случае, если требуется организовать гарантированную доставку сообщений, необходимо, чтобы процессы обменивались сообщениями-подтверждениями. Проблема потери сообщений встает также, если используется блокирующий **`receive`** – в этом случае процесс-получатель может оказаться заблокированным навечно. Поэтому в такую схему часто добавляется дополнительный примитив, позволяющий процессу-получателю проверить, есть ли для него сообщение, но не блокироваться, если его нет.

Адресация.

Другая важная проблема – организовать адресацию сообщений. Одно из решений – так называемая **прямая адресация**, при которой каждому из процессов присваивается некоторый идентификатор, и сообщения адресуются этим идентификаторам. При этом процесс-получатель может указать явно идентификатор отправителя, от которого он желает получить сообщение, либо получать сообщения от любого отправителя.

Иное решение заключается в том, чтобы предоставить специальную структуру данных – **почтовый ящик,** или **очередь сообщений**, которая по сути своей является буфером, рассчитанным на определенное количество сообщений. В этом случае сообщения адресуются не процессам, а почтовым ящикам, при этом один и тот же ящик может использоваться и несколькими отправителями, и несколькими получателями. Такая схема, называемая **косвенной адресацией**, обеспечивает дополнительную гибкость. Заметим, что связь между процессом-получателем или отправителем и почтовым ящиком может быть не только статической (т.е. раз навсегда заданной при создании ящика), но и динамической; в последнем случае для установления и разрыва связи используются дополнительные примитивы (**`connect/disconnect`**). Кроме того, поскольку почтовый ящик является самостоятельным объектом, необходимо наличие примитивов создания и удаления ящика (**`create/destroy`**).

Длина сообщения.

Немаловажным аспектом является формат сообщений. В той или иной системе могут допускаться как сообщения фиксированной длины, так и переменной. В последнем случае в заголовке сообщения, помимо отправителя и получателя, должна указываться длина сообщения. Выбор того или иного варианта зависит от целей, которые преследует система обмена сообщениями, и от предполагаемой архитектуры ВС. Так, если предполагается возможность передачи больших объемов данных, то сообщения с переменной длиной будут более гибким решением и позволят сократить накладные расходы на большое количество коротких сообщений, где значительную часть занимает сам заголовок. С другой стороны, если обмен происходит между процессами на одной машине, немаловажную роль играет эффективность. Здесь, возможно, было бы уместно ограничить длину сообщения, с тем, чтобы использовать для их передачи системные буфера с быстрым доступом.

В заключение отметим еще раз, что механизм обмена сообщениями является мощным и гибким средством синхронизации, пригодным для использования как на однопроцессорных системах и системах с общей памятью, так и в распределенных ВС. Однако, по сравнению с семафорами и мониторами, он, как правило, является менее быстрым.

БИЛЕТ 33

Классические задачи синхронизации процессов.

«Обедающие философы»

Рассмотрим одну из классических задач, демонстрирующих проблему разделения доступа к критическим ресурсам – «задачу об обедающих философах» [2]. Данная задача иллюстрирует ситуацию, когда процессы конкурируют за право исключительного доступа к ограниченному числу ресурсов. Пять философов собираются за круглым столом, перед каждым из них стоит блюдо со спагетти, и между каждыми двумя соседями лежит вилка. Каждый из философов некоторое время размышляет, затем берет две вилки (одну в правую руку, другую в левую) и ест спагетти, затем кладет вилки обратно на стол и опять размышляет и так далее. Каждый из них ведет себя независимо от других, однако вилок запасено ровно столько, сколько философов, хотя для еды каждому из них нужно две. Таким образом, философы должны совместно использовать имеющиеся у них вилки (ресурсы). Задача состоит в том, чтобы найти алгоритм, который позволит философам организовать доступ к вилкам таким образом, чтобы каждый имел возможность насытиться, и никто не умер с голоду.

Рассмотрим простейшее решение, использующее семафоры. Когда один из философов хочет есть, он берет вилку слева от себя, если она в наличии, а затем - вилку справа от себя. Закончив есть, он возвращает обе вилки на свои места.

Данный алгоритм может быть представлен следующим способом:

```
#define N 5                     /* число философов*/
void philosopher (int i)          /* i – номер философа от 0
до 4*/
{
while (TRUE)
     {
     think();                   /*философ думает*/
     take_fork(i);              /*берет левую вилку*/
     take_fork((i+1)%N);             /*берет правую вилку*/
     eat();                     /*ест*/
     put_fork(i);               /*кладет обратно левую вилку*/
     put_fork((i+1)%N);         /* кладет обратно правую вилку
     */
}
}
```

Функция **`take_fork()`** описывает поведение философа по захвату вилки: он ждет, пока указанная вилка не освободится, и забирает ее.

На первый взгляд, все просто, однако, данное решение может привести к тупиковой ситуации. Что произойдет, если все философы захотят есть в одно и то же время? Каждый из них получит доступ к своей левой вилке и будет находиться в состоянии ожидания второй вилки до бесконечности.

Другим решением может быть алгоритм, который обеспечивает доступ к вилкам только четырем из пяти философов. Тогда всегда среди четырех философов по крайней мере один будет иметь доступ к двум вилкам. Данное решение не имеет

тупиковой ситуации. Алгоритм решения может быть представлен следующим образом:

```
# define N 5
# define LEFT (i-1)%N  /* номер легого соседа для i-ого
                        философа */
# define RIGHT (i+1)%N /* номер правого соседа для i-ого
                        философа*/
# define THINKING 0              /* философ думает */
# define HUNGRY 1           /* философ голоден */
# define EATING 2           /* философ ест */

typedef int semaphore;   /* тип данных «семафор» */
int state[N];          /* массив состояний философов */
semaphore mutex=1;/* семафор для критической секции */
semaphore s[N];   /* по одному семафору на философа */

void philosopher (int i)
     /* i : номер философа от 0 до N-1 */
{
while (TRUE)           /* бесконечный цикл */
{
  think();           /* философ думает */
       take_forks(i);       /*философ берет обе вилки  или
                            блокируется */
       eat();               /* философ ест */
       put_forks(i);        /* философ освобожает обе вилки
                            */
}
}

void take_forks(int i)
     /* i : номер философа от 0 до N-1 */
{
down(&mutex);      /* вход в критическую секцию */
     state[i] = HUNGRY; /*записываем, что i-ый  философ
                        голоден */
test(i);      /* попытка взять обе вилки */
up(&mutex);   /* выход из критической секции */
down(&s[i]);  /* блокируемся, если вилок нет */
}

void put_forks(i)
         /* i : номер философа от 0 до N-1 */
{
down(&mutex);      /* вход в критическую секцию */
state[i] = THINKING;   /* философ закончил есть */
     test(LEFT);
/* проверить может ли левый сосед сейчас есть */
```

```
     test(RIGHT);
     /* проверить может ли правый сосед сейчас есть*/
up(&mutex);   /* выход из критической секции */
}

void test(i)
          /* i : номер философа от 0 до N-1 */
{
if (state[i] == HUNGRY && state[LEFT] != EATING &&
state[RIGHT] != EATING)
{
  state[i] = EATING;
       up (&s[i]);
}
}
```

БИЛЕТ 34

Задача «читателей и писателей»

Другой классической задачей синхронизации доступа к ресурсам является задача «читателей и писателей» [2], иллюстрирующая широко распространенную модель совместного доступа к данным. Представьте себе ситуацию, например, в системе резервирования билетов, когда множество конкурирующих процессов хотят читать и обновлять одни и те же данные. Несколько процессов могут читать данные одновременно, но когда один процесс начинает записывать данные (обновлять базу данных проданных билетов), ни один другой процесс не должен иметь доступ к данным, даже для чтения. Возникает вопрос, как спланировать работу такой системы? Одно из решений представлено ниже:

```
typedef int semaphore; /* тип данных «семафор» */
semaphore mutex = 1;   /* контроль за доступом к
                       «rc» (разделямый  ресурс) */
semaphore db = 1;      /* контроль за доступом к базе
                       данных */
int rc = 0;            /* кол-во процессов читающих или
                       пишущих */

void reader (void)
{
     while (TRUE)           /* бесконечный цикл */
     {
          down(&mutex);     /* получить эксклюзивный доступ
                            к «rc»*/
          rc = rc + 1;      /* еще одним читателем больше */
          if (rc == 1) down(&db);     /* если это первый
                                      читатель, нужно
                                      заблокировать эксклюзивный
                                      доступ к базе */
     up(&mutex);       /*освободить ресурс rc */
          read_data_base(); /* доступ к данным */
          down(&mutex);     /*получить эксклюзивный доступ к
                            «rc»*/
          rc = rc - 1:      /* теперь одним читателем меньше
                            */
          if (rc == 0) up(&db);  /*если это был последний
                                 читатель, разблокировать
                                 эксклюзивный доступ к базе
                                 данных */
          up(&mutex);       /*освободить разделяемый ресурс
                            rc */
          use_data_read();  /* некритическая секция */
     }
}

void writer (void)
{
     while(TRUE)            /* бесконечный цикл */
```

```
    {
        think_up_data();   /* некритическая секция */
        down(&db);          /* получить эксклюзивный доступ
                               к данным*/
        write_data_base(); /* записать данные */
        up(&db);            /* отдать эксклюзивный доступ */
    }
}
```

В этом примере, первый процесс, обратившийся к базе данных по чтению, осуществляет операцию **down()** над семафором **db**, тем самым блокируя эксклюзивный доступ к базе, который нужен для записи. Число процессов, осуществляющих чтение в данный момент, определяется переменной **rc** (обратите внимание! Так как переменная **rc** является разделяемым ресурсом – ее изменяют все процессы, обращающиеся к базе данных по чтению – то доступ к ней охраняется семафором **mutex**). Когда читающий процесс заканчивает свою работу, он уменьшает значение **rc** на единицу. Если он является последним читателем, он также совершает операцию **up** над семафором **db**, тем самым разрешая заблокированному писателю, если таковой имелся, получить эксклюзивный доступ к базе для записи.

Надо заметить, что приведенный алгоритм дает преимущество при доступе к базе данных процессам-читателям, так как процесс, ожидающий доступа по записи, будет ждать до тех пор, пока все читающие процессы не окончат работу, и если в это время появляется новый читающий процесс, он тоже беспрепятственно получит доступ. Это может привести к неприятной ситуации в том случае, если в фазе, когда ресурс доступен по чтению, и имеется ожидающий процесс-писатель, будут появляться новые и новые читающие процессы. К примеру, представим себе, что новый читающий процесс появляется каждую секунду и чтение длится в среднем 2 секунды. Количество читающих процессов в этом случае будет приблизительно константным, и процесс-писатель никогда не получит доступ к данным.

Чтобы этого избежать, можно модифицировать алгоритм таким образом, чтобы в случае, если имеется хотя бы один ожидающий процесс-писатель, новые процессы-читатели не получали доступа к ресурсу, а ожидали, когда процесс-писатель обновит данные. В этой ситуации процесс-писатель должен будет ожидать окончания работы с ресурсом только тех читателей, которые получили доступ раньше, чем он его запросил. Однако, обратная сторона данного решения в том, что оно несколько снижает производительность процессов-читателей, так как вынуждает их ждать в тот момент, когда ресурс не занят в эксклюзивном режиме.

БИЛЕТ 35

Задача о «спящем парикмахере»

Еще одна классическая задача на синхронизацию процессов – это так называемая «задача о спящем парикмахере» [2]. Рассмотрим парикмахерскую, в которой работает один парикмахер, имеется одно кресло для стрижки и несколько кресел в приемной для посетителей, ожидающих своей очереди. Если в парикмахерской нет посетителей, парикмахер засыпает прямо на своем рабочем месте. Появившийся посетитель должен его разбудить, в результате чего парикмахер приступает к работе. Если в процессе стрижки появляются новые посетители, они должны либо подождать своей очереди, либо покинуть парикмахерскую, если в приемной нет свободного кресла для ожидания. Задача состоит в том, чтобы корректно запрограммировать поведение парикмахера и посетителей.

Одно из возможных решений этой задачи представлено ниже. Процедура **`barber()`** описывает поведение парикмахера (она включает в себя бесконечный цикл – ожидание клиентов и стрижку). Процедура **`customer()`** описывает поведение посетителя. Несмотря на кажущуюся простоту задачи, понадобится целых 3 семафора: **`customers`** – подсчитывает количество посетителей, ожидающих в очереди, **`barbers`** – обозначает количество свободных парикмахеров (в случае одного парикмахера его значение либо 0, либо 1) и **`mutex`** – используется для синхронизации доступа к разделяемой переменной **`waiting`**. Переменная **`waiting`**, как и семафор **`customers`**, содержит количество посетителей, ожидающих в очереди, она используется в программе для того, чтобы иметь возможность проверить, имеется ли свободное кресло для ожидания, и при этом не заблокировать процесс, если кресла не окажется. Заметим, что как и в предыдущем примере, эта переменная является разделяемым ресурсом, и доступ к ней охраняется семафором **`mutex`**. Это необходимо, так как для обычной переменной, в отличие от семафора, чтение и последующее изменение не являются неделимой операцией.

```
#define CHAIRS 5
typedef int semaphore; /* тип данных «семафор» */
     semaphore customers = 0;    /* посетители, ожидающие в
                                 очереди */
     semaphore barbers = 0;      /* парикмахеры, ожидающие
                              посетителей */
     semaphore mutex = 1;    /* контроль за доступом к
                              переменной waiting */
int waiting = 0;
void barber()
{
while (true) {
          down(customers);   /* если customers == 0, т.е.
                    посетителей нет, то заблокируемся до
                    появления посетителя */
          down(mutex);       /* получаем доступ к waiting */
          waiting = wating - 1;  /* уменьшаем кол-во
                    ожидающих клиентов */
          up(barbers);       /* парикмахер готов к работе */
          up(mutex);         /* освобождаем ресурс waiting */
```

```
     cut_hair();        /* процесс стрижки */
}
void customer()
{
     down(mutex);  /* получаем доступ к waiting */
          if (waiting < CHAIRS)  /* есть место для ожидания
                                 */
     {
               waiting = waiting + 1; /* увеличиваем кол-во
                              ожидающих клиентов */
               up(customers);          /* если парикмахер
                              спит, это его разбудит */
               up(mutex);         /* освобождаем ресурс
                              waiting */
               down(barbers);     /* если парикмахер занят,
                              переходим в состояние ожидания,
                              иначе – занимаем парикмахера*/
          get_haircut();          /* процесс стрижки */
     }
     else
     {
           up(mutex);         /* нет свободного кресла для
                              ожидания – придется уйти */
     }
}
```

БИЛЕТ 36 Сигналы. Примеры программирования.

Сигналы.

Сигналы представляют собой средство уведомления процесса о наступлении некоторого события в системе. Инициатором посылки сигнала может выступать как другой процесс, так и сама ОС. Сигналы, посылаемые ОС, уведомляют о наступлении некоторых строго предопределенных ситуаций (как, например,

завершение порожденного процесса, прерывание процесса нажатием комбинации Ctrl-C, попытка выполнить недопустимую машинную инструкцию, попытка недопустимой записи в канал и т.п.), при этом каждой такой ситуации сопоставлен свой сигнал. Кроме того, зарезервирован один или несколько номеров сигналов, семантика которых определяется пользовательскими процессами по своему усмотрению (например, процессы могут посылать друг другу сигналы с целью синхронизации).

Количество различных сигналов в современных версиях UNIX около 30, каждый из них имеет уникальное имя и номер. Описания представлены в файле **<signal.h>**. В таблице приведено несколько примеров сигналов:

Числовое значение	Константа	Значение сигнала
2	SIGINT	Прерывание выполнения по нажатию Ctrl-C
3	SIGQUIT	Аварийное завершение работы
9	SIGKILL	Уничтожение процесса
14	SIGALRM	Прерывание от программного таймера
18	SIGCHLD	Завершился процесс-потомок

Сигналы являются механизмом асинхронного взаимодействия, т.е. момент прихода сигнала процессу заранее неизвестен. Однако процесс может предвидеть возможность получения того или иного сигнала и установить определенную реакцию на его приход. В этом плане сигналы можно рассматривать как программный аналог аппаратных прерываний.

При получении сигнала процессом возможны три варианта реакции на полученный сигнал:

- Процесс реагирует на сигнал стандартным образом, установленным по умолчанию (для большинства сигналов действие по умолчанию – это завершение процесса).
- Процесс может установить специальную обработку сигнала, в этом случае по приходу сигнала вызывается функция-обработчик, определенная процессом (при этом говорят, что сигнал перехватывается)
- Процесс может проигнорировать сигнал.

Для каждого сигнала процесс может устанавливать свой вариант реакции, например, некоторые сигналы он может игнорировать, некоторые перехватывать, а на остальные установить реакцию по умолчанию. При этом в процессе своей работы процесс может изменять вариант реакции на тот или иной сигнал. Однако, необходимо отметить, что некоторые сигналы невозможно ни перехватить, ни игнорировать. Они используются ядром ОС для управления работой процессов (например, **SIGKILL, SIGSTOP**).

Если в процесс одновременно доставляется несколько различных сигналов, то порядок их обработки не определен. Если же обработки ждут несколько экземпляров одного и того же сигнала, то ответ на вопрос, сколько экземпляров будет доставлено в процесс – все или один – зависит от конкретной реализации ОС.

Отдельного рассмотрения заслуживает ситуация, когда сигнал приходит в момент выполнения системного вызова. Обработка такой ситуации в разных версиях UNIX реализована по-разному, например, обработка сигнала может быть отложена до завершения системного вызова; либо системный вызов автоматически перезапускается после его прерывания сигналом; либо системный вызов вернет –1, а в переменной **errno** будет установлено значение **EINTR**

Для отправки сигнала существует системный вызов **kill()**:

```
#include <sys/types.h>
    #INCLUDE <SIGNAL.H>
    INT KILL (PID_T PID, INT SIG)
```

Первым параметром вызова служит идентификатор процесса, которому посылается сигнал (в частности, процесс может послать сигнал самому себе). Существует также возможность одновременно послать сигнал нескольким процессам, например, если значение этого параметра есть 0, сигнал будет передан всем процессам, которые принадлежат той же группе, что и процесс, посылающий сигнал, за исключением процессов с идентификаторами 0 и 1.

Во втором параметре передается номер посылаемого сигнала. Если этот параметр равен 0, то будет выполнена проверка корректности обращения к **kill()** (в частности, существование процесса с идентификатором **pid**), но никакой сигнал в действительности посылаться не будет.

Если процесс-отправитель не обладает правами привилегированного пользователя, то он может отправить сигнал только тем процессам, у которых реальный или эффективный идентификатор владельца процесса совпадает с реальным или эффективным идентификатором владельца процесса-отправителя.

Для определения реакции на получение того или иного сигнала в процессе служит системный вызов **signal()**:

```
    #INCLUDE <SIGNAL.H>
void (*signal  ( int sig, void (*disp) (int))) (int)
```

где аргумент **sig** — номер сигнала, для которого устанавливается реакция, а **disp** — либо определенная пользователем функция-обработчик сигнала, либо одна из констант: **SIG_DFL** и **SIG_IGN**. Первая из них указывает, что необходимо установить для данного сигнала обработку по умолчанию, т.е. стандартную реакцию системы, а вторая — что данный сигнал необходимо игнорировать. При успешном завершении функция возвращает указатель на предыдущий обработчик данного сигнала (он может использоваться процессом, например, для восстановления прежней реакции на сигнал).

Как видно из прототипа вызова **signal()**, определенная пользователем функция-обработчик сигнала должна принимать один целочисленный аргумент (в нем будет передан номер обрабатываемого сигнала), и не возвращать никаких значений.

Отметим одну особенность реализации сигналов в ранних версиях UNIX: каждый раз при получении сигнала его диспозиция (т.е. действие при получении сигнала) сбрасывается на действие по умолчанию, т.о. если процесс желает многократно обрабатывать сигнал своим собственным обработчиком, он должен каждый раз при обработке сигнала заново устанавливать реакцию на него (см.)

В заключении отметим, что механизм сигналов является достаточно ресурсоемким, ибо отправка сигнала представляет собой системный вызов, а доставка сигнала - прерывание выполнения процесса-получателя. Вызов функции-обработчика и возврат требует операций со стеком. Сигналы также несут весьма ограниченную информацию.

2. Обработка сигнала.

В данном примере при получении сигнала **SIGINT** четырежды вызывается специальный обработчик, а в пятый раз происходит обработка по умолчанию.

```
#include <sys/types.h>
#include <signal.h>
#include <stdio.h>
int count = 0;
void SigHndlr (int s)   /* обработчик сигнала */
{
    printf("\n I got SIGINT %d time(s) \n",
        ++ count);
    if (count == 5) signal (SIGINT, SIG_DFL);
    /* ставим обработчик сигнала по умолчанию */
    else signal (SIGINT, SigHndlr);
    /* восстанавливаем обработчик сигнала */
}

int main(int argc, char **argv)
{
    signal (SIGINT, SigHndlr); /* установка реакции на
сигнал */
    while (1); /*”тело программы” */
    return 0;
}
```

3. Программа “Будильник”.

Программа “Будильник”. Существуют задачи, в которых необходимо прервать выполнение процесса по истечении некоторого количества времени. Средствами ОС “заводится” будильник, который будет поторапливать ввести некоторое имя. Системный вызов **alarm()**:

```
#include <unistd.h>
unsigned int alarm(unsigned int seconds);
```

инициализирует отложенное появление сигнала **SIGALRM** - процесс запрашивает ядро отправить ему самому сигнал по прошествии определенного времени**.**

```
#include <unistd.h>
#include <signal.h>
#include <stdio.h>

void alrm(int s) /*обработчик сигнала SIG_ALRM */
{
```

```
     printf("\n жду имя \n");
     alarm(5); /* заводим будильник */
          signal(SIGALRM, alrm); /* переустанавливаем
          реакцию на сигнал */
}

int main(int argc, char **argv)
{
     char s[80];
     signal(SIGALRM, alrm);
     /* установка обработчика alrm на приход сигнала
     SIG_ALRM */
     alarm(5); /* заводим будильник */
     printf("Введите имя \n");
     for (;;)
     {
          printf("имя:");
               if (gets(s) != NULL) break; /* ожидаем ввода
               имени */
     };
     printf("OK! \n");
     return 0;
}
```

В начале программы мы устанавливаем реакцию на сигнал **SIGALRM** - функцию **alrm()**, далее мы заводим будильник, запрашиваем "*Введите имя*" и ожидаем ввода строки символов. Если ввод строки задерживается, то будет вызвана функция **alrm()**, которая напомнит, что программа "ждет имя", опять заведет будильник и поставит себя на обработку сигнала **SIGALRM** еще раз. И так будет до тех пор, пока не будет введена строка. Здесь имеется один нюанс: если в момент выполнения системного вызова возникает событие, связанное с сигналом, то система прерывает выполнение системного вызова и возвращает код ответа, равный «-1».

4. Двухпроцессный вариант программы "Будильник".

```
     #include <signal.h>
     #include <sys/types.h>
     #include <unistd.h>
     #include <stdio.h>

void alr(int s)
{
     printf("\n Быстрее!!! \n");
     signal(SIGALRM, alr);
     /* переустановка обработчика alr на приход сигнала
     SIGALRM */
}
```

```
int main(int argc, char **argv)
{
     char s[80];
     int pid;

     signal(SIGALRM, alr);
     /* установка обработчика alr на приход сигнала SIGALRM
     */
     if  (pid = fork()) {
          for (;;)
          {
                    sleep(5);     /*приостанавливаем процесс
                         на 5 секунд */
               kill(pid, SIGALRM);
                    /*отправляем сигнал SIGALRM процессу-
                    сыну */
          }
     }
     else {
          printf("Введите имя \n");
          for (;;)
          {
               printf("имя:");
                    if (gets(s) != NULL) break; /*ожидаем
                    ввода имени*/
          }
          printf("OK!\n");
          kill(getppid(), SIGKILL);
          /* убиваем зациклившегося отца */
     }
     return 0;
}
```

В данном случае программа реализуется в двух процессах. Как и в предыдущем примере, имеется функция реакции на сигнал **`alr()`**, которая выводит на экран сообщение и переустанавливает функцию реакции на сигнал опять же на себя. В основной программе мы также указываем **`alr()`** как реакцию на **`SIGALRM`**. После этого мы запускаем сыновний процесс, и отцовский процесс (бесконечный цикл) "засыпает" на 5 единиц времени, после чего сыновнему процессу будет отправлен сигнал **`SIGALRM`.** Все, что ниже цикла, будет выполняться в процессе-сыне: мы ожидаем ввода строки, если ввод осуществлен, то происходит уничтожение отца (**`SIGKILL`**).

БИЛЕТ 37 неименованные каналы

Программные каналы

Одним из простейших средств взаимодействия процессов в операционной системе UNIX является механизм каналов. Неименованный канал есть некая сущность, в которую можно помещать и извлекать данные, для чего служат два файловых дескриптора, ассоциированных с каналом: один для записи в канал, другой — для чтения. Для создания канала служит системный вызов **`pipe()`**:

```
#include <unistd.h>

    INT PIPE(INT *FD)
```

Данный системный вызов выделяет в оперативной памяти некоторое ограниченное пространство и возвращает через параметр **`fd`** массив из двух файловых дескрипторов: один для записи в канал — **`fd[1]`**, другой для чтения — **`fd[0]`**. Эти дескрипторы являются дескрипторами открытых файлов, с которыми можно работать, используя такие системные вызовы как **`read()`**, **`write()`**, **`dup()`** и так далее**.** Однако следует четко понимать различия между обычным файлом и каналом.

Основные отличительные свойства канала следующие:

- В отличие от файла, к неименованному каналу невозможен доступ по имени, т.е. единственная возможность использовать канал – это те файловые дескрипторы, которые с ним ассоциированы
- Канал не существует вне процесса, т.е. для существования канала необходим процесс, который его создаст и в котором он будет существовать, для файла это не так.
- Канал реализует модель последовательного доступа к данным (FIFO), т.е. данные из канала можно прочитать только в той же последовательности, в каком они были записаны. Это означает, что для файловых дескрипторов, ассоциированных с каналом, не определена операция **`lseek()`** (при попытке обратиться к этому вызову произойдет ошибка).

Кроме того, существует ряд отличий в поведении операций чтения и записи в канал, а именно:

При чтении из канала:

- если прочитано меньше байтов, чем находится в канале, оставшиеся сохраняются в канале;
- если делается попытка прочитать больше данных, чем имеется в канале, и при этом существуют открытые дескрипторы записи, ассоциированные с каналом, будет прочитано (т.е. изъято из канала) доступное количество данных, после чего читающий процесс блокируется до тех пор, пока в канале

не появится достаточное количество данных для завершения операции чтения.

- процесс может избежать такого блокирования, изменив для канала режим блокировки с использованием системного вызова **`fcntl()`**. В неблокирующем режиме в ситуации, описанной выше, будет прочитано доступное количество данных, и управление будет сразу возвращено процессу.
- При закрытии записывающей стороны канала, в него помещается символ EOF. После этого процесс, осуществляющий чтение, может выбрать из канала все оставшиеся данные и признак конца файла, благодаря которому блокирования при чтении в этом случае не происходит.

При записи в канал:

- если процесс пытается записать большее число байтов, чем помещается в канал (но не превышающее предельный размер канала) записывается возможное количество данных, после чего процесс, осуществляющий запись, блокируется до тех пор, пока в канале не появится достаточное количество места для завершения операции записи;
- процесс может избежать такого блокирования, изменив для канала режим блокировки с использованием системного вызова **`fcntl()`**. В неблокирующем режиме в ситуации, описанной выше, будет записано возможное количество данных, и управление будет сразу возвращено процессу.
- если же процесс пытается записать в канал порцию данных, превышающую предельный размер канала, то будет записано доступное количество данных, после чего процесс заблокируется до появления в канале свободного места любого размера (пусть даже и всего 1 байт), затем процесс разблокируется, вновь производит запись на доступное место в канале, и если данные для записи еще не исчерпаны, вновь блокируется до появления свободного места и т.д., пока не будут записаны все данные, после чего происходит возврат из вызова **`write()`**
- если процесс пытается осуществить запись в канал, с которым не ассоциирован ни один дескриптор чтения, то он получает сигнал **`SIGPIPE`** (тем самым ОС уведомляет его о недопустимости такой операции)**.**

В стандартной ситуации (при отсутствии переполнения) система гарантирует атомарность операции записи, т. е. при одновременной записи нескольких процессов в канал их данные не перемешиваются.

5. Использование канала.

Пример использования канала в рамках одного процесса – копирование строк. Фактически осуществляется посылка данных самому себе.

```
#include <unistd.h>
#include <stdio.h>
int main(int argc, char **argv)
{
     char *s = ”chanel”;
     char buf[80];
     int pipes[2];
     pipe(pipes);
```

```
    write(pipes[1], s, strlen(s) + 1);
    read(pipes[0], buf, strlen(s) + 1);
    close(pipes[0]);
    close(pipes[1]);
    printf("%s\n", buf);
    return 0;
}
```

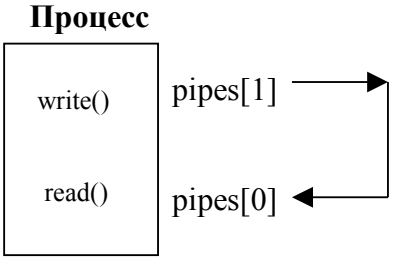


Рис. 7 Обмен через канал в рамках одного процесса.

Чаще всего, однако, канал используется для обмена данными между несколькими процессами. При организации такого обмена используется тот факт, что при порождении сыновнего процесса посредством системного вызова **`fork()`** наследуется таблица файловых дескрипторов процесса-отца, т.е. все файловые дескрипторы, доступные процессу-отцу, будут доступны и процессу-сыну. Таким образом, если перед порождением потомка был создан канал, файловые дескрипторы для доступа к каналу будут унаследованы и сыном. В итоге обоим процессам оказываются доступны дескрипторы, связанные с каналом, и они могут использовать канал для обмена данными (см. Рис. 8 и).

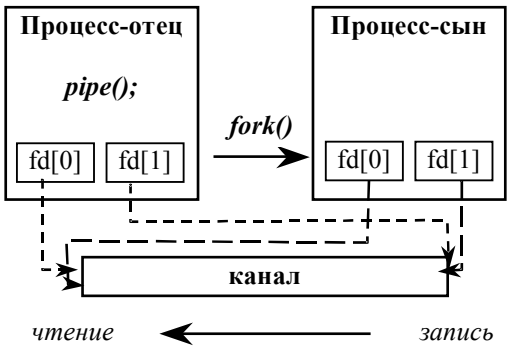


Рис. 8 Пример обмена данными между процессами через канал.

6. Схема взаимодействия процессов с использованием канала.

```
#include <sys/types.h>
#include <unistd.h>

int main(int argc, char **argv)
{
    int fd[2];
    pipe(fd);
    if (fork())
```

```
     {/*процесс-родитель*/
               close(fd[0]); /* закрываем ненужный дескриптор
               */
          write (fd[1], …);
          …
          close(fd[1]);
          …
     }
     else
     {/*процесс-потомок*/
               close(fd[1]); /* закрываем ненужный дескриптор
               */
          while(read (fd[0], …))
          {
               …
          }
          …
     }
}
```

Аналогичным образом может быть организован обмен через канал между двумя потомками одного порождающего процесса и вообще между любыми родственными процессами, единственным требованием здесь, как уже говорилось, является необходимость создавать канал в порождающем процессе прежде, чем его дескрипторы будут унаследованы порожденными процессами.

Как правило, канал используется как однонаправленное средство передачи данных, т.е. только один из двух взаимодействующих процессов осуществляет запись в него, а другой процесс осуществляет чтение, при этом каждый из процессов закрывает не используемый им дескриптор. Это особенно важно для неиспользуемого дескриптора записи в канал, так как именно при закрытии пишущей стороны канала в него помещается символ конца файла. Если, к примеру, в рассмотренном процесс-потомок не закроет свой дескриптор записи в канал, то при последующем чтении из канала, исчерпав все данные из него, он будет заблокирован, так как записывающая сторона канала будет открыта, и следовательно, читающий процесс будет ожидать очередной порции данных.

7. Реализация конвейера.

Пример реализации конвейера **`print|wc`** – вывод программы **`print`** будет подаваться на вход программы **`wc`**. Программа **`print`** печатает некоторый текст. Программа **`wc`** считает количество прочитанных строк, слов и символов.

```
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>
int main(int argc, char **argv)
{
     int fd[2];
     pipe(fd);  /*организован канал*/
     if (fork())
     {
```

```
        /*процесс-родитель*/
             dup2(fd[1], 1); /* отождествили стандартный
             вывод с файловым дескриптором канала,
             предназначенным для записи */
             close(fd[1]);   /* закрыли файловый дескриптор
             канала, предназначенный для записи */
             close(fd[0]);   /* закрыли файловый дескриптор
             канала, предназначенный для чтения */
             exelp("print", "print", 0); /* запустили
             программу print */
    }
    /*процесс-потомок*/
        dup2(fd[0], 0); /* отождествили стандартный ввод с
        файловым дескриптором канала,   предназначенным
        для чтения*/
        close(fd[0]);   /* закрыли файловый дескриптор
        канала, предназначенный для чтения */
        close(fd[1]);  /* закрыли файловый дескриптор
        канала, предназначенный для записи */
        execl("/usr/bin/wc", "wc", 0); /* запустили
        программу wc */
}
```

8. Совместное использование сигналов и каналов – «пинг-понг».

Пример программы с использованием каналов и сигналов для осуществления связи между процессами – весьма типичной ситуации в системе. При этом на канал возлагается роль среды двусторонней передачи информации, а на сигналы – роль системы синхронизации при передаче информации. Процессы посылают друг другу целое число, всякий раз увеличивая его на 1. Когда число достигнет некоего максимума, оба процесса завершаются.

```
#include <signal.h>
    #include <sys/types.h>
    #include <sys/wait.h>
    #include <unistd.h>
    #include <stdlib.h>
#include <stdio.h>

#define MAX_CNT 100
int target_pid, cnt;
int fd[2];
int status;

void SigHndlr(int s)
{
        /* в обработчике сигнала происходит и чтение, и
        запись */
```

```
    signal(SIGUSR1, SigHndlr);

    if (cnt < MAX_CNT)
    {
        read(fd[0], &cnt, sizeof(int));
        printf("%d \n", cnt);
        cnt++;
        write(fd[1], &cnt, sizeof(int));
            /* посылаем сигнал второму: пора читать из
            канала */
        kill(target_pid, SIGUSR1);
    }
    else
        if (target_pid == getppid())
        {
                /* условие окончания игры проверяется
                потомком */
                printf("Child is going to be
                terminated\n");
            close(fd[1]); close(fd[0]);
            /* завершается потомок */
            exit(0);
        } else
            kill(target_pid, SIGUSR1);
}

int main(int argc, char **argv)
{
    pipe(fd); /* организован канал */
    signal (SIGUSR1, SigHndlr);
        /* установлен обработчик сигнала для обоих
        процессов */
    cnt = 0;

    if (target_pid = fork())
    {
            /* Предку остается только ждать завершения
            потомка */
        while(wait(&status) == -1);
            printf("Parent is going to be terminated\n");
        close(fd[1]); close(fd[0]);
        return 0;
    }
    else
    {
        /* процесс-потомок узнает PID родителя */
        target_pid = getppid();
        /* потомок начинает пинг-понг */
        write(fd[1], &cnt, sizeof(int));
```

```
            kill(target_pid, SIGUSR1);
            for(;;); /* бесконечный цикл */
        }
}
```

Билет 38

Именованные каналы (FIFO)

Рассмотренные выше программные каналы имеют важное ограничение: так как доступ к ним возможен только посредством дескрипторов, возвращаемых при порождении канала, необходимым условием взаимодействия процессов через канал является передача этих дескрипторов по наследству при порождении процесса. Именованные каналы (FIFO-файлы) расширяют свою область применения за счет того, что подключиться к ним может любой процесс в любое время, в том числе и после создания канала. Это возможно благодаря наличию у них имен**.**

FIFO-файл представляет собой отдельный тип файла в файловой системе UNIX, который обладает всеми атрибутами файла, такими как имя владельца, права доступа и размер. Для его создания в UNIX System V.3 и ранее используется системный вызов **`mknod()`**, а в BSD UNIX и System V.4 – вызов **`mkfifo()`** (этот вызов поддерживается и стандартом POSIX):

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
     INT MKNOD (CHAR *PATHNAME, MODE_T MODE, DEV);

#include <sys/types.h>
#include <sys/stat.h>
     INT MKFIFO (CHAR *PATHNAME, MODE_T MODE);
```

В обоих вызовах первый аргумент представляет собой имя создаваемого канала, во втором указываются права доступа к нему для владельца, группы и прочих пользователей, и кроме того, устанавливается флаг, указывающий на то, что создаваемый объект является именно FIFO-файлом (в разных версиях ОС он может иметь разное символьное обозначение – **`S_IFIFO`** или **`I_FIFO`**). Третий аргумент вызова **`mknod()`** игнорируется.

После создания именованного канала любой процесс может установит с ним связь посредством системного вызова **`open()`**. При этом действуют следующие правила:

- если процесс открывает FIFO-файл для чтения, он блокируется до тех пор, пока какой-либо процесс не откроет тот же канал на запись
- если процесс открывает FIFO-файл на запись, он будет заблокирован до тех пор, пока какой-либо процесс не откроет тот же канал на чтение
- процесс может избежать такого блокирования, указав в вызове **`open()`** специальный флаг (в разных версиях ОС он может иметь

разное символьное обозначение – **O_NONBLOCK** или **O_NDELAY**). В этом случае в ситуациях, описанных выше, вызов **open()** сразу же вернет управление процессу

Правила работы с именованными каналами, в частности, особенности операций чтения-записи, полностью аналогичны неименованным каналам.

Ниже рассматривается пример, где один из процессов является сервером, предоставляющим некоторую услугу, другой же процесс, который хочет воспользоваться этой услугой, является клиентом. Клиент посылает серверу запросы на предоставление услуги, а сервер отвечает на эти запросы.

9. Модель «клиент-сервер».

Процесс-сервер запускается на выполнение первым, создает именованный канал, открывает его на чтение в неблокирующем режиме и входит в цикл, пытаясь прочесть что-либо. Затем запускается процесс-клиент, подключается к каналу с известным ему именем и записывает в него свой идентификатор. Сервер выходит из цикла, прочитав идентификатор клиента, и печатает его.

```
/* процесс-сервер*/
#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>

int main(int argc, char **argv)
    {
        int fd;
        int pid;
        mkfifo("fifo", S_IFIFO | 0666);
        /*создали специальный файл FIFO с открытыми для
        всех правами доступа на чтение и запись*/
        fd = open("fifo", O_RDONLY | O_NONBLOCK);
        /* открыли канал на чтение*/
        while  (read (fd, &pid, sizeof(int)) == -1) ;
        printf("Server %d  got message from %d !\n",
        getpid(), pid);
        close(fd);
        unlink("fifo");/*уничтожили именованный канал*/
        return 0;
    }
```

```
     /* процесс-клиент*/
     #include <sys/types.h>
     #include <sys/stat.h>
     #include <unistd.h>
     #include <fcntl.h>
int main(int argc, char **argv)
     {
          int fd;
          int pid = getpid( );
          fd = open("fifo", O_RDWR);
          write(fd, &pid, sizeof(int));
          close(fd);
          return 0;
}
```

Билет 39 Трассировка процессов.

Трассировка процессов.

Обзор форм межпроцессного взаимодействия в UNIX был бы не полон, если бы мы не рассмотрели простейшую форму взаимодействия, используемую для отладки — трассировку процессов. Принципиальное отличие трассировки от остальных видов межпроцессного взаимодействия в том, что она реализует модель «главный-подчиненный»: один процесс получает возможность управлять ходом выполнения, а также данными и кодом другого.

В UNIX трассировка возможна только между родственными процессами: процесс-родитель может вести трассировку только непосредственно порожденных им потомков, при этом трассировка начинается только после того, как процесс-потомок дает разрешение на это.

Далее схема взаимодействия процессов путем трассировки такова: выполнение отлаживаемого процесса-потомка приостанавливается всякий раз при получении им какого-либо сигнала, а также при выполнении вызова **`exec()`**. Если в это время отлаживающий процесс осуществляет системный вызов **`wait()`**, этот вызов немедленно возвращает управление. В то время, как трассируемый процесс находится в приостановленном состоянии, процесс-отладчик имеет возможность анализировать и изменять данные в адресном пространстве отлаживаемого процесса и в пользовательской составляющей его контекста. Далее, процесс-отладчик возобновляет выполнение трассируемого процесса до следующей приостановки (либо, при пошаговом выполнении, для выполнения одной инструкции).

Основной системный вызов, используемый при трассировке,– это **`ptrace()`**, прототип которого выглядит следующим образом:

```
#include <sys/ptrace.h>
int ptrace(int cmd, pid, addr, data);
```

где **cmd** – код выполняемой команды, **pid** – идентификатор процесса-потомка, **addr** – некоторый адрес в адресном пространстве процесса-потомка, **data** – слово информации.
Чтобы оценить уровень предоставляемых возможностей, рассмотрим основные коды - **cmd** операций этой функции.
cmd = PTRACE_TRACEME — **ptrace()** с таким кодом операции сыновний процесс вызывает в самом начале своей работы, позволяя тем самым трассировать себя**.** Все остальные обращения к вызову **ptrace()** осуществляет процесс-отладчик.
cmd = PTRACE_PEEKDATA — чтение слова из адресного пространства отлаживаемого процесса по адресу **addr**, **ptrace()** возвращает значение этого слова.
cmd = PTRACE_PEEKUSER — чтение слова из контекста процесса**.** Речь идет о доступе к пользовательской составляющей контекста данного процесса, сгруппированной в некоторую структуру, описанную в заголовочном файле **<sys/user.h>**. В этом случае параметр **addr** указывает смещение относительно начала этой структуры. В этой структуре размещена такая информация, как регистры, текущее состояние процесса, счетчик адреса и так далее. **ptrace()** возвращает значение считанного слова.
cmd = PTRACE_POKEDATA — запись данных, размещенных в параметре **data**, по адресу **addr** в адресном пространстве процесса-потомка.
cmd = PTRACE_POKEUSER — запись слова из **data** в контекст трассируемого процесса со смещением **addr**. Таким образом можно, например, изменить счетчик адреса трассируемого процесса, и при последующем возобновлении трассируемого процесса его выполнение начнется с инструкции, находящейся по заданному адресу.
cmd = PTRACE_GETREGS, PTRACE_GETFREGS — чтение регистров общего назначения (в т.ч. с плавающей точкой) трассируемого процесса и запись их значения по адресу **data**.
cmd = PTRACE_SETREGS, PTRACE_SETFREGS — запись в регистры общего назначения (в т.ч. с плавающей точкой) трассируемого процесса данных, расположенных по адресу **data** в трассирующем процессе.
cmd = PTRACE_CONT — возобновление выполнения трассируемого процесса**.** Отлаживаемый процесс будет выполняться до тех пор, пока не получит какой-либо сигнал, либо пока не завершится.
cmd = PTRACE_SYSCALL, PTRACE_SINGLESTEP — эта команда, аналогично **PTRACE_CONT**, возобновляет выполнение трассируемой программы, но при этом произойдет ее остановка после того, как выполнится одна инструкция. Таким образом, используя **PTRACE_SINGLESTEP**, можно организовать пошаговую отладку. С помощью команды **PTRACE_SYSCALL** возобновляется выполнение трассируемой программы вплоть до ближайшего входа или выхода из системного вызова. Идея использования **PTRACE_SYSCALL** в том, чтобы иметь возможность контролировать значения аргументов, переданных в системный вызов трассируемым процессом, и возвращаемое значение, переданное ему из системного вызова.
cmd = PTRACE_KILL — завершение выполнения трассируемого процесса.

Общая схема использования механизма трассировки.

Рассмотрим некоторый модельный пример, демонстрирующий общую схему построения отладочной программы (см. также Рис. 9):

```
     ...
if ((pid = fork()) == 0)
{
     ptrace(PTRACE_TRACEME, 0, 0, 0);
          /* сыновний процесс разрешает трассировать себя */
     exec("трассируемый процесс", 0);
          /* замещается телом процесса, который необходимо
          трассировать */
}
else
{
     /* это процесс, управляющий трассировкой */
     wait((int ) 0);
          /* процесс приостанавливается до тех пор, пока от
          трассируемого процесса не придет сообщение о том,
          что он приостановился */

     for(;;)
          {
          ptrace(PTRACE_SINGLESTEP, pid, 0, 0);
               /* возобновляем выполнение трассируемой
               программы */
          wait((int ) 0);
               /* процесс приостанавливается до тех пор, пока
               от трассируемого процесса не придет сообщение
               о том, что он приостановился */
          …
          ptrace(cmd, pid, addr, data);
               /* теперь выполняются любые действия над
               трассируемым процессом */
          …
     }
}
```

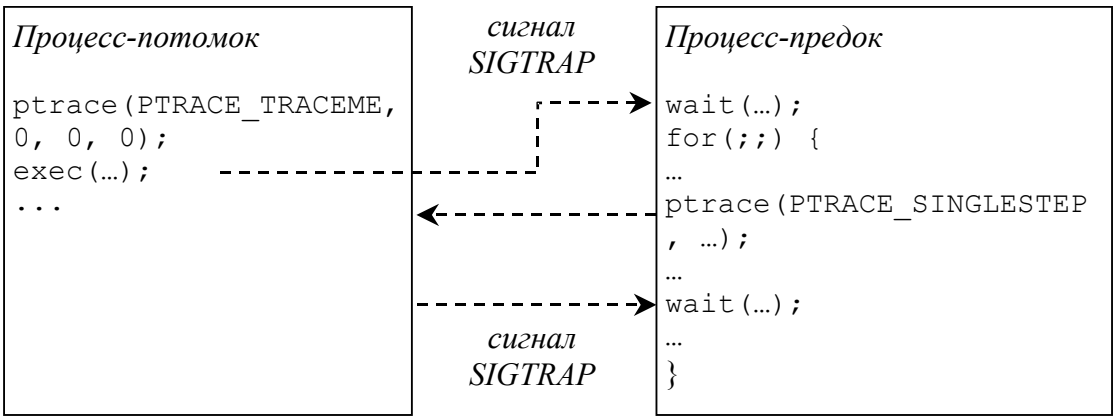


Рис. 9 Общая схема трассировки процессов

Предназначение процесса-потомка — разрешить трассировку себя. После вызова **ptrace(PTRACE_TRACEME, 0, 0, 0)** ядро устанавливает для этого процесса бит трассировки. Сразу же после этого можно заместить код процесса-потомка кодом программы, которую необходимо отладить. Отметим, что при выполнении системного вызова **exec()**, если для данного процесса ранее был установлен бит трассировки, ядро перед передачей управления в новую программу посылает процессу сигнал **SIGTRAP**. При получении данного сигнала трассируемый процесс приостанавливается, и ядро передает управление процессу-отладчику, выводя его из ожидания в вызове **wait()**.
Процесс-родитель вызывает **wait()** и переходит в состояние ожидания до того момента, пока потомок не перейдет в состояние трассировки. Проснувшись, управляющий процесс, выполняя функцию **ptrace(cmd, pid, addr, data)** с различными кодами операций, может производить любое действие с трассируемой программой, в частности, читать и записывать данные в адресном пространстве трассируемого процесса, а также разрешать дальнейшее выполнение трассируемого процесса или производить его пошаговое выполнение. Схема пошаговой отладки показана в примере выше и на рисунке: на каждом шаге процесс-отладчик разрешает выполнение очередной инструкции отлаживаемого процесса и затем вызывает **wait()** и погружается в состояние ожидания, а ядро возобновляет выполнение трассируемого потомка, исполняет трассируемую команду и вновь передает управление отладчику, выводя его из ожидания .

10. Трассировка процессов.

```
/* Процесс-сын: */
int main(int argc, char **argv)
{
          /* деление на ноль – здесь процессу будет послан
          сигнал SIGFPE – floating point exception */
     return argc/0;
}

Процесс-родитель:
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
#include <signal.h>
#include <sys/ptrace.h>
#include <sys/user.h>
#include <sys/wait.h>
int main(int argc, char *argv[])
{
     pid_t pid;
     int status;
     struct user_regs_struct REG;
     if  ((pid = fork()) == 0) {
               /*находимся в процессе-потомке, разрешаем
               трассировку */
          ptrace(PTRACE_TRACEME, 0, 0, 0);
```

```
                execl("son", "son", 0);      /* замещаем тело
                процесса */
                /* здесь процесс-потомок будет остановлен с
                сигналом SIG_TRAP, ожидая команды продолжения
                выполнения от управляющего процесса*/
        }
        /* в процессе-родителе */
        while (1) {
                /* ждем, когда отлаживаемый процесс
                приостановится */
            wait(&status);
                /*читаем содержимое регистров отлаживаемого
                процесса */
            ptrace(PTRACE_GETREGS, pid, &REG, &REG);
                /* выводим статус отлаживаемого процесса,
                номер сигнала, который его остановил и
                значения прочитанных регистров  */
                printf("signal = %d, status = %#x, EIP=%#x
                ESP=%#x\n", WSTOPSIG(status), status, REG.eip,
                REG.esp);
            if (WSTOPSIG(status) != SIGTRAP) {
                if (!WIFEXITED(status)) {
                        /* завершаем выполнение
                        трассируемого процесса */
                    ptrace (PTRACE_KILL, pid, 0, 0);
                }
                break;
            }
                /* разрешаем выполнение трассируемому процессу
                */
            ptrace (PTRACE_CONT, pid, 0, 0);
        }
}
```

ДЛЯ БИЛЕТОВ 40-42 ОБЩАЯ ЧАСТЬ

Именование разделяемых объектов.

Для всех средств IPC приняты общие правила именования объектов, позволяющие процессу получить доступ к такому объекту. Для именования объекта IPC используется ключ, представляющий собой целое число. Ключи являются уникальными во всей UNIX-системе идентификаторами объектов IPC, и зная ключ для некоторого объекта, процесс может получить к нему доступ. При этом процессу возвращается дескриптор объекта, который в дальнейшем используется для всех операций с ним. Проведя аналогию с файловой системой, можно сказать, что ключ аналогичен имени файла, а получаемый по ключу дескриптор – файловому дескриптору, получаемому во время операции открытия файла. Ключ для каждого объекта IPC задается в момент его создания тем процессом, который его порождает, а все процессы, желающие получить в дальнейшем доступ к этому объекту, должны указывать тот же самый ключ.

Итак, все процессы, которые хотят работать с одним и тем же IPC-ресурсом, должны знать некий целочисленный ключ, по которому можно получить к нему доступ. В принципе, программист, пишущий программы для работы с разделяемым ресурсом, может просто жестко указать в программе некоторое константное значение ключа для именования разделяемого ресурса. Однако, возможна ситуация, когда к моменту запуска такой программы в системе уже существует разделяемый ресурс с таким значением ключа, и в виду того, что ключи должны быть уникальными во всей системе, попытка породить второй ресурс с таким же ключом закончится неудачей (подробнее этот момент будет рассмотрен ниже).

Генерация ключей: функция ftok().

Как видно, встает проблема именования разделяемого ресурса: необходим некий механизм получения заведомо уникального ключа для именования ресурса, но вместе с тем нужно, чтобы этот механизм позволял всем процессам, желающим работать с одним ресурсом, получить одно и то же значение ключа.

Для решения этой задачи служит функция **`ftok()`**:

```
#include <sys/types.h>
#include <sys/ipc.h>

     key_t ftok(char *filename, char proj);
```

Эта функция генерирует значение ключа по некоторой строке символов и добавочному символу, передаваемым в качестве параметров. Гарантируется, что полученное таким образом значение будет отличаться от всех других значений, сгенерированных функцией **`ftok()`** с другими значениями параметров, и в то же время, при повторном запуске **`ftok()`** с теми же параметрами, будет получено то же самое значение ключа.

Смысл второго аргумента функции **`ftok()`** – добавочного символа – в том, что он позволяет генерировать разные значения ключа по одному и тому же значению первого параметра – строки. Это позволяет программисту поддерживать несколько версий своей программы, которые будут использовать одну и ту же строку, но разные добавочные символы для генерации ключа, и тем самым получат возможность в рамках одной системы работать с разными разделяемыми ресурсами.

Следует заметить, что функция **ftok()** не является системным вызовом, а предоставляется библиотекой.

Общие принципы работы с разделяемыми ресурсами.

Рассмотрим некоторые моменты, общие для работы со всеми разделяемыми ресурсами IPC. Как уже говорилось, общим для всех ресурсов является механизм именования. Кроме того, для каждого IPC-ресурса поддерживается идентификатор его владельца и структура, описывающая права доступа к нему. Подобно файлам, права доступа задаются отдельно для владельца, его группы и всех остальных пользователей; однако, в отличие от файлов, для разделяемых ресурсов поддерживается только две категории доступа: по чтению и записи. Априори считается, что возможность изменять свойства ресурса и удалять его имеется только у процесса, эффективный идентификатор пользователя которого совпадает с идентификатором владельца ресурса. Владельцем ресурса назначается пользователь, от имени которого выполнялся процесс, создавший ресурс, однако создатель может передать права владельца другому пользователю. В заголовочном файле **<sys/ipc.h>** определен тип **struct ipc_perm**, который описывает права доступа к любому IPC-ресурсу. Поля в этой структуре содержат информацию о создателе и владельце ресурса и их группах, правах доступа к ресурсу и его ключе.

Для создания разделяемого ресурса с заданным ключом, либо подключения к уже существующему ресурсу с таким ключом используются ряд системных вызовов, имеющих общий суффикс **get**. Общими параметрами для всех этих вызовов являются ключ и флаги. В качестве значения ключа при создании любого IPC-объекта может быть указано значение **IPC_PRIVATE**. При этом создается ресурс, который будет доступен только породившему его процессу. Такие ресурсы обычно порождаются родительским процессом, который затем сохраняет полученный дескриптор в некоторой переменной и порождает своих потомков. Так как потомкам доступен уже готовый дескриптор созданного объекта, они могут непосредственно работать с ним, не обращаясь предварительно к «**get**»-методу. Таким образом, созданный ресурс может совместно использоваться родительским и порожденными процессами. Однако, важно понимать, что если один из этих процессов повторно вызовет «**get**»-метод с ключом **IPC_PRIVATE**, в результате будет получен другой, совершенно новый разделяемый ресурс, так как при обращении к «**get**»-методу с ключом **IPC_PRIVATE** всякий раз создается новый объект нужного типа.

Отметим, что даже если ни один процесс не подключен к разделяемому ресурсу, система не удаляет его автоматически. Удаление объектов IPC является обязанностью одного из работающих с ним процессов и для этого определена специальная функция. Для этого системой предоставляются соответствующие функции по управлению объектами System V IPC.

БИЛЕТ 40

Очередь сообщений.

Итак, одним из типов объектов System V IPC являются очереди сообщений. Очередь сообщений представляет собой некое хранилище типизированных сообщений, организованное по принципу FIFO. Любой процесс может помещать новые сообщения в очередь и извлекать из очереди имеющиеся там сообщения. Каждое сообщение имеет тип, представляющий собой некоторое целое число. Благодаря наличию типов сообщений, очередь можно интерпретировать двояко — рассматривать ее либо как сквозную очередь неразличимых по типу сообщений, либо как некоторое объединение подочередей, каждая из которых содержит элементы определенного типа. Извлечение сообщений из очереди происходит согласно принципу FIFO – в порядке их записи, однако процесс-получатель может указать, из какой подочереди он хочет извлечь сообщение, или, иначе говоря, сообщение какого типа он желает получить – в этом случае из очереди будет извлечено самое «старое» сообщение нужного типа (см. Рис. 10).

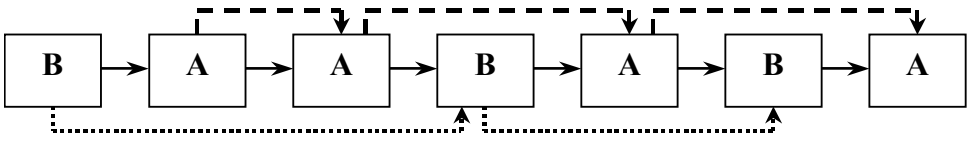


Рис. 10 Типизированные очереди сообщений

Рассмотрим набор системных вызовов, поддерживающий работу с очередями сообщений.

Доступ к очереди сообщений.

Для создания новой или для доступа к существующей используется системный вызов:

```
#INCLUDE <SYS/TYPES.H>
#INCLUDE <SYS/IPC.H>
#INCLUDE <SYS/MESSAGE.H>
INT MSGGET (KEY_T KEY, INT MSGFLAG)
```

В случае успеха вызов возвращает положительный дескриптор очереди, который может в дальнейшем использоваться для операций с ней, в случае неудачи -1. Первым аргументом вызова является ключ, вторым – флаги, управляющие поведением вызова. Подробнее детали процесса создания/подключения к ресурсу описаны выше.

Отправка сообщения.

Для отправки сообщения используется функция **msgsnd():**

```
#INCLUDE <SYS/TYPES.H>
#INCLUDE <SYS/IPC.H>
#INCLUDE <SYS/MSG.H>
INT MSGSND (INT MSQID, CONST VOID *MSGP, SIZE_T MSGSZ,
INT MSGFLG)
```

Ее первый аргумент — идентификатор очереди, полученный в результате вызова **msgget()**. Второй аргумент — указатель на буфер, содержащий реальные данные и тип сообщения, подлежащего посылке в очередь, в третьем аргументе указывается размер буфера.

В качестве буфера необходимо указывать структуру, содержащую следующие поля (в указанном порядке):

long msgtype — тип сообщения

char msgtext[ ] — данные (тело сообщения)

В заголовочном файле **<sys/msg.h>** определена константа **MSGMAX**, описывающая максимальный размер тела сообщения. При попытке отправить сообщение, у которого число элементов в массиве **msgtext** превышает это значение, системный вызов вернет –1.

Четвертый аргумент данного вызова может принимать значения 0 или **IPC_NOWAIT**. В случае отсутствия флага **IPC_NOWAIT** вызывающий процесс будет блокирован (т.е. приостановит работу), если для посылки сообщения недостаточно системных ресурсов, т.е. если полная длина сообщений в очереди будет больше максимально допустимого. Если же флаг **IPC_NOWAIT** будет установлен, то в такой ситуации выход из вызова произойдет немедленно, и возвращаемое значение будет равно –1.

В случае удачной записи возвращаемое значение вызова равно 0.

Получение сообщения.

Для получения сообщения имеется функция **msgrcv():**

```
#INCLUDE <SYS/TYPES.H>
#INCLUDE <SYS/IPC.H>
#INCLUDE <SYS/MSG.H>
INT MSGRCV (INT MSQID, VOID *MSGP, SIZE_T MSGSZ, LONG
MSGTYP, INT MSGFLG)
```

Первые три аргумента аналогичны аргументам предыдущего вызова: это дескриптор очереди, указатель на буфер, куда следует поместить данные, и максимальный размер (в байтах) тела сообщения, которое можно туда поместить. Буфер, используемый для приема сообщения, должен иметь структуру, описанную выше.

Четвертый аргумент указывает тип сообщения, которое процесс желает получить. Если значение этого аргумента есть 0, то будет получено сообщение любого типа. Если значение аргумента **msgtyp** больше 0, из очереди будет извлечено сообщение указанного типа. Если же значение аргумента **msgtyp** отрицательно, то тип принимаемого сообщения определяется как наименьшее значение среди типов, которые меньше модуля **msgtyp**. В любом случае, как уже говорилось, из подочереди с заданным типом (или из общей очереди, если тип не задан) будет выбрано самое старое сообщение.

Последним аргументом является комбинация (побитовое сложение) флагов. Если среди флагов не указан **IPC_NOWAIT**, и в очереди не найдено ни одного сообщения, удовлетворяющего критериям выбора, процесс будет заблокирован до появления такого сообщения. (Однако, если такое сообщение существует, но его длина превышает указанную в аргументе **msgsz**, то процесс заблокирован не будет, и вызов сразу вернет –1. Сообщение при этом останется в очереди). Если же флаг **IPC_NOWAIT** указан, то вызов сразу вернет –1.

Процесс может также указать флаг **MSG_NOERROR** – в этом случае он может прочитать сообщение, даже если его длина превышает указанную емкость буфера. В этом случае в буфер будет записано первые **msgsz** байт из тела сообщения, а остальные данные отбрасываются.

В случае удачного чтения возвращаемое значение вызова равно фактической длине тела полученного сообщения в байтах.

Управление очередью сообщений.

Функция управления очередью сообщений выглядит следующим образом:

```
#INCLUDE <SYS/TYPES.H>
#INCLUDE <SYS/IPC.H>
#INCLUDE <SYS/MSG.H>
INT MSGCTL(INT MSQID, INT CMD, STRUCT MSGID_DS *BUF)
```

Данный вызов используется для получения или изменения процессом управляющих параметров, связанных с очередью и уничтожения очереди. Ее аргументы — идентификатор ресурса, команда, которую необходимо выполнить, и структура, описывающая управляющие параметры очереди. Тип **msgid_ds** описан в заголовочном файле **<sys/message.h>**, и представляет собой структуру, в полях которой хранятся права доступа к очереди, статистика обращений к очереди, ее размер и т.п.

Возможные значения аргумента **cmd**:

IPC_STAT – скопировать структуру, описывающую управляющие параметры очереди по адресу, указанному в параметре **buf**
IPC_SET – заменить структуру, описывающую управляющие параметры очереди, на структуру, находящуюся по адресу, указанному в параметре **buf**
IPC_RMID – удалить очередь. Как уже говорилось, удалить очередь может только процесс, у которого эффективный идентификатор пользователя совпадает с владельцем или создателем очереди, либо процесс с правами привилегированного пользователя.

11. Использование очереди сообщений.

Пример программы, где основной процесс читает некоторую текстовую строку из стандартного ввода, и в случае, если строка начинается с буквы **'a'**, эта строка в качестве сообщения будет передана процессу **A**, если **'b'** - процессу **B,** если **'q'** - то процессам **A** и **B**, затем будет осуществлен выход. Процессы **A** и **B** распечатывают полученные строки на стандартный вывод.

Основной процесс.

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
#include <string.h>
#include <unistd.h>
#include <stdio.h>

struct {
     long mtype;   /* тип сообщения */
     char Data[256];    /* сообщение */
} Message;

int main(int argc, char **argv)
{
     key_t key;  int msgid;  char str[256];

     key = ftok("/usr/mash",'s');
          /*получаем уникальный ключ, однозначно
          определяющий доступ к ресурсу */
     msgid=msgget(key, 0666 | IPC_CREAT);
          /*создаем очередь сообщений , 0666 определяет
          права доступа */

     for(;;) {
          /* запускаем вечный цикл */
               gets(str); /* читаем из стандартного ввода
               строку */
          strcpy(Message.Data, str);
          /* и копируем ее в буфер сообщения */
          switch(str[0]){
               case 'a':
               case 'A':
                    Message.mtype = 1;
```

```
                    /* устанавливаем тип */
                         msgsnd(msgid, (struct msgbuf*)
                         (&Message), strlen(str) + 1, 0);
                         /* посылаем сообщение в очередь */
                    break;
               case 'b':
               case 'B':
                    Message.mtype = 2;
                         msgsnd(msgid, (struct msgbuf*)
                         (&Message), strlen(str) + 1, 0);
                    break;
               case 'q':
               case 'Q':
                    Message.mtype = 1;
                         msgsnd(msgid, (struct msgbuf*)
                         (&Message), strlen(str) + 1, 0);
                    Message.mtype = 2;
                         msgsnd(msgid, (struct msgbuf*)
                         (&Message), strlen(str) + 1, 0);
                    sleep(10);
                         /* ждем получения сообщений
                         процессами A и B */
                    msgctl(msgid, IPC_RMID, NULL);
                    /* уничтожаем очередь*/
                    return 0;
               default:
                    break;
          }
     }
}
```

Процесс-приемник A

```
     /* процесс B аналогичен с точностью до четвертого
     параметра в msgrcv */

     #include <sys/types.h>
     #include <sys/ipc.h>
     #include <sys/msg.h>
     #include <stdio.h>

     struct {
               long mtype;
               char Data[256];
     } Message;

     int main(int argc, char **argv)
     {
     key_t key;  int msgid;

     key = ftok("/usr/mash",'s');
```

```
/* получаем ключ по тем же параметрам */
msgid = msgget(key, 0666 | IPC_CREAT);
/*подключаемся к очереди сообщений */
for(;;) {
     /* запускаем вечный цикл */
          msgrcv(msgid, (struct msgbuf*) (&Message),
          256, 1, 0);
     /* читаем сообщение с типом 1*/
          if (Message.Data[0]=='q' ||
          Message.Data[0]=='Q') break;
          printf("\nПроцесс-приемник A: %s",
          Message.Data);
}
return 0;
}
```

Благодаря наличию типизации сообщений, очередь сообщений предоставляет возможность мультиплексировать сообщения от различных процессов, при этом каждая пара взаимодействующих через очередь процессов может использовать свой тип сообщений, и таким образом, их данные не будут смешиваться.

В качестве иллюстрации приведем следующий стандартный пример взаимодействия. Рассмотрим еще один пример - пусть существует **процесс-сервер** и несколько **процессов-клиентов**. Все они могут обмениваться данными, используя одну очередь сообщений. Для этого сообщениям, направляемым от клиента к серверу, присваиваем значение типа **1**. При этом процесс, отправивший сообщение, в его теле передает некоторую информацию, позволяющую его однозначно идентифицировать. Тогда сервер, отправляя сообщение конкретному процессу, в качестве его типа указывает эту информацию (например, **PID** процесса). Таким образом, сервер будет читать из очереди только сообщения типа **1**, а клиенты — сообщения с типами, равными идентификаторам их процессов.

12. Очередь сообщений. Модель «клиент-сервер»

server

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
#include <string.h>

int main(int argc, char **argv)
{
     struct {
          long mestype;
          char mes [100];
          } messageto;
```

```
        struct {
             long mestype;
             long mes;
             } messagefrom;

        key_t key;
        int mesid;

        key = ftok("example",'r');
        mesid = msgget (key, 0666 | IPC_CREAT);

        while(1)
        {
             if (msgrcv(mesid, &messagefrom,
             sizeof(messagefrom), 1, 0) <= 0) continue;
             messageto.mestype = messagefrom.mes;
             strcpy( messageto.mes, "Message for client");
             msgsnd (mesid, &messageto, sizeof(messageto),
             0);
        }
        msgctl (mesid, IPC_RMID, 0);
        return 0;
    }
```

client

```
    #include <sys/types.h>
    #include <sys/ipc.h>
    #include <sys/msg.h>
    #include <unistd.h>
    #include <stdio.h>

int main(int argc, char **argv)
    {
        struct {
             long mestype;   /*описание структуры
             сообщения*/
             long mes;
```

```
        } messageto;
    struct {
        long mestype;    /*описание структуры
        сообшения*/
        char mes[100];
        } messagefrom;
    key_t key;
    int mesid;
    long pid = getpid();
    key = ftok("example", 'r');
    mesid = msgget(key, 0);    /*присоединение к
    очереди сообщений*/
    messageto.mestype = 1;
    messageto.mes = pid;
    msgsnd (mesid, &messageto, sizeof(messageto),
    0); /* отправка */
    while ( msgrcv (mesid, &messagefrom,
    sizeof(messagefrom), pid, 0) <= 0);
    /*прием сообщения */

    printf("%s\n", messagefrom.mes);
    return 0;
}
```

БИЛЕТ 41

Разделяемая память

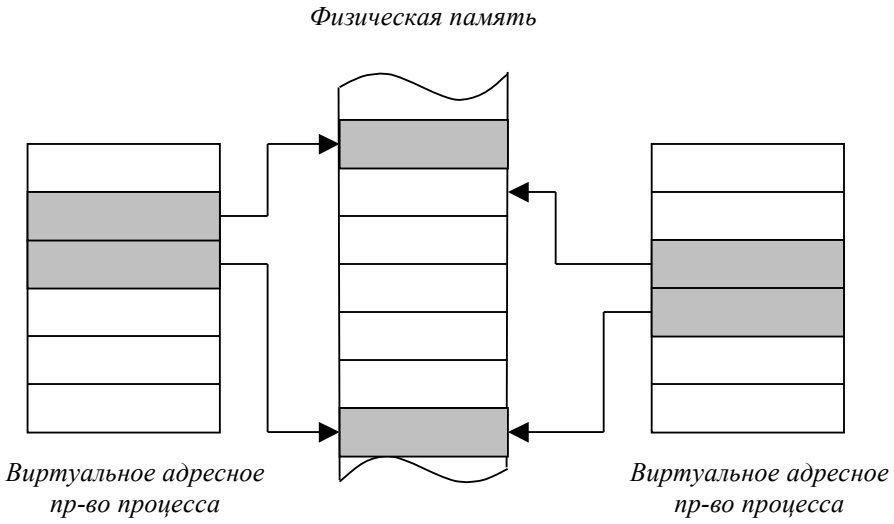


Рис. 11 Разделяемая память

Механизм разделяемой памяти позволяет нескольким процессам получить отображение некоторых страниц из своей виртуальной памяти на общую область физической памяти. Благодаря этому, данные, находящиеся в этой области памяти, будут доступны для чтения и модификации всем процессам, подключившимся к данной области памяти.

Процесс, подключившийся к разделяемой памяти, может затем получить указатель на некоторый адрес в своем виртуальном адресном пространстве, соответствующий данной области разделяемой памяти. После этого он может работать с этой областью памяти аналогично тому, как если бы она была выделена динамически (например, путем обращения к **`malloc()`**), однако, как уже говорилось, сама по себе разделяемая область памяти не уничтожается автоматически даже после того, как процесс, создавший или использовавший ее, перестанет с ней работать.

Рассмотрим набор системных вызовов для работы с разделяемой памятью.

Создание общей памяти.

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
     int shmget (key_t key, int size, int shmemflg)
```

Аргументы этого вызова: **`key`** - ключ для доступа к разделяемой памяти; **`size`** задает размер области памяти, к которой процесс желает получить доступ. Если в результате вызова **`shmget()`** будет создана новая область разделяемой памяти, то ее размер будет соответствовать значению **`size`**. Если же процесс подключается к существующей области разделяемой памяти, то значение **`size`** должно быть не более ее размера, иначе вызов вернет –1. Заметим, что если процесс при подключении к существующей области разделяемой памяти указал в аргументе **`size`** значение, меньшее ее фактического размера, то впоследствии он сможет получить доступ только к первым **`size`** байтам этой области.

Отметим, что в заголовочном файле **`<sys/shm.h>`** определены константы **`SHMMIN`** и **`SHMMAX`**, задающий минимально возможный и максимально возможный размер области разделяемой памяти. Если процесс пытается создать область разделяемой памяти, размер которой не удовлетворяет этим границам, системный вызов **`shmget()`** окончится неудачей.

Третий параметр определяет флаги, управляющие поведением вызова. Подробнее алгоритм создания/подключения разделяемого ресурса был описан выше.

В случае успешного завершения вызов возвращает положительное число – дескриптор области памяти, в случае неудачи - -1.

Доступ к разделяемой памяти.

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
char *shmat(int shmid, char *shmaddr, int shmflg)
```

При помощи этого вызова процесс подсоединяет область разделяемой памяти, дескриптор которой указан в **`shmid`**, к своему виртуальному адресному пространству. После выполнения этой операции процесс сможет читать и

модифицировать данные, находящиеся в области разделяемой памяти, адресуя ее как любую другую область в своем собственном виртуальном адресном пространстве.
В качестве второго аргумента процесс может указать виртуальный адрес в своем адресном пространстве, начиная с которого необходимо подсоединить разделяемую память. Чаще всего, однако, в качестве значения этого аргумента передается 0, что означает, что система сама может выбрать адрес начала разделяемой памяти. Передача конкретного адреса в этом параметре имеет смысл в том случае, если, к примеру, в разделяемую память записываются указатели на нее же (например, в ней хранится связанный список) – в этой ситуации для того, чтобы использование этих указателей имело смысл и было корректным для всех процессов, подключенных к памяти, важно, чтобы во всех процессах адрес начала области разделяемой памяти совпадал.
Третий аргумент представляет собой комбинацию флагов. В качестве значения этого аргумента может быть указан флаг **SHM_RDONLY**, который указывает на то, что подсоединяемая область будет использоваться только для чтения.
Эта функция возвращает адрес, начиная с которого будет отображаться присоединяемая разделяемая память. В случае неудачи вызов возвращает **-1**.

Открепление разделяемой памяти.

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
int shmdt(char *shmaddr)
```

Данный вызов позволяет отсоединить разделяемую память, ранее присоединенную посредством вызова **shmat()**.
Параметр **shmaddr** - адрес прикрепленной к процессу памяти, который был получен при вызове **shmat().**
В случае успешного выполнения функция возвращает 0, в случае неудачи -1

Управление разделяемой памятью.

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
int shmctl(int shmid, int cmd, struct shmid_ds *buf)
```

Данный вызов используется для получения или изменения процессом управляющих параметров, связанных с областью разделяемой памяти, наложения и снятия блокировки на нее и ее уничтожения. Аргументы вызова — дескриптор области памяти, команда, которую необходимо выполнить, и структура, описывающая управляющие параметры области памяти. Тип **shmid_ds** описан в заголовочном файле **<sys/shm.h>**, и представляет собой структуру, в полях которой хранятся права доступа к области памяти, ее размер, число процессов, подсоединенных к ней в данный момент, и статистика обращений к области памяти.
Возможные значения аргумента **cmd**:
IPC_STAT – скопировать структуру, описывающую управляющие параметры области памяти по адресу, указанному в параметре **buf**
IPC_SET – заменить структуру, описывающую управляющие параметры области памяти, на структуру, находящуюся по адресу, указанному в параметре **buf**.
Выполнить эту операцию может процесс, у которого эффективный идентификатор

пользователя совпадает с владельцем или создателем очереди, либо процесс с правами привилегированного пользователя, при этом процесс может изменить только владельца области памяти и права доступа к ней.
IPC_RMID – удалить очередь. Как уже говорилось, удалить очередь может только процесс, у которого эффективный идентификатор пользователя совпадает с владельцем или создателем очереди, либо процесс с правами привилегированного пользователя.
SHM_LOCK, SHM_UNLOCK – блокировать или разблокировать область памяти. Выполнить эту операцию может только процесс с правами привилегированного пользователя.

13. Общая схема работы с общей памятью в рамках одного процесса.

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>

int putm(char *);
int waitprocess(void);

int main(int argc, char **argv)
{
     key_t key;
     int shmid;
     char *shmaddr;

     key = ftok("/tmp/ter",'S');
     shmid = shmget(key, 100, 0666|IPC_CREAT);
          shmaddr = shmat(shmid, NULL, 0); /* подключение к
          памяти */

     putm(shmaddr); /* работа с ресурсом */
     waitprocess();

          shmctl(shmid,IPC_RMID,NULL); /* уничтожение
          ресурса */
     return 0;
}
```

В данном примере считается, что **putm()** и **waitprocess()** – некие пользовательские функции, определенные в другом месте

БИЛЕТ 42

Семафоры.

Семафоры представляют собой одну из форм IPC и, как правило, используются для синхронизации доступа нескольких процессов к разделяемым ресурсам, так как сами по себе другие средства IPC не предоставляют механизма синхронизации. Как уже говорилось, семафор представляет собой особый вид числовой переменной, над которой определены две неделимые операции: уменьшение ее значения с возможным блокированием процесса и увеличение значения с возможным разблокированием одного из ранее заблокированных процессов. Объект System V IPC представляет собой набор семафоров. Как правило, использование семафоров в качестве средства синхронизации доступа к другим разделяемым объектам предполагает следующую схему:

- с каждым разделяемым ресурсом связывается один семафор из набора;
- положительное значение семафора означает возможность доступа к ресурсу (ресурс свободен), неположительное – отказ в доступе (ресурс занят);
- перед тем как обратиться к ресурсу, процесс уменьшает значение соответствующего ему семафора, при этом, если значение семафора после уменьшения должно оказаться отрицательным, то процесс будет заблокирован до тех пор, пока семафор не примет такое значение, чтобы при уменьшении его значение оставалось неотрицательным;
- закончив работу с ресурсом, процесс увеличивает значение семафора (при этом разблокируется один из ранее заблокированных процессов, ожидающих увеличения значения семафора, если таковые имеются);
- в случае реализации взаимного исключения используется двоичный семафор, т.е. такой, что он может принимать только значения 0 и 1: такой семафор всегда разрешает доступ к ресурсу не более чем одному процессу одновременно

Рассмотрим набор вызовов для оперирования с семафорами в UNIX System V.

Доступ к семафору

Для получения доступа к массиву семафоров (или его создания) используется системный вызов:

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>

    INT SEMGET (KEY_T KEY, INT NSEMS, INT SEMFLAG);
```

Первый параметр функции **`semget()`** – ключ для доступа к разделяемому ресурсу, второй - количество семафоров в создаваемом наборе (длина массива семафоров) и третий параметр – флаги, управляющие поведением вызова. Подробнее процесс создания разделяемого ресурса описан выше. Отметим семантику прав доступа к такому типу разделяемых ресурсов, как семафоры: процесс, имеющий право доступа к массиву семафоров по чтению, может

проверять значение семафоров; процесс, имеющий право доступа по записи, может как проверять, так и изменять значения семафоров.
В случае, если среди флагов указан **IPC_CREAT**, аргумент **nsems** должен представлять собой положительное число, если же этот флаг не указан, значение **nsems** игнорируется. Отметим, что в заголовочном файле **<sys/sem.h>** определена константа **SEMMSL**, задающая максимально возможное число семафоров в наборе. Если значение аргумента **nsems** больше этого значения, вызов **semget()** завершится неудачно.
В случае успеха вызов **semget()** возвращает положительный дескриптор созданного разделяемого ресурса, в случае неудачи -1.

Операции над семафором

Используя полученный дескриптор, можно производить изменять значения одного или нескольких семафоров в наборе, а также проверять их значения на равенство нулю, для чего используется системный вызов **semop()**:

```
#include <sys/types.h>
#include<sys/ipc.h>
#include<sys/sem.h>
```

```
INT SEMOP (INT SEMID, STRUCT SEMBUF *SEMOP, SIZE_T
NOPS)
```

Этому вызову передаются следующие аргументы:
semid – дескриптор массива семафоров;
semop – массив из объектов типа **struct sembuf**, каждый из которых задает одну операцию над семафором;
nops – длина массива **semop**. Количество семафоров, над которыми процесс может одновременно производить операцию в одном вызове **semop()**, ограничено константой **SEMOPM**, описанной в файле **<sys/sem.h>**. Если процесс попытается вызвать **semop()** с параметром **nops**, большим этого значения, этот вызов вернет неуспех.
Структура имеет **sembuf** вид:

```
struct sembuf {
     short sem_num;      /* номер семафора в векторе */
     short sem_op;  /* производимая операция */
     short sem_flg;      /* флаги операции */
}
```

Поле операции в структуре интерпретируется следующим образом:

Пусть значение семафора с номером **sem_num** равно **sem_val**.

1. если значение операции не равно нулю:
   - оценивается значение суммы **sem_val + sem_op**.
   - если эта сумма больше либо равна нулю, то значение данного семафора устанавливается равным этой сумме: **sem_val = sem_val + sem_op**
   - если же эта сумма меньше нуля, то действие процесса будет приостановлено до тех пор, пока значение суммы **sem_val + sem_op** не станет больше либо равно нулю, после чего значение семафора устанавливается равным этой сумме: **sem_val = sem_val + sem_op**
2. Если код операции **sem_op** равен нулю:

- Если при этом значение семафора (**sem_val**) равно нулю, происходит немедленный возврат из вызова
- Иначе происходит блокирование процесса до тех пор, пока значение семафора не обнулится, после чего происходит возврат из вызова

Таким образом, ненулевое значение поля **sem_op** обозначает необходимость прибавить к текущему значению семафора значение **sem_op**, а нулевое – дождаться обнуления семафора.

Поле **sem_flg** в структуре **sembuf** содержит комбинацию флагов, влияющих на выполнение операции с семафором. В этом поле может быть установлен флаг **IPC_NOWAIT**, который предписывает соответствующей операции над семафором не блокировать процесс, а сразу возвращать управление из вызова **semop()**. Вызов **semop()** в такой ситуации вернет –1. Кроме того, в этом поле может быть установлен флаг **SEM_UNDO**, в этом случае система запомнит изменение значения семафора, произведенные данным вызовом, и по завершении процесса автоматически ликвидирует это изменение. Это предохраняет от ситуации, когда процесс уменьшил значение семафора, начав работать с ресурсом, а потом, не увеличив значение семафора обратно, по какой-либо причине завершился. В этом случае остальные процессы, ждущие доступа к ресурсу, оказались бы заблокированы навечно.

Управление массивом семафоров.

```
#include <sys/types.h>
#include<sys/ipc.h>
#include<sys/sem.h>
```

```
INT SEMCTL (INT SEMID, INT NUM, INT CMD, UNION SEMUN
ARG)
```

С помощью этого системного вызова можно запрашивать и изменять управляющие параметры разделяемого ресурса, а также удалять его.

Первый параметр вызова – дескриптор массива семафоров. Параметр **num** представляет собой индекс семафора в массиве, параметр **cmd** задает операцию, которая должна быть выполнена над данным семафором. Последний аргумент имеет тип **union semun** и используется для считывания или задания управляющих параметров одного семафора или всего массива, в зависимости от значения аргумента **cmd**. Тип данных **union semun** определен в файле **<sys/sem.h>** и выглядит следующим образом:

```
union semun {
    int val; // значение одного семафора
    struct semid_ds *buf;  /* параметры массива семафоров в
целом */
    ushort   *array;       /* массив значений семафоров */
}
```

где **struct semid_ds** – структура, описанная в том же файле, в полях которой хранится информация о всем наборе семафоров в целом, а именно, количество семафоров в наборе, права доступа к нему и статистика доступа к массиву семафоров.

Приведем некоторые наиболее часто используемые значения аргумента **cmd**:

IPC_STAT – скопировать управляющие параметры набора семафоров по адресу **arg.buf**
IPC_SET – заменить управляющие параметры набора семафоров на те, которые указаны в **arg.buf**. Чтобы выполнить эту операцию, процесс должен быть владельцем или создателем массива семафоров, либо обладать правами привилегированного пользователя, при этом процесс может изменить только владельца массива семафоров и права доступа к нему.
IPC_RMID – удалить массив семафоров. Чтобы выполнить эту операцию, процесс должен быть владельцем или создателем массива семафоров, либо обладать правами привилегированного пользователя
GETALL, SETALL – считать / установить значения всех семафоров в массив, на который указывает **arg.array**
GETVAL – возвратить значение семафора с номером **num**. Последний аргумент вызова игнорируется.
SETVAL – установить значение семафора с номером num равным **arg.val**
В случае успешного завершения вызов возвращает значение, соответствующее конкретной выполнявшейся операции (0, если не оговорено иное), в случае неудачи – -1.

14. Работа с разделяемой памятью с синхронизацией семафорами.

Программа будет оперировать с разделяемой памятью.
1 процесс – создает ресурсы “разделяемая память” и “семафоры”, далее он начинает принимать строки со стандартного ввода и записывает их в разделяемую память.
2 процесс – читает строки из разделяемой памяти.
Таким образом мы имеем критический участок в момент, когда один процесс еще не дописал строку, а другой ее уже читает. Поэтому следует установить некоторые синхронизации и задержки.
1й процесс:

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
#include <string.h>
#define NMAX  256
int main(int argc, char **argv)
{
     key_t key;
     int semid, shmid;
     struct sembuf sops;
     char *shmaddr;
     char str[NMAX];

     key = ftok(“/usr/ter/exmpl”, ’S’);
     /* создаем уникальный ключ */
     semid = semget(key, 1, 0666 | IPC_CREAT);
          /* создаем один семафор с определенными правами
          доступа */
```

```
    shmid = shmget(key, NMAX, 0666 | IPC_CREAT);
        /* создаем разделяемую память на 256 элементов */
    shmaddr = shmat(shmid, NULL, 0);
        /* подключаемся к разделу памяти, в shaddr –
        указатель на буфер с разделяемой памятью */
     semctl(semid,0,SETVAL, (int) 0);
    /* инициализируем семафор значением 0 */
    sops.sem_num = 0;
    sops.sem_flg = 0;
    do { /* запуск цикла */
        printf("Введите строку:");
            if (fgets(str, NMAX, stdin) == NULL)
            {
            /* окончание ввода */
                /* пишем признак завершения – строку "Q"
                */
            strcpy(str, "Q");
        }
            /* в текущий момент семафор открыт для этого
            процесса */
            strcpy(shmaddr, str); /* копируем строку в
            разд. память */
            /* предоставляем второму процессу возможность
            войти */
            sops.sem_op = 3; /* увеличение семафора на 3
            */
        semop(semid, &sops, 1);
            /* ждем, пока семафор будет открыт для 1го
            процесса -  для следующей итерации цикла */
            sops.sem_op = 0; /* ожидание обнуления
            семафора */
        semop(semid, &sops, 1);
    } while (str[0] != 'Q');
        /* в данный момент второй процесс уже дочитал из
        разделяемой памяти и отключился от нее – можно ее
        удалять*/
        shmdt(shmaddr) ; /* отключаемся от разделяемой
        памяти */
    shmctl(shmid, IPC_RMID, NULL);
    /* уничтожаем разделяемую память */
    semctl(semid, 0, IPC_RMID, (int) 0);
    /* уничтожаем семафор */
    return 0;
}
```

2й процесс:

```
    /* необходимо корректно определить существование
    ресурса, если он есть - подключиться */
```

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
#include <string.h>
#define NMAX  256

int main(int argc, char **argv)
{
     key_t key;
     int semid, shmid;
     struct sembuf sops;
     char *shmaddr;
     char str[NMAX];

     key = ftok("/usr/ter/exmpl",'S');
     /* создаем тот же самый ключ */
     semid = semget(key, 1, 0666 | IPC_CREAT);
     shmid = shmget(key, NMAX, 0666 | IPC_CREAT);
          /* аналогично предыдущему процессу  -
          инициализации ресурсов */
     shmaddr = shmat(shmid, NULL, 0);
     sops.sem_num = 0;
     sops.sem_flg = 0;
     /* запускаем цикл */
     do {
               printf("Waiting… \n"); /* ожидание на семафоре
               */
          sops.sem_op = -2;
               /* будем ожидать, пока "значение семафора" +
               "значение sem_op" не станет положительным,
               т.е. пока значение семафора не станет как
               минимум 3 (3-2=1 > 0) */
          semop(semid, &sops, 1);
          /* теперь значение семафора равно 1 */
               strcpy(str, shmaddr); /* копируем строку из
               разд.памяти */
               /*критическая секция - работа с разделяемой
               памятью - в этот момент  первый процесс к
               разделяемой памяти доступа не имеет*/
               if (str[0] == 'Q')
               {
                     /*завершение работы - освобождаем
                     разделяемую память */
                     shmdt(shmaddr);
               }
          /*после работы – обнулим семафор*/
          sops.sem_op=-1;
          semop(semid, &sops, 1);
```

```
            printf("Read from shared memory: %s\n", str);
    } while (str[0] != 'Q');
    return 0;
}
```

Отметим, что данный пример демонстрирует два разных приема использования семафоров для синхронизации: первый процесс блокируется в ожидании обнуления семафора, т.е. для того, чтобы он мог войти в критическую секцию, значение семафора должно стать нулевым; второй процесс блокируется при попытке уменьшить значение семафора до отрицательной величины, для того, чтобы этот процесс мог войти в критическую секцию, значение семафора должно быть не менее 3. Обратите внимание, что в данном примере, помимо взаимного исключения процессов, достигается строгая последовательность действий двух процессов: они получают доступ к критической секции строго по очереди.

БИЛЕТ 43

Механизм сокетов.

Средства межпроцессного взаимодействия ОС UNIX, представленные в системе IPC, решают проблему взаимодействия двух процессов, выполняющихся в рамках одной операционной системы. Однако, очевидно, их невозможно использовать, когда требуется организовать взаимодействие процессов в рамках сети. Это связано как с принятой системой именования, которая обеспечивает уникальность только в рамках данной системы, так и вообще с реализацией механизмов разделяемой памяти, очереди сообщений и семафоров, – очевидно, что для удаленного взаимодействия они не годятся. Следовательно, возникает необходимость в каком-то дополнительном механизме, позволяющем общаться

двум процессам в рамках сети. Однако если разработчики программ будут иметь два абсолютно разных подхода к реализации взаимодействия процессов, в зависимости от того, на одной машине они выполняются или на разных узлах сети, им, очевидно, придется во многих случаях создавать два принципиально разных куска кода, отвечающих за это взаимодействие. Понятно, что это неудобно и хотелось бы в связи с этим иметь некоторый унифицированный механизм, который в определенной степени позволял бы абстрагироваться от расположения процессов и давал бы возможность использования одних и тех же подходов для локального и нелокального взаимодействия. Кроме того, как только мы обращаемся к сетевому взаимодействию, встает проблема многообразия сетевых протоколов и их использования. Понятно, что было бы удобно иметь какой-нибудь общий интерфейс, позволяющий пользоваться услугами различных протоколов по выбору пользователя.

Обозначенные проблемы был призван решить механизм, впервые появившийся в UNIX – BSD (4.2) и названный сокетами (**sockets**).

Сокеты представляют собой в определенном смысле обобщение механизма каналов, но с учетом возможных особенностей, возникающих при работе в сети. Кроме того, они предоставляют больше возможностей по передаче сообщений, например, могут поддерживать передачу экстренных сообщений вне общего потока данных. Общая схема работы с сокетами любого типа такова: каждый из взаимодействующих процессов должен на своей стороне создать и отконфигурировать сокет, после чего процессы должны осуществить соединение с использованием этой пары сокетов. По окончании взаимодействия сокеты уничтожаются.

Механизм сокетов чрезвычайно удобен при разработке взаимодействующих приложений, образующих систему «клиент-сервер». Клиент посылает серверу запросы на предоставление услуги, а сервер отвечает на эти запросы.

Схема использования механизма сокетов для взаимодействия в рамках модели «клиент-сервер» такова. Процесс-сервер запрашивает у ОС сокет и, получив его, присваивает ему некоторое имя (адрес), которое предполагается заранее известным всем клиентам, которые захотят общаться с данным сервером. После этого сервер переходит в режим ожидания и обработки запросов от клиентов. Клиент, со своей стороны, тоже создает сокет и запрашивает соединение своего сокета с сокетом сервера, имеющим известное ему имя (адрес). После того, как соединение будет установлено, клиент и сервер могут обмениваться данными через соединенную пару сокетов. Ниже мы подробно рассмотрим функции, выполняющие все необходимые действия с сокетами, и напишем пример небольшой серверной и клиентской программы, использующих сокеты.

Типы сокетов. Коммуникационный домен.

Сокеты подразделяются на несколько типов в зависимости от типа коммуникационного соединения, который они используют. Два основных типа коммуникационных соединений и, соответственно, сокетов представляет собой соединение с использованием виртуального канала и датаграммное соединение.

Соединение с использованием виртуального канала – это последовательный поток байтов, гарантирующий надежную доставку сообщений с сохранением порядка их следования. Данные начинают передаваться только после того, как виртуальный канал установлен, и канал не разрывается, пока все данные не будут переданы. Примером соединения с установлением виртуального канала является механизм

каналов в UNIX, аналогом такого соединения из реальной жизни также является телефонный разговор. Заметим, что границы сообщений при таком виде соединений не сохраняются, т.е. приложение, получающее данные, должно само определять, где заканчивается одно сообщение и начинается следующее. Такой тип соединения может также поддерживать передачу экстренных сообщений вне основного потока данных, если это возможно при использовании конкретного выбранного протокола.

Датаграммное соединение используется для передачи отдельных пакетов, содержащих порции данных – датаграмм. Для датаграмм не гарантируется доставка в том же порядке, в каком они были посланы. Вообще говоря, для них не гарантируется доставка вообще, надежность соединения в этом случае ниже, чем при установлении виртуального канала. Однако датаграммные соединения, как правило, более быстрые. Примером датаграммного соединения из реальной жизни может служить обычная почта: письма и посылки могут приходить адресату не в том порядке, в каком они были посланы, а некоторые из них могут и совсем пропадать.

Поскольку сокеты могут использоваться как для локального, так и для удаленного взаимодействия, встает вопрос о пространстве адресов сокетов. При создании сокета указывается так называемый *коммуникационный домен*, к которому данный сокет будет принадлежать. Коммуникационный домен определяет форматы адресов и правила их интерпретации. Мы будем рассматривать два основных домена: для локального взаимодействия – домен **AF_UNIX** и для взаимодействия в рамках сети – домен **AF_INET** (префикс AF обозначает сокращение от «address family» – семейство адресов). В домене **AF_UNIX** формат адреса – это допустимое имя файла, в домене **AF_INET** адрес образуют имя хоста + номер порта.

Заметим, что фактически коммуникационный домен определяет также используемые семейства протоколов. Так, для домена **AF_UNIX** это будут внутренние протоколы ОС, для домена **AF_INET** – протоколы семейства TCP/IP. Современные системы поддерживают и другие коммуникационные домены, например BSD UNIX поддерживает также третий домен – **AF_NS**, использующий протоколы удаленного взаимодействия Xerox NS.

Ниже приведен набор функций для работы с сокетами.

Создание и конфигурирование сокета.

Создание сокета.

```
#include <sys/types.h>
#include <sys/socket.h>
int socket (int domain, int type, int protocol)
```

Функция создания сокета так и называется – **socket().** У нее имеется три аргумента. Первый аргумент – **domain** – обозначает коммуникационный домен, к которому должен принадлежать создаваемый сокет. Для двух рассмотренных нами доменов соответствующие константы будут равны, как мы уже говорили, **AF_UNIX** и **AF_INET**. Второй аргумент – **type** – определяет тип соединения, которым будет пользоваться сокет (и, соответственно, тип сокета). Для двух основных рассматриваемых нами типов сокетов это будут константы **SOCK_STREAM** для соединения с установлением виртуального канала и

SOCK_DGRAM для датаграмм[5]. Третий аргумент – **protocol** – задает конкретный протокол, который будет использоваться в рамках данного коммуникационного домена для создания соединения. Если установить значение данного аргумента в 0, система автоматически выберет подходящий протокол. В наших примерах мы так и будем поступать. Однако здесь для справки приведем константы для протоколов, используемых в домене **AF_INET:**

IPPROTO_TCP – обозначает протокол TCP (корректно при создании сокета типа **SOCK_STREAM**)

IPPROTO_UDP – обозначает протокол UDP (корректно при создании сокета типа **SOCK_DGRAM**)

Функция **socket()** возвращает в случае успеха положительное целое число – дескриптор сокета, которое может быть использовано в дальнейших вызовах при работе с данным сокетом. Заметим, что дескриптор сокета фактически представляет собой файловый дескриптор, а именно, он является индексом в таблице файловых дескрипторов процесса, и может использоваться в дальнейшем для операций чтения и записи в сокет, которые осуществляются подобно операциям чтения и записи в файл (подробно эти операции будут рассмотрены ниже).

В случае если создание сокета с указанными параметрами невозможно (например, при некорректном сочетании коммуникационного домена, типа сокета и протокола), функция возвращает –1.

Связывание.

Для того чтобы к созданному сокету мог обратиться какой-либо процесс извне, необходимо присвоить ему адрес. Как мы уже говорили, формат адреса зависит от коммуникационного домена, в рамках которого действует сокет, и может представлять собой либо путь к файлу, либо сочетание IP-адреса и номера порта. Но в любом случае связывание сокета с конкретным адресом осуществляется одной и той же функцией `bind`:

```
#include <sys/types.h>
#include <sys/socket.h>
     int bind (int sockfd, struct sockaddr *myaddr, int
     addrlen)
```

Первый аргумент функции – дескриптор сокета, возвращенный функцией **socket()**; второй аргумент – указатель на структуру, содержащую адрес сокета. Для домена **AF_UNIX** формат структуры описан в **<sys/un.h>** и выглядит следующим образом:

```
#include <sys/un.h>
struct sockaddr_un {
     short sun_family; /* == AF_UNIX */
     char sun_path[108];
};
```

---

[5] Заметим, что данный аргумент может принимать не только указанные два значения, например, тип сокета **SOCK_SEQPACKET** обозначает соединение с установлением виртуального канала со всеми вытекающими отсюда свойствами, но при этом сохраняются границы сообщений; однако данный тип сокетов не поддерживается ни в домене **AF_UNIX**, ни в домене **AF_INET**, поэтому мы его здесь рассматривать не будем

Для домена **AF_INET** формат структуры описан в **<netinet/in.h>** и выглядит следующим образом:

```
#include <netinet/in.h>
struct sockaddr_in {
     short sin_family; /* == AF_INET */
     u_short sin_port;  /* port number */
     struct in_addr sin_addr;  /* host IP address */
     char sin_zero[8];  /* not used */
};
```

Последний аргумент функции задает реальный размер структуры, на которую указывает **myaddr.**

Важно отметить, что если мы имеем дело с доменом **AF_UNIX** и адрес сокета представляет собой имя файла, то при выполнении функции **bind()** система в качестве побочного эффекта создает файл с таким именем. Поэтому для успешного выполнения **bind()** необходимо, чтобы такого файла не существовало к данному моменту. Это следует учитывать, если мы «зашиваем» в программу определенное имя и намерены запускать нашу программу несколько раз на одной и той же машине – в этом случае для успешной работы **bind()** необходимо удалять файл с этим именем перед связыванием. Кроме того, в процессе создания файла, естественно, проверяются права доступа пользователя, от имени которого производится вызов, ко всем директориям, фигурирующим в полном путевом имени файла, что тоже необходимо учитывать при задании имени. Если права доступа к одной из директорий недостаточны, вызов **bind()** завершится неуспешно.

В случае успешного связывания **bind()** возвращает 0, в случае ошибки – -1**.**

Предварительное установление соединения.

Сокеты с установлением соединения. Запрос на соединение.

Различают **сокеты с предварительным установлением соединения**, когда до начала передачи данных устанавливаются адреса сокетов отправителя и получателя данных – такие сокеты соединяются друг с другом и остаются соединенными до окончания обмена данными; и **сокеты без установления соединения**, когда соединение до начала передачи данных не устанавливается, а адреса сокетов отправителя и получателя передаются с каждым сообщением. Если тип сокета – виртуальный канал, то сокет ***должен*** устанавливать соединение, если же тип сокета – датаграмма, то, как правило, это сокет без установления соединения, хотя последнее не является требованием. Для установления соединения служит следующая функция:

```
#include <sys/types.h>
#include <sys/socket.h>
     int connect (int sockfd, struct sockaddr *serv_addr,
     int addrlen);
```

Здесь первый аргумент – дескриптор сокета, второй аргумент – указатель на структуру, содержащую адрес сокета, с которым производится соединение, в формате, который мы обсуждали выше, и третий аргумент содержит реальную длину этой структуры. Функция возвращает 0 в случае успеха и –1 в случае неудачи, при этом код ошибки можно посмотреть в переменной **errno.**

Заметим, что в рамках модели «клиент-сервер» клиенту, вообще говоря, не важно, какой адрес будет назначен его сокету, так как никакой процесс не будет пытаться непосредственно установить соединение с сокетом клиента. Поэтому клиент может не вызывать предварительно функцию **bind()**, в этом случае при вызове **connect()** система автоматически выберет приемлемые значения для локального адреса клиента. Однако сказанное справедливо только для взаимодействия в рамках домена **AF_INET**, в домене **AF_UNIX** клиентское приложение само должно позаботиться о связывании сокета.

Сервер: прослушивание сокета и подтверждение соединения.

Следующие два вызова используются сервером только в том случае, если используются сокеты с предварительным установлением соединения.

```
#include <sys/types.h>
#include <sys/socket.h>
int listen (int sockfd, int backlog);
```

Этот вызов используется процессом-сервером для того, чтобы сообщить системе о том, что он готов к обработке запросов на соединение, поступающих на данный сокет. До тех пор, пока процесс – владелец сокета не вызовет **listen(),** все запросы на соединение с данным сокетом будут возвращать ошибку. Первый аргумент функции – дескриптор сокета. Второй аргумент, **backlog,** содержит максимальный размер очереди запросов на соединение. ОС буферизует приходящие запросы на соединение, выстраивая их в очередь до тех пор, пока процесс не сможет их обработать. В случае если очередь запросов на соединение переполняется, поведение ОС зависит от того, какой протокол используется для соединения. Если конкретный протокол соединения не поддерживает возможность перепосылки (retransmission) данных, то соответствующий вызов **connect()** вернет ошибку **ECONNREFUSED**. Если же перепосылка поддерживается (как, например, при использовании TCP), ОС просто выбрасывает пакет, содержащий запрос на соединение, как если бы она его не получала вовсе. При этом пакет будет присылаться повторно до тех пор, пока очередь запросов не уменьшится и попытка соединения не увенчается успехом, либо пока не произойдет тайм-аут, определенный для протокола. В последнем случае вызов **connect()** завершится с ошибкой **ETIMEDOUT**. Это позволит клиенту отличить, был ли процесс-сервер слишком занят, либо он не функционировал. В большинстве систем максимальный допустимый размер очереди равен 5.

Конкретное соединение устанавливается при помощи вызова **accept()**:

```
#include <sys/types.h>
#include <sys/socket.h>
     int accept (int sockfd, struct sockaddr *addr, int
     *addrlen);
```

Этот вызов применяется сервером для удовлетворения поступившего клиентского запроса на соединение с сокетом, который сервер к тому моменту уже прослушивает (т.е. предварительно была вызвана функция **listen()**). Вызов **accept()** извлекает первый запрос из очереди запросов, ожидающих соединения, и устанавливает с ним соединение. Если к моменту вызова **accept()** очередь запросов на соединение пуста, процесс, вызвавший **accept()**, блокируется до поступления запросов.

Когда запрос поступает и соединение устанавливается, **accept()** создает новый сокет, который будет использоваться для работы с данным соединением, и возвращает дескриптор этого нового сокета, соединенного с сокетом клиентского процесса. При этом первоначальный сокет продолжает оставаться в состоянии прослушивания. Через новый сокет осуществляется обмен данными, в то время как старый сокет продолжает обрабатывать другие поступающие запросы на соединение (напомним, что именно первоначально созданный сокет связан с адресом, известным клиентам, поэтому все клиенты могут слать запросы только на соединение с этим сокетом). Это позволяет процессу-серверу поддерживать несколько соединений одновременно. Обычно это реализуется путем порождения для каждого установленного соединения отдельного процесса-потомка, который занимается собственно обменом данными только с этим конкретным клиентом, в то время как процесс-родитель продолжает прослушивать первоначальный сокет и порождать новые соединения (см.).
Во втором параметре передается указатель на структуру, в которой возвращается адрес клиентского сокета, с которым установлено соединение, а в третьем параметре возвращается реальная длина этой структуры. Благодаря этому сервер всегда знает, куда ему в случае надобности следует послать ответное сообщение. Если адрес клиента нас не интересует, в качестве второго аргумента можно передать **NULL**.

Прием и передача данных.

Собственно для приема и передачи данных через сокет используются три пары функций.

```
#include <sys/types.h>
#include <sys/socket.h>
     int  send(int  sockfd,  const  void *msg, int len,
     unsigned int flags);
     int recv(int sockfd, void *buf, int len, unsigned int
     flags);
```

Эти функции используются для обмена только через сокет с предварительно установленным соединением. Аргументы функции **send(): sockfd** – дескриптор сокета, через который передаются данные, **msg** и **len** - сообщение и его длина. Если сообщение слишком длинное для того протокола, который используется при соединении, оно не передается и вызов возвращает ошибку **EMSGSIZE.** Если же сокет окажется переполнен, т.е. в его буфере не хватит места, чтобы поместить туда сообщение, выполнение процесса блокируется до появления возможности поместить сообщение. Функция **send()** возвращает количество переданных байт в случае успеха и -1 в случае неудачи. Код ошибки при этом устанавливается в **errno.** Аргументы функции **recv()** аналогичны: **sockfd** – дескриптор сокета, **buf** и **len** – указатель на буфер для приема данных и его первоначальная длина. В случае успеха функция возвращает количество считанных байт, в случае неудачи -1[6].
Последний аргумент обеих функций – **flags** – может содержать комбинацию специальных опций. Нас будут интересовать две из них:

---

[6] Отметим, что, как уже говорилось, при использовании сокетов с установлением виртуального соединения границы сообщений не сохраняются, поэтому приложение, принимающее сообщения, может принимать данные совсем не теми же порциями, какими они были посланы. Вся работа по интерпретации сообщений возлагается на приложение.

MSG_OOB – этот флаг сообщает ОС, что процесс хочет осуществить прием/передачу экстренных сообщений

MSG_PEEK – данный флаг может устанавливаться при вызове **recv()**. При этом процесс получает возможность прочитать порцию данных, не удаляя ее из сокета, таким образом, что последующий вызов **recv()** вновь вернет те же самые данные.

Другая пара функций, которые могут использоваться при работе с сокетами с предварительно установленным соединением – это обычные **read()** и **write()**, в качестве дескриптора которым передается дескриптор сокета.

И, наконец, пара функций, которая может быть использована как с сокетами с установлением соединения, так и с сокетами без установления соединения:

```
#include <sys/types.h>
#include <sys/socket.h>
     int sendto(int sockfd, const void *msg, int len,
     unsigned  int flags, const struct sockaddr *to, int
     tolen);
     int recvfrom(int sockfd, void *buf, int len, unsigned
     int flags, struct sockaddr *from, int *fromlen);
```

Первые 4 аргумента у них такие же, как и у рассмотренных выше. В последних двух в функцию **sendto()** должны быть переданы указатель на структуру, содержащую адрес получателя, и ее размер, а функция **recvfrom()** в них возвращает соответственно указатель на структуру с адресом отправителя и ее реальный размер. Отметим, что перед вызовом **recvfrom()** параметр **fromlen** должен быть установлен равным первоначальному размеру структуры **from.** Здесь, как и в функции **accept**, если нас не интересует адрес отправителя, в качестве **from** можно передать **NULL.**

Завершение работы с сокетом.

Если процесс закончил прием либо передачу данных, ему следует закрыть соединение. Это можно сделать с помощью функции **shutdown():**

```
# include <sys/types.h>
# include <sys/socket.h>
int shutdown (int sockfd, int mode);
```

Помимо дескриптора сокета, ей передается целое число, которое определяет режим закрытия соединения. Если **mode=0,** то сокет закрывается для чтения, при этом все дальнейшие попытки чтения будут возвращать **EOF**. Если **mode=1**, то сокет закрывается для записи, и при осуществлении в дальнейшем попытки передать данные будет выдан кода неудачного завершения (-1). Если **mode=2**, то сокет закрывается и для чтения, и для записи.

Аналогично файловому дескриптору, дескриптор сокета освобождается системным вызовом **close()**. При этом, разумеется, даже если до этого не был вызван **shutdown()**, соединение будет закрыто. Таким образом, в принципе, если по окончании работы с сокетом мы собираемся закрыть соединение и по чтению, и по записи, можно было бы сразу вызвать **close()** для дескриптора данного сокета, опустив вызов **shutdown()**. Однако, есть небольшое различие с тем случаем, когда предварительно был вызван **shutdown()**. Если используемый для соединения протокол гарантирует доставку данных (т.е. тип сокета – виртуальный канал), то вызов **close()** будет блокирован до тех пор, пока система будет

пытаться доставить все данные, находящиеся «в пути» (если таковые имеются), в то время как вызов **`shutdown()`** извещает систему о том, что эти данные уже не нужны и можно не предпринимать попыток их доставить, и соединение закрывается немедленно. Таким образом, вызов **`shutdown()`** важен в первую очередь для закрытия соединения сокета с использованием виртуального канала.

Резюме: общая схема работы с сокетами.

Мы рассмотрели все основные функции работы с сокетами. Обобщая изложенное, можно изобразить общую схему работы с сокетами с установлением соединения в следующем виде:

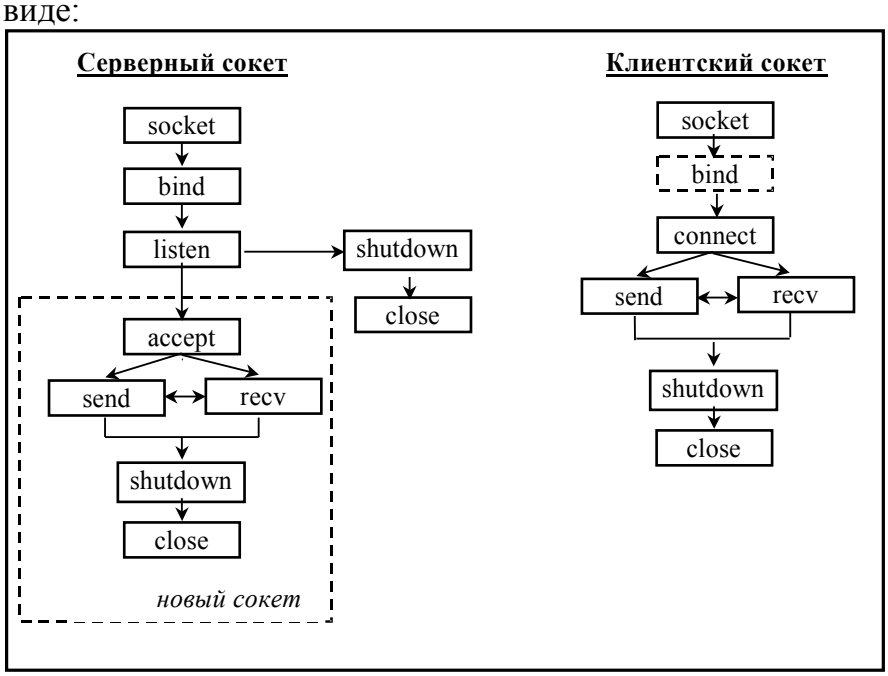


Рис. 12 Схема работы с сокетами с установлением соединения

БИЛЕТ 44

Общая схема работы с сокетами без предварительного установления соединения проще, она такова:

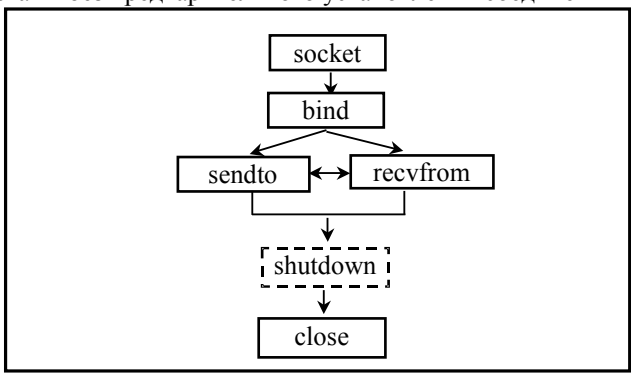


Рис. 13 Схема работы с сокетами без установления соединения

Билет 46. Основные правила работы с файлами. Типовые программные интерфейсы работы с файлами.

Файловая система (ФС) - часть операционной системы, представляющая собой совокупность организованных наборов данных, хранящихся на внешних запоминающих устройствах, и программных средств, гарантирующих именованный доступ к этим данным и их защиту

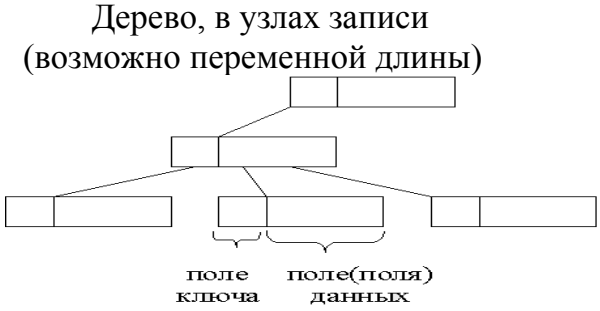
Возможности предоставляемые ФС определяют эксплутационные качества ФС. От оптимальности организации зависит область применимости ФС.
ФС – компонент ОС обеспечивающий именованный доступ к данным. Данные называются ***файлами***, их имена - ***именами файлов***.
Ранее работа с данными осуществлялась через координаты их на внешних носителях. Это было неудобно, т.к. надо было помнить о местонахождении данных, перемещение программы с одного носителя на другой тоже вызывало трудности. Если возникала потребность менять входные данные в программе, то это тоже было не легко.
ФС совершила революцию. Появилась возможность ассоциировать с совокупностью данных некоторое имя и осуществлять доступ к данным через указатель имени.
ОС брала на себя функции размещения данных, ассоциированных с именем, сохранение информации, соответствующей данному имени.

Структурная организация файлов

Существует множество разновидностей структурной организации файлов. Наиболее популярные:
1. Файл, как последовательность байтов (обмен от 1 до фиксированного числа байтов) Т.е. файл – это набор данных, практически не имеющих никакой структуры. Соответственно вопрос выделения логической структуры – это уже проблема пользователя. Пользователь записывает данные, как последовательность байтов, считывает их и сам уже интерпретирует. Как ни странно, на сегодняшний день – это одна из самых распространенных моделей структурной организации файлов. Таким образом организуются файловые системы Unix, Windows, т.е. файл там может быть представлен как просто последовательность байтов
2. Файл, как последовательность записей переменной длины (обмен в терминах записи, информация в виде последовательности записей, поле данных + символ конца записи, последовательный доступ) В этом случае каждая запись, кроме содержательной информации, должна была иметь некоторую специальную информацию. эта специальная информация могла быть либо полем, которому указывалась длина записи, либо специальная информация могла представляться в виде специального кода - маркера конца или начала записи. При такой организации внутренней фрагментации практически не было, за исключением тех потерь, которые приходились на разметку файла по записи, т.е. либо указатели длины, либо маркеры начала и конца. В этом плане эффективность организации хранения была относительно хорошей. С другой стороны такая организация исключала прямой доступ к записи. Т.е. для того, чтобы добраться до i-ой записи нужно было промотать все предыдущие: либо пересчитать маркеры начала и конца, либо пробежаться по списку через указатели длины. Файлы такой организации имели сложность с точки зрения редактирования, т.е. изменение длины существующей записи с большой вероятностью приводило к проблеме. Поскольку увеличение

записи – это вообще затруднительная операция, а уменьшение – тоже есть некоторая проблема. Т.е. есть какая-то внутренняя проблема, которая приводила к неэффективности редактирования такого рода файлов. Записи постоянной длины организованы были так, что в пределах размера записи никаких проблем не возникало. Проблемы возникали только в том случае, если происходило либо удаление записи, либо вставка новой записи.

3. Файл, как последовательность записей постоянной длины (обмен в терминах записей постоянной длины) Исторически этот вариант структурной организации появился из-за использования такого носителя информации, как перфокарты. Т.е. было удобно делать файл, который был прямым аналогом колоды перфокарт. Соответственно это означает, что читать из файла или писать данные в этот файл система позволяла порциями размером в 80 байт. Понятно, что такая организации файла достаточно эффективна по скорости доступа, т.е. был прямой доступ к любой записи, потому что координаты записи внутри вычислялись всегда очень просто: (номер записи)*(размер записи). С другой стороны – внутренняя фрагментация. Один байт используется в записи и вся запись размером в 80 байтов становится занятой.
4. Иерархическая организация файла (дерево) (поиск, сортировка и т.д. осуществляется по ключам). Суть: структура файла представима в виде дерева. В каждом узле этого дерева находится информация о записи. Информация о записи – это два содержательных поля: поле ключа и поле данных. Соответственно дерево организовано таким образом, что в нем оптимизирован доступ к записям по указанию ключа, т.е. записи отсортированы по одинаковым ключам, и разные ключи отсортированы по возрастанию ключей. Поле данных может быть произвольного размера. Место расположения записи может быть в общем случае произвольно, т.е. ФС может разместить запись, где захочет, по своим каким-то критериям. имеются накладные расходы, связанные с древовидной организацией - с организацией ключей. Обычно, это достаточно специализированные ФС, которые используются или могут использоваться в высокопроизводительных, либо специальных ВС.



.

Атрибуты файла

- имя
- права доступа

- персонификация (создатель, владелец)
- тип файла
- размер записи
- размер файла
- указатель чтения / записи
- время создания
- время последней модификации
- время последнего обращения
- предельный размер файла
- .....

Полный состав атрибутов файла и способ их представления определяется конкретной файловой системой.

Основные правила работы с файлами

Операционная система и файловая система обеспечивают регистрацию возможности того или иного процесса работать с содержимым файлов. «Сеанс работы» с содержимым файла:

Начало «открытие» файла (регистрация в системе возможности работы процесса с содержимым файла)

Открытие – создание внутрисистемной структуры данных, кот. описывает состояние этого файла, проверяет права доступа, объявляет операционной системе тот факт, что с данным файлом будет работать тот или иной процесс. При открытии файла система формирует внутренние наборы данных, необходимые для работы с содержимым файла.

Работа с содержимым файла, с атрибутами файла

Завершение «закрытие» файла – информация системе о завершении работы процесса с «открытым» файлом

Закрытие файла. Закрытие файла - информация операционной системе о том, что работа с файлом завершена.

Операция закрытия файла имеет 2 вида:

закрыть и сохранить текущее содержимое файла;

уничтожить файл.

Типовые программные интерфейсы работы с файлами

open – открытие / создание файла

«r» - на чтение

«w» - на запись

… и т.д.

close – закрытие

read / write – читать, писать (относительно положения указателя чтения / запись, read/write по дескриптору а не по имени)

delete – удалить файл из файловой системы (напрямую или дескриптор)

seek – позиционирование указателя чтение/запись

rename – переименование файла

read / write _attributes – чтение, модификация
атрибутов файла.

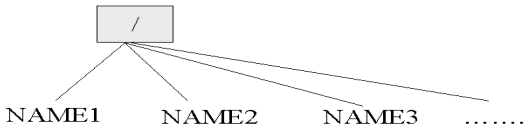
Файловый дескриптор – системная структура данных, содержащая информацию о актуальном состоянии «открытого» файла.
Файловый дескриптор содержит актуальную информацию о открытом файле. Через ФД можно получить информацию о значении указателей чтения\записи.
В некоторых ФС каталог – отдельное внутреннее образование, в UNIX каталог – файл специального типа. Если это файл, то для него можно использовать программные интерфейсы для работы с файлами.
Каталог – компонент файловой системы, содержащий информацию о содержащихся в файловой системе файлах. Специальные файлы – каталоги.

Билет 47 .Файловые системы. Модели реализации файловых систем. Понятие индексного дескриптора.

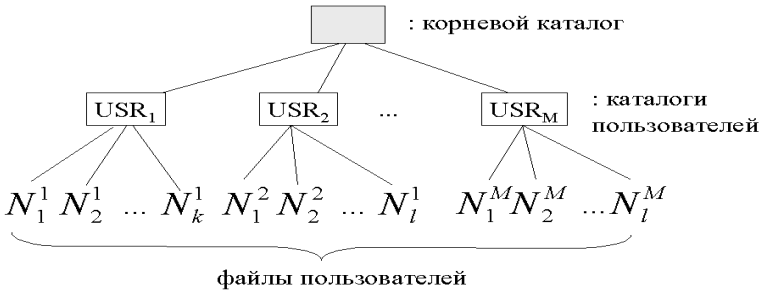
Файловый дескриптор – системная структура данных, содержащая информацию о актуальном состоянии «открытого» файла.

Файловый дескриптор содержит актуальную информацию о открытом файле. Через ФД можно получить информацию о значении указателей чтения\записи.
В некоторых ФС каталог – отдельное внутреннее образование, в UNIX каталог – файл специального типа. Если это файл, то для него можно использовать программные интерфейсы для работы с файлами.
Каталог – компонент файловой системы, содержащий информацию о содержащихся в файловой системе файлах.Специальные файлы – каталоги.

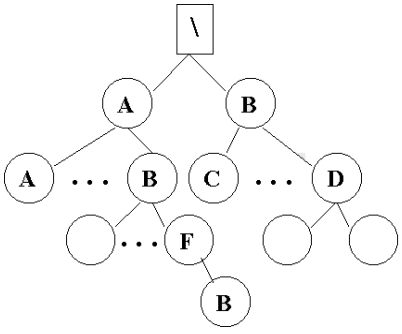
Модель одноуровневой файловой системы. Традиционно-простая организация каталога – одноуровневая модель ФС В ФС существует один каталог, в котором находятся все файлы находящиеся в системе.
Проблемы:
1.Коллизия имен. Каждое имя должно быть единственно.
2. нагрузка на работу с системой, если много файлов.
3. неудобно структурировать.



Модель **двухуровневой файловой системы.**
Модель, которая появилась в реальных системах на начальных этапах после одноуровневой.
В системе существует объединение каталогов пользователей, для каждого пользователя реализована одноуровневая модель.
Проблемы 1 и 2 исчезают, а 3 остается.



Иерархические файловые системы

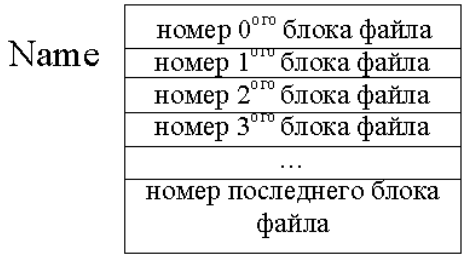


За основу логической организации такой файловой системы берется дерево. В корне дерева находится, так называемый, корень файловой системы - каталог нулевого уровня. В этом каталоге могут находиться либо файлы пользователей, либо каталоги первого уровня. Каталоги первого и следующих уровней организуются по аналогичному принципу. Файлы пользователя в этом дереве представляются листьями. Пустой каталог также может быть листом. Таким образом образуется древовидная структура файловой системы, где в узлах находятся каталоги, а листьями являются либо файлы, либо пустые каталоги.
Остановимся на правилах именования в иерархической файловой системе. В данном случае используется механизм, основанный на понятии имени файла (name) и полного имени файла (path_name). Полное имя файла – это путь от

корневого каталога до листа (такой путь всегда будет уникальным). Существует также относительное именование, т.е. когда нет необходимости указания полного пути при работе с файлами. Это происходит в случае, когда программа вызывает файл и подразумевается, что он находится в том же каталоге, что и программа. В данном случае появляется понятие текущего каталога, т. е. каталога, на работу с которым настроена файловая система в данный момент времени. В рамках одного каталога имена файлов одного уровня должны быть разными.

Индексные узлы (дескрипторы)

ФС организует кластеризованное, компактное хранение информации о размещении блоков файлов в специальной структуре, которая называется индексные узлы или индексные дескрипторы. В этой структуре находится информация о размещении блоков файлов по ФС. Т.е. соответственно есть таблица, в которой размещаются индексные узлы файла. При открытии файла осуществляется поиск по этой таблице соответствующего индексного узла, и после этого в памяти аккумулируется только индексный узел открытого файла, а не вся таблица.



Достоинства:

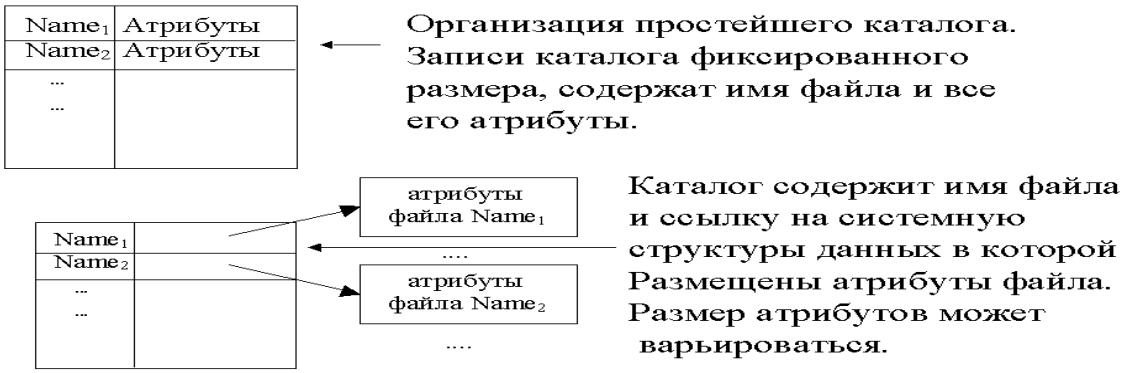
нет необходимости в размещении в ОЗУ информации всей FAT о все файлах системы, в памяти размещаются атрибуты, связанные только с открытыми файлами.

Недостатки:

размер файла и размер индексного узла (в общем случае прийти к размерам таблицы размещения). Решение:

- ограничение размера файла
- иерархическая организация индексных узлов

Модели организации каталогов



Каталог, кот. Представлен в виде таблицы:

1.Первая модель – простейший каталог. Каталог представляется в виде таблицы, которая содержит имя файла и его атрибуты. Соответственно каждая запись фиксированного размера. Проблема в том, что в этом случае каталог получается достаточно большой по объему, т.к. последовательность атрибутов может быть достаточно большой, в частности, в атрибуты может включаться и информация о размещении файла, о проблемах возникающих в таких случаях только что уже говорилось

«-» каталог очень большой по объему.

2. Каталог содержит имя файла – поле фиксированного размера, и ссылку уже на системные структуры данных или системную структуру данных, в которых находятся атрибуты соответствующих файлов. В этом случае получается более менее гибкую организацию по части размера атрибутов (они здесь могут быть достаточно произвольной длины.

«+» более гибкая организация по размеру атрибутов.

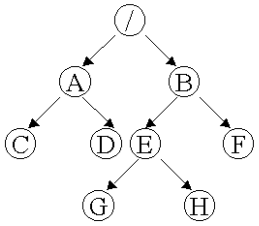
Оба вида основаны на фиксированном размере записи каталога, есть ограничения на длину имени. Проблема длинных имен: короткие неудобны – размещаем суффиксы имени в атрибутах.

Варианты соответствия: имя файла – содержимое файла

Содержимому любого файла соответствует единственное имя файла.

1. Исторически было взаимно однозначное соответствие между именем и содержимым файла. Т.е. для каждого содержимого файла существовало единственное имя файла и для каждого зарегистрированного файла в ФС существовало единственное содержимое.

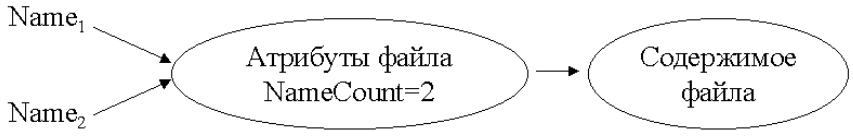
Примеры:



Содержимому файла может соответствовать два и более имен файла.

2. . В этом случае имя файла выносится на некоторый отдельный уровень из атрибутов, и получается, что есть содержимое файла, есть атрибут файла и может быть произвольное количество имен. Более того, эти имена могут размещаться в различных каталогах, если есть иерархическая ФС. Т.е. тем самым нарушается древообразность ФС

“Жесткая” связь

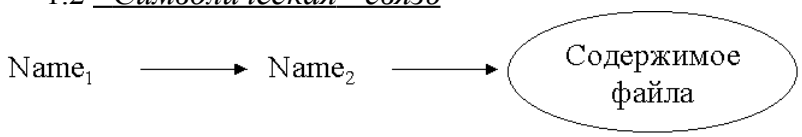


Для одного и того же содержимого файла может существовать>= 2 имен файла. Имя файла выносим из атрибутов. Есть содержимое файла, атрибуты и любое количество имен.

“Жесткая” связь Есть содержимое файла, есть атрибут файла, и одним из полей атрибутов является количество имен у этого файла, и есть произвольное количество имен, которые как-то распределены по каталогам ФС. В этом случае каждое из этих имен равнозначно, т.е. имеет место некоторая симметричная организация: каждое из имен синонимов равноправно, т.е. нет никакого старшинства в не зависимости от порядка образования этих файлов.

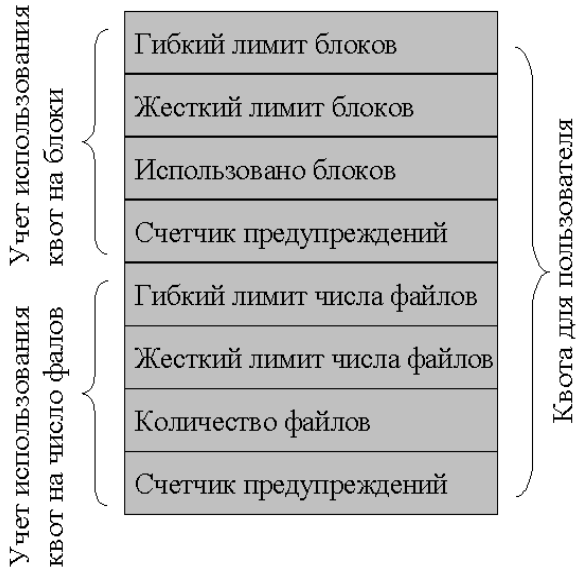
“Символическая” связь есть файл с именем $Name_2$, этому имени соответствуют атрибуты и соответствует содержимое, и есть специальный файл $Name_1$, который ссылается на имя $Name_2$. В этом случае имеет место ассиметричное именование файлов. Имя $Name_2$ позволяет организовывать более широкую работу с файлом, можно удалять, если файл будет удален, соответственно содержимое пропадет. Имя $Name_1$ имеет свои правила интепретации, поскольку они идут через ссылку на имя $Name_2$,т.е. можно осуществлять доступ к содержимому, но если удалить файл $Name_1$, то содержимое $Name_2$ останется неизменным.

1.2 *“Символическая” связь*



Билет 48 *Файловые системы.* Координация использования пространства внешней памяти. Квотирование пространства ФС. Надежность ФС. Проверка целостности ФС.

Квотирование пространства файловой системы



Одним из ресурсов ВС является **лимитированное пространство ФС**, выделенное для пользователя. Количество имен файла ограничено сверху размером раздела. Пространство, занимаемое файлами лимитировано. Для каждого из этих ресурсов определяется 2 квоты : на блоки и на файлы.

Существуют жесткие и гибкие лимиты . С **гибким лимитом блоков** связывается два параметра – это непосредственно значение, которое лимитирует использование ресурсов – это первый параметр. Второй параметр – это так называемый счетчик предупреждений. Работа осуществляется следующим образом. Пользователь заходит в свой номер, система определяет статус использования соответствующего лимита. Если реально используемый лимит, предположим это лимит на блоки файлов, превосходит гибкий лимит блоков, то пользователю при входе в систему дается предупреждение, что он израсходовал свой лимит, и счетчик предупреждения уменьшается на 1. Пользователю дается возможность заходить к себе в номер и работать до тех пор пока счетчик предупреждений не станет равен 0. Так только он станет равен 0 работа пользователя будет блокирована.

Жесткое лимитирование – этот тот лимит, который перейти нельзя никогда. Если при входе пользователя в систему обнаружится, что он превзошел жесткий лимит, то его работа блокируется сразу все зависимости от других ситуаций. Гибкий и жесткий лимит – это есть средства, в принципе, возможно использование различных

Если пользователь попытался превзойти жесткий лимит, то он блокируется сразу.

Надежность файловой системы

Критерий надежность файловой системы достаточно просто формулируется. Т.е. нужно обеспечить работу таким образом, чтобы при возникновении внештатной ситуации объем потерянной информации был минимальной. Внештатная ситуация может быть самой разнообразной – сбой в любом из узлов компьютера, в том числе на носителе, в котором находится файловая система, физическое уничтожение информации (в том числе и случайное) и т.д. Первым и наиболее исторически

традиционным путем решения этой проблемы является резервное копирование или резервное архивирование файла.

=>

Нужно, чтобы при возникновении внештатной ситуации потеря информации стремилась к нулю.

Резервное копирование должно проходить в ограниченный промежуток времени.

Различные модели резервного копирования

1.Копируются не все файлы файловой системы (избирательность архивирования по типам файлов); Не копируются *.obj
и системные (те, которые можно переустановить).

2. Создается мастеркопия архива по расписанию, или в произвольные моменты времени. Копируются все файлы, которые
были изменены с момента создания последней мастеркопии. Копии получаются очень большие.

3. Использование компрессии при архивировании (риск потери всего архива из-за ошибки в чтении/записи сжатых данных);

4. Проблема архивирования «на ходу» (во время копирования происходят изменения файлов, создание, удаление каталогов
и т.д.). Возможность потери информации из-за внештатной ситуации (например отключение от питания)

5. Распределенное хранение резервных копий. Лучше держать много копий в разных местах – хоть где-нибудь, да
останется.

1.1.1.1.1.1.1.1Стратегии архивирования

Физическая архивация

1.«один в один»;
При этом можно копировать свободные блоки, что не надо. Решение этой проблемы – п.2

2. интеллектуальная физическия архивация (копируются только использованные блоки файловой системы);

3. проблема обработки дефектных блоков. Чем больше носителей, тем больше дефектных блоков.

Логическая архивация – копирование файлов (а не блоков), модифицированных после заданной даты.

Проверка целостности файловой системы

Проблема – при аппаратных или программных сбоях возможна потеря информации:

• потеря модифицированных данных в «обычных» файлах;

• потеря системной информации (содержимое каталогов, списков системных блоков, индексные узлы и т.д.)

Необходим контроль целостности или непротиворечивости файловой системы.

Модельная стратегия контроля

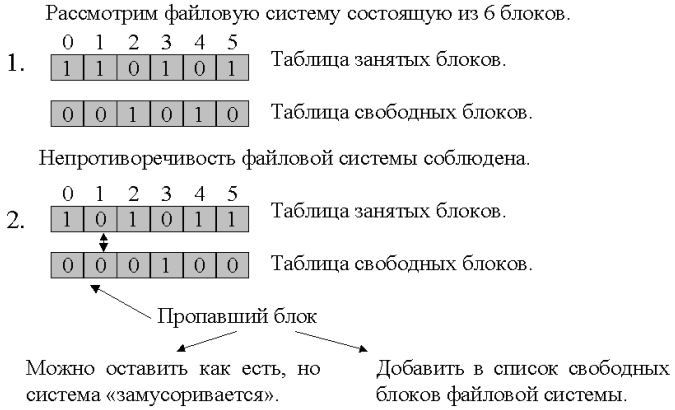
1.Формируются две таблицы:
- таблица занятых блоков;
- таблица свободных блоков;

(размеры таблиц соответствуют размеру файловой системы – число записей равно числу блоков ФС)
Изначально все записи таблиц обнуляются.
2. Анализируется список свободных блоков. Для каждого номера свободного блока увеличивается на 1 соответствующая ему запись в таблице свободных.
3. Анализируются все индексные узлы. Для каждого блока, встретившегося в индексном узле, увеличивается его счетчик на 1 в таблице занятых блоков.
4. Анализ содержимого таблиц и коррекция ситуаций.

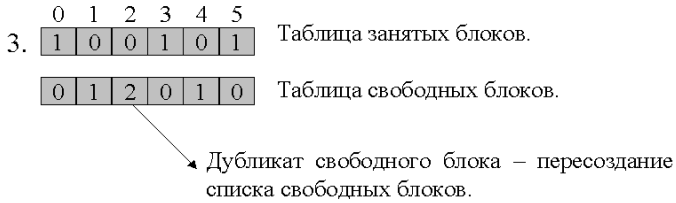
Варианты анализа таблиц

1.Таблица занятых блоков и таблица свободных блоков дополняют друг друга до всех единиц, тогда все в порядке,
целостность системы соблюдена.

2. Пропавший блок – не числится ни среди свободных, ни среди занятых. Можно оставить как есть и ждать претензий со стороны пользователя, но система замусоривается. Считаем свободным.

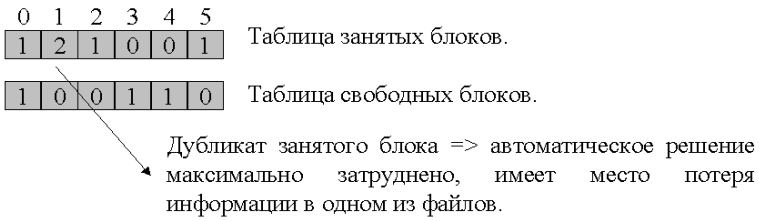


3. таблица занятых блоков корректна, а какой-то из свободных блоков дважды или более раз посчитан свободным, т.е. список свободных блоков (таблица) не корректен. В этом случае нужно запустить процесс пересоздания списка свободных блоков. Т.е. нужно запустить процесс, который проанализирует все индексные дескрипторы и соответственно сформирует список свободных блоков.



4.Дубликат занятого блока. Блок повстречался в 2х индексных дискрипторах. Локализуем проблему на уровне файлов.

4.



Действие:

1.$Name_1$ ---> копируется $Name_1^2$

2.$Name_2$ ---> копируется $Name_2^2$

3.Удаляются $Name_1$, $Name_2$

4.Запускается переопределение списка свободных блоков

5.Обратное переименование файлов и фиксация факта их возможной проблемности.

Билет 49. ОС Unix: файловая система

Появление ФС UNIX совершило революцию в файловых системах по нескольким направлениям – в СПО, в организации ОС.
Например:
1.UNIX был первой ОС, разработанной с помощью языка высокого уровня.
2. Элегантная и развитая система управления процессами.
3. Древовидная организация ФС. (Древовидная в общем случае, но есть средства нарушающие древовидность.) Такая организация – не абсолютное первенство UNIX, это было заимствовано из Maltix, однако UNIX - первая файловая система с древовидной организацией, которая получила широкое распространение.
4. Использование концепции файлов. Все представляется в виде файлов. Все работает через унифицированный интерфейс.

Организация ФС Unix

Файловая система операционной системы **UNIX** является примером многопользовательской иерархической файловой системой с трехуровневой организацией прав доступа к содержимому файлов.

Файл Unix – это специальным образом именованный набор данных, размещенный в файловой системе.
•***обычный файл*** (regular file) – традиционный тип файла, содержащий данные пользователя. Интерпретация содержимого файла производится программой, обрабатывающей файл.
ОС Unix трактует понятие файла шире традиционного. В частности, в системе в качестве файла рассматриваются:
•***каталог*** (directory) – специальный файл, обеспечивающий иерархическую организацию файловой системы. С каталогом ассоциируются все файлы, которые принадлежат данному каталогу.
•***специальный файл устройств*** (special device file) – система позволяет ассоциировать внешние устройства с драйверами и предоставляет доступ к внешним устройствам, согласно общим интерфейсам работы с файлами.
•***именованный канал*** (named pipe) – специальная разновидность файлов, позволяющая организовывать передачу данных между взаимодействующими процессами;
•***ссылка*** (link) – позволяет создавать дополнительные ссылки к содержимому файла из различных точек файловой системы; Они могут нарушать древовидность организации ФС.
•***сокет*** (socket) – средство взаимодействия процессов в пределах сети ЭВМ.

ОС UNIX поддерживает широкий диапазон типов файлов. Каталог в ОСУ тоже файл.

Права доступа

Категории пользователей:

1. пользователь (владелец)
2. группа (всепользователи, которые принадлежат группе владельца за исключением самого владельца)
3. все пользователи системы (все пользователи системы, за исключением группы владельца и самого владельца.)

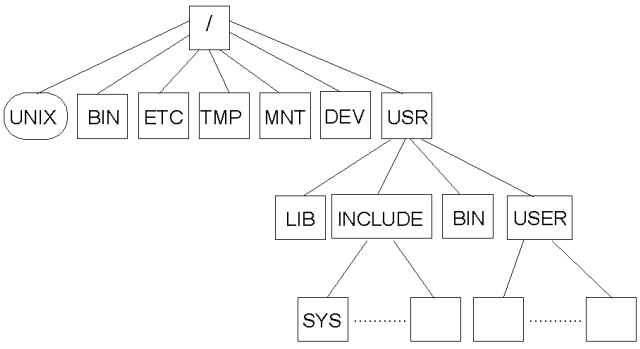
Права

1. на чтение
2. на запись
3. на исполнение (0исполняемым файлом может быть только файл полученный в результате сборки и ли командный файл.

Интерпретация этих прав зависит от типа файла.
Так для обычных файлов это традиционные права на чтение, запись данных файла и исполнение содержимого файла в качестве процесса.
Интерпретация прав доступа для других типов файлов может различаться.
Например, для файлов каталогов это:

- право на чтение каталога – получение списка имен файлов;
- право на исполнение каталога – получение дополнительной информации о файлах (т.е. тогда, когда требуется информация, большая чем имя файла), право на использование каталога в качестве текущего, возможность использования имени каталога внутри имени файла;
- право на запись – возможность создания, переименования и удаления файла в каталоге.

Логическая структура каталогов



Все UNIX-системы имеют соглашения о логической структуре каталогов, расположенных в корне файловой системы. Это упрощает работу операционной системы, ее обслуживание и переносимость. Эти соглашения используются при работе почтовой системы, системы печати и т.д.

Содержимое основных каталогов:
Корневой каталог / является основой любой файловой системы ОС UNIX. Все остальные файлы и каталоги располагаются в рамках структуры, порожденной корневым каталогом, независимо от их физического положения на диске.
/unix - файл загрузки ядра ОС.

/bin - файлы, реализующие общедоступные команды системы.
/etc - в этом каталоге находятся файлы, определяющие настройки системы (в частности, файл passwd), а также команды, необходимые для управления содержимым подобных специальных файлов.
/tmp - каталог для хранения временных системных файлов. При перезагрузке системы не гарантируется сохранение его содержимого. Обычно этот каталог открыт на запись для всех пользователей системы.
/mnt - каталог, к которому осуществляется монтирование дополнительных физических файловых систем для получения единого дерева логической файловой системы. Заметим, что это лишь соглашение, в общем случае можно примонтировать к любому каталогу.
/dev - каталог содержит специальные файлы устройств, с которыми ассоциированы драйверы устройств. Каждый из файлов имеет ссылку на соответствующий драйвер и указание типа устройства (блок- или байт-ориентированные). Этот каталог может содержать несколько подкаталогов, группирующих специальные файлы по типам. Таким образом, имеется возможность легко добавлять и удалять новые устройства в систему.
/lib - здесь находятся библиотечные файлы языка Си и других языков программирования.
/usr - размещается вся информация, связанная с обеспечением работы пользователей. Здесь также имеется подкаталог, содержащий часть библиотечных файлов (**/usr/lib**), подкаталог **/usr/users** (или **/usr/home**), который становится текущим при входе пользователя в систему, подкаталог, где находятся дополнительные команды (**/usr/bin**), подкаталог, содержащий файлы заголовков (**/usr/include**), в котором, в свою очередь, подкаталог, содержащий include-файлы, характеризующие работу системы (например, signal.h - интерпретация сигналов).

Билет 50. Модель версии System V

Структура ФС

Файловая система Unix может занимать раздел диска (partition). Количество разделов на каждом диске, их размеры определяются при предварительной подготовке устройства (разметка). Unix рассматривает разделы как отдельные, независимые устройства.

Суперблок файловой системы содержит оперативную информацию о текущем состоянии файловой системы, а также данные о параметрах настройки, в частности:

•размер логического блока (512б, 1024б, 2048б);
•размер файловой системы в логических блоках (включая суперблок);
•максимальное количество индексных дескрипторов (определяет размер области индексных дескрипторов);
•число свободных блоков;
•число свободных индексных дескрипторов;
•специальные флаги;
•массив номеров свободных блоков;
•массив номеров свободных индексных дескрипторов;
•и др.

В ОП постоянно находится актуальная копия суперблока.

Область (пространство) индексных дескрипторов.

Индексный дескриптор – это специальная структура данных файловой системы, которая ставится во взаимно однозначное соответствие с каждым файлом.

Размер пространства индексных дескрипторов определяется параметром генерации файловой системы по количеству индексных дескрипторов, которые указаны в суперблоке.

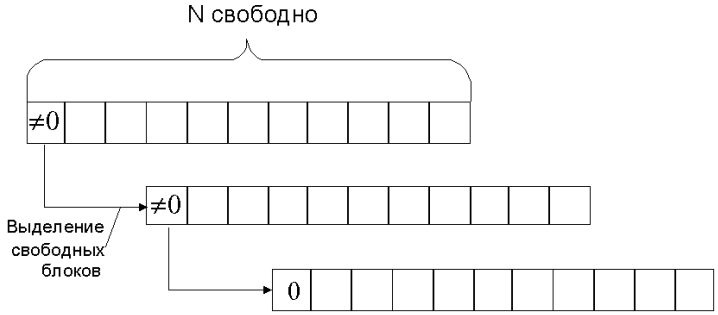
Содержит:

1.Тип файла
2. права доступа к файлу
3. число имен каталогов ФС, ассоциированных с данным индексным дискриптором.
4. идентификатор владельца
5. размер файлда в байтах
6. время послдней модификации
7.Массив номеров блоков файлов

Блоки файлов.

Это пространство на системном устройстве, в котором размещается вся информация, хранящаяся в файлах и о файлах, которая не поместилась в предыдущие блоки файловой системы.

Работа с массивами номеров свободных блоков

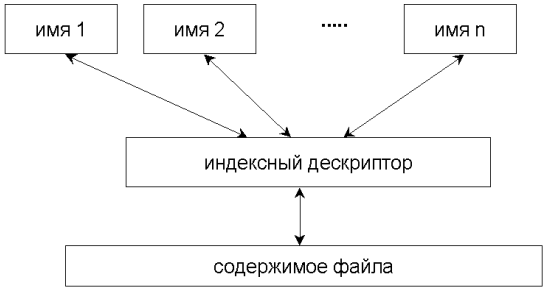


В суперблоке файловой системы размещается ***массив номеров свободных блоков,*** этот массив является началом полного списка содержащего номера всех свободных блоков файловой системы.
Все свободные блоки ФС организованы в однонаправленный список, структурная организация которого следующая: 1-й элемент этого списка – это есть массив из N ссылок, которые размещаются в суперблоке. N зависит от конкретной ОС, пусть это будет 100. 0-й элемент этого массива есть номер блока из пространства блоков ФС, в котором находится продолжение этого списка. Соответственно 0-й элемент этого блока есть ссылка на следующий массив из N ссылок и т.д. ФС оперативно работает с этим массивом. Если в нем есть свободные места, то при освобождении блоков, они записываются на свободные места, если требуются новые блоки, то они выбираются из этого массива. Если массив исчерпывается, то информация берется из следующего блока. Если массив полностью заполнен, т.е. освобождается много блоков, то выбирается следующий свободный блок и этот массив скидывается на этот блок. Это достаточно важная информация, которая в каждый момент отражает состояние ФС.
Оперативный доступ к списку осуществляется посредством использования массива в суперблоке.

Работа с массивом свободных ИД

Массив номеров свободных индексных дескрипторов содержит оперативный набор номеров свободных индексных дескрипторов. Размер массива - Nиндекс.
При освобождении индексного дескриптора, если есть свободное место в массиве, то номер освободившегося индексного дескриптора записывается в соответствующий элемент массива. Если свободного места в массиве нет, то этот номер «забывается».
При запросе нового индексного дескриптора осуществляется поиск в массиве, если массив не пустой, то все в порядке, если массив пустой – происходит операция обновления его содержимого (происходит просмотр области индексных дескрипторов и занесение в массив обнаруженных свободных). Т.е. массив свободных индексных дескрипторов – это своеобразный буфер.

Индексные дескрипторы

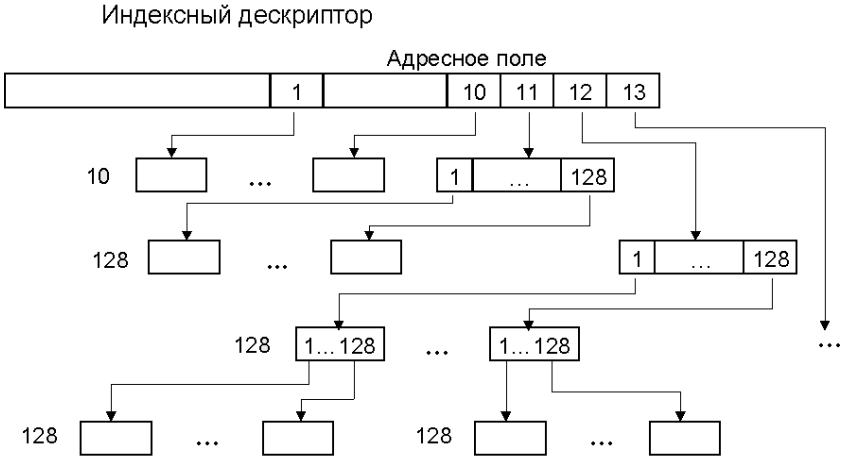


Индексный дескриптор (ИД) – описатель файла, содержит все необходимые для работы с файлом служебные атрибуты.

Через ИД осуществляется доступ к содержимому файлов. Любое имя файла в системе ассоциировано с единственным ИД, но это соответствие неоднозначно. Т.е. ИД может соответствовать произвольное количество имен.

Структура индексного дескриптора:
•тип файла, права, атрибуты выполнения (если = 0, то ИД свободен);
•число имен, которые ассоциированы с данным ИД;
•идентификаторы владельца-пользователя, владельца-группы;
•размер файла в байтах;
•время последнего доступа к файлу;
•время последней модификации содержимого файла;
•время последней модификации ИД (за исключением времени доступа и времени модификации файла)
•массив номеров блоков файла.

Адресация блоков файла



Для простоты изложения будем считать, что размер блока равен 512 байт.
Размещение данных файла задается списком его блоков.
Это снимает проблемы непрерывных файловых систем, т.е. систем, где блоки файла располагаются последовательно. Таким образом реально блоки файла могут

быть разбросаны по диску, но логически они образуют цепочку, содержащую весь набор данных.

Ключом, задающим подобное расположение служит массив номеров блоков файла, содержащий список из 13 номеров блоков на диске, хранящихся в **ИД**.
Первые десять указывают на десять блоков некоторого файла.
Если файл занимает более 10 блоков, то 11 элемент указывает на косвенный блок, содержащий до 128 адресов дополнительных блоков файла (это еще 70656 байт).
Большие файлы используют 12-ый элемент, который указывает на блок, содержащий 128 указателей на блоки, каждый из которых содержит по 128 адресов блоков файла.
Еще в больших файлах аналогично используется 13 элемент.

Трехкратная косвенная адресация позволяет создавать файлы длиной (10+128+128*128+128*128*128)*512 байт.

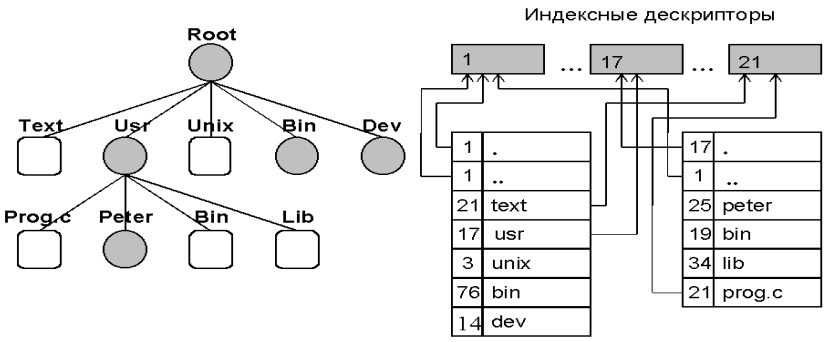
Таким образом,
если файл меньше 512 байт, то необходимо одно обращение к диску,
если длина файла находится в пределах 512-70565 байт, то - два и так далее.

Приведенный способ адресации позволяет иметь прямой и быстрый доступ к файлам. Эта возможность также усиливается кэшированием диска, позволяющим хранить в памяти наиболее используемые блоки.

При ***открытии файла*** соответствующий **ИД** считывается в память и системе становятся доступны все номера блоков данного файла.
Для одного и того же файла, открываемого несколько раз, в памяти находится только один **ИД.**
Система фиксирует число открытий данного файла и, когда этот счетчик обнуляется, резидентный образ **ИД** переписывается на диск. Если при этом изменений в файле не было и не модифицировался **ИД,** то запись не выполняется.

Указанные особенности существенно влияют на эффективность файловой системы.

Файл каталог



. Файл каталог для ФС System V представляет собой таблицу, каждая запись которой состоит из 16 байтов. Первые 2 байта – это номер индексного дескриптора. Последующие 14 байтов – это поле для имени файла. Соответственно имеется

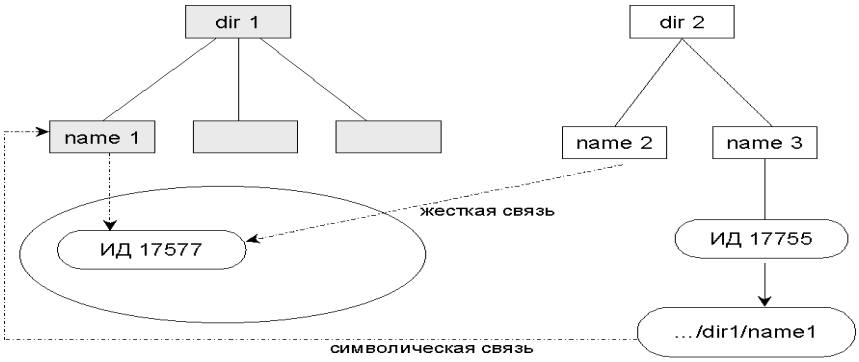
предопределенные записи в этих полях – это первые две строчки. 1-я строчка – это ссылка на самого себя, т.е. в этой строчке находится имя «.» (точка) и номер индексного этого файла каталога. Следующая запись – это ссылка на родительский каталог, соответственно в нем имеется номер индексного дескриптора и имя «..» (две точки).Содержимое файла – таблица. 1-е поле – это номер индексного дескриптора (ИД), которому соответствует имя Name из второго поля.

Размеры полей в общем случае могут быть различные.
Например размер поля ИД – 2 байта (ограничение числа ИД в файловой системе 65535), размер поля Name – 14 байт (соответственно ограничение на длину имени).

В Unix две первые строки любого каталога имеют фиксированное содержание: имя «•» - ссылка на самого себя, имя «••» - ссылка на родительский каталог.

Видно, что при такой реализации имя файла “отделено” от других его атрибутов. Это позволяет, в частности, один и тот же файл внести в несколько каталогов. При этом, как отмечалось выше, данный файл может иметь разные имена в разных каталогах, но ссылаться они будут на один и тот же **ИД**, который является ключом для доступа к данным файла. При обсуждении понятия **ИД** говорилось, что каждая новая ссылка к **ИД** отмечается в специальном поле. Рассмотрим пример, иллюстрирующий связь файлов с каталогами.

Установление связей



Древовидность файловой системы Unix нарушается возможностью установления ссылок на одни и те же индексные дескрипторы из различных каталогов. Это может быть достигнуто за счет использования средств установления дополнительных связей.

Существует две разновидности этой операции.

Жесткая связь -. с одним и тем же индексным дескриптором будет ассоциироваться два или более имени, размещенных в произвольных точках ФС. При этом каждое из этих имен равноценно.
Для этого используется команда: **ln …dir1/name1 …dir2/name2** – (дли индексного дескриптора, с которым ассоциировано имя **name1** добавляется еще одно имя – **name2**).

Все имена, ассоциированные таким образом с индексным дескриптором равноправны.
При этом увеличивается значение поля индексного дескриптора *число имен, которые ассоциированы с данным ИД.*
Нельзя устанавливать жесткую связь для файлов-каталогов.

Установление ***символической связи*** - косвенная адресация на существующее имя файла.
В ФС можно создать специальный *файл ссылку*, содержимое которого размещается в индексном дескрипторе этого файла. Этим содержимым является текстовая строка, указывающая полное имя того файла, с которым нужно ассоциировать новое имя (имя файла-ссылки). Т.е. если name 1 и name 2 - это абсолютно равноправные файлы, то name 3 – это текстовая (символьная ссылка). Для name 3 создается свой индексный дескриптор и через него организуется ссылка на файл name 1, при этом уже в индексном дескрипторе файла name 1 никакой информации о дополнительных ссылках на этот файл нет. Т.е. здесь уже некоторая асимметричная модель множественного именования содержимого файла. Если будет нужно удалить файл name 1, то система позволит это сделать, потому что нигде в информации, связанной с этим файлом, не указывается, что на него есть текстовая ссылка. И соответственно, когда после удаления этого файла произойдет обращение по ссылке /dir1/name1, то уже возникнут какие-то проблемы. Следует помнить, что не на любой файл можно установить ссылку.

Для этих целей используется команда: **ln –s …dir1/name1 …dir2/name3** – в результате образуется специальный ***файл - ссылка.***

Достоинства ФС модели версии System V
1)Оптимизация в работе со списками номеров свободных индексных дескрипторов и блоков.
2)Организация косвенной адресации блоков файлов, позволяющая использовать эффективный доступ к значительному количеству блоков файла.

Недостатки ФС модели версии System V

1)Концентрация важной информации в суперблоке - ключевая информация сконцентрирована в суперблоке файловой системы, физическая потеря содержимого суперблока может приводит к значительным проблемам, касающимся целостности файловой системы.
2)Проблема надежности (много ссылочных структур, возможна потеря данных при сбоях).
3)Фрагментация файла по диску – т.е. при достаточно больших размерах файла его блоки могут произвольным образом размещаться на физическом МД? Что может приводить к выполнению значительного числа механических операций перемещения головок устройства при чтении/записи данных файла.

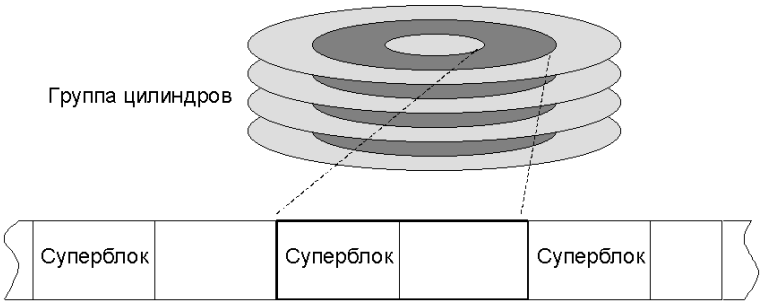
4)Организация каталога накладывает ограничения на возможную длину имени файла (14 символов).

Билет 51. Модель версии FFS BSD

В Unix 4.2 BSD разработана модель организации файловой системы, которая получила название Fast File System - FFS (быстрая файловая система).

Основной идеей данной модели файловой системы является **кластеризация дискового пространства файловой системы**, с целью минимизации времени чтения/записи файла, а также уменьшения объёма не используемого пространства внутри выделенных блоков.

Суть кластеризации заключается в следующем. Дисковое пространство, также, как и в модели s5fs имеет суперблок в котором размещена ключевая информация файловой системы (структура суперблоков s5fs и ffs, в общем случае, логически идентична), далее, дисковое пространство разделено на области одинакового размера, называемые группами цилиндров. Далее, стратегия функционирования файловой системы такова, что она старается разместить содержимое файлов (блоки файлов) в пределах одной группы цилиндров, при этом стараясь располагать файлы в той же группе цилиндров, что и каталог в котором они расположены.



Группа цилиндров:
- копия суперблока
- информация о свободных блоках (битовый массив) и о свободных индексных дескрипторах
- массив индексных дескрипторов (ИД)
- блоки файлов

Стратегии размещения

1)Новый каталог помещается в группу цилиндров, число свободных индексных дескрипторов в которой больше среднего значения во всей файловой системе в данный момент времени, а также имеющей минимальное число дескрипторов каталогов в себе;

2)для обеспечения равномерности использования блоков данных файл разбивается на несколько частей, при этом первая часть файла располагается в той же группе цилиндров, что и его дескриптор, при размещении последующих частей используется группа цилиндров, в которой число свободных блоков превышает среднее значение. Длина первой части выбирается таким образом, чтобы она адресовалось непосредственно индексным дескриптором (т.е. не «косвенно»), остальные части разбиваются фиксированным образом, например по 1 мегабайту;
3)последовательные блоки файлов размещаются исходя из оптимизации физического доступа (см. ниже)

Dt – технологический промежуток времени, который затрачивает система на передачу и прием устройством МД команды на чтение очередного блока. За это время диск проворачивается и головки обмена «пропускают» начало очередного блока, поэтому если мы будем читать следующий блок, то головка будет вынуждена ожидать полного поворота диска на начало блока. В связи с этим эффективнее читать не последовательные блоки (в этом случае нужно ожидать полного поворота диска), а блоки, размещенные на диске через один, два... (смещение определяется поворотом диска за время Dt).

Внутренняя организация блоков

Блоки	0				1				…	N			
Фрагменты		1	2	3	4	5	6	7	…				
Маска	1	0	0	0	0	1	1	1					

Обмен происходит блоками. Блоки могут быть достаточно большого размера (до 64 Кб). В системе может быть принято разбиение блока на равные фрагменты (на 2, 4, 8). То есть все пространство разделяется на «маленькие блоки» - фрагменты. Фрагменты группируются по 2, 4 или 8 в блоки (т.е. если фрагмент содержит 512 байт, то блок может быть размера 1024, 2048, 4096).

При этом блоком в этой системе может называться только «выровненный» до размера кратности набор фрагментов. Т.е. при кратности 4 (см. рисунок выше), фрагменты 0 – 3 – входят в один блок, а фрагменты 1 – 4 нет.

Для хранения информации о свободных фрагментах используется битовая маска: каждому фрагменту на диске соответствует ровно 1 бит в этой маске (этот механизм упрощает алгоритм поиска свободных фрагментов и уменьшает «фрагментацию» свободного пространства).

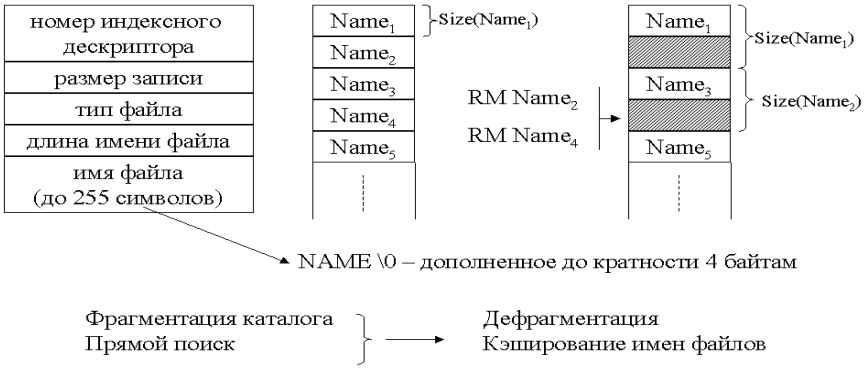
Формат индексного дескриптора аналогичен, используемому в s5fs - в нём в качестве элементов по-прежнему используются блоки, а не фрагменты, но при размещении информации в файлах используется следующее простое правило: все блоки указанные в индексном дескрипторе, кроме последнего, должны использоваться только целиком; блок может использоваться для нескольких файлов только при хранении их последних байт, не занимающих всех фрагментов полного блока (см. рисунок ниже). Т.о. для хранения информации об использовании последнего блока недостаточно только размера файла, хранимого в

дескрипторе System 5, необходимо также хранить информацию, об используемых фрагментах в этом блоке.

Выделение пространства для файла происходит только в момент когда процесс выполняет системный вызов write. Операционная система при этом руководствуется следующим алгоритмом:

1. Если в уже выделенном файлу блоке есть достаточно места, то новые данные помещаются в это свободное пространство.
2. Если последний блок файла использует все фрагменты (т.е. это полный блок) и свободного в нём места не достаточно для записи новых данных, то частью новых данных заполняется всё свободное место. Если остаток данных превышает по размеру один полный блок, то новый выделяется полный блок и записываются данные в этот полный блок. Процесс повторяется до тех пор, пока остаток не окажется меньше чем полный блок. В этом случае ищется блок с необходимыми по размеру фрагментами или выделятся новый полный блок. Остаток данных записывается в этот блок.
3. Файл содержит один или более фрагмент (они естественным образом содержатся в одном блоке) и последний фрагмент недостаточен для записи новых данных. Если размер новых данных в сумме с размером данных, хранимых в неполном блоке, превышает размер полного блока, то выделяется новый полный блок. Содержимое старого неполного блока копируется в начало выделенного блока и остаток заполняется новыми данными. Процесс далее повторяется, как указано в пункте 2 выше. В противном случае (если размер новых данных в сумме с размером данных, хранимых в неполном блоке, не превышает размер полного блока) ищется блок с необходимыми по размеру фрагментами или выделятся новый полный блок. Остаток данных записывается в этот блок.

Структура каталога FFS



Поддержка длинных имен файлов.

Любая запись содержит:

- номер индексного дескриптора;
- длина записи в каталоге;
- длина имени файла;
- имя файла (дополненное до кратности слова).

Структура каталога немного изменяется. К двум содержательным полям добавляется номер индексного дескриптора, размер записи, т.е. записи каталога, тип файла, длина имени и имени разрешается быть длиной до 256 символов. Соответственно может возникнуть некоторое недопонимание, т.к. есть параметр размер записи и длина имени. Суть использования того и другого параметра заключается в том, что при удалении информации (какого-то имени) из каталога свободное пространство присоединяется к предыдущей записи и получается, что размер больше той содержательной информации, которая имеется. Соответственно может появиться внутренняя фрагментация.

Билет 52. Управление внешними устройствами. Архитектура организации управления внешними устройствами, основные подходы, характеристики.

Архитектура.

Непосредственное управление Внешними устройствами ЦП. В основном требуется переместить данные из ВУ в ОЗУ (и наоборот). ЦП по своей инициативе почти никогда не обращается к ОЗУ.
Историческая модель основана на том, что управление осуществлялось с помощью ЦП.
Когда говорится о том, что организовано управление внешним устройством, то подразумевается, что реализуется два потока информации:

- поток управляющей информации (команды).
- поток данных, т.е. поток той информации, которая начинает двигаться от ОП к внешнему устройству, за счет потока управляющей информации.

Поток управляющей информации обеспечивает управление ВУ, поток данных начинает двигаться от ВУ к ОЗУ в результате выполнения 1ого потока. Оба потока обрабатывает ЦП, что «отвлекает» его от других задач пользователя.

Синхронное управление внешними устройствами с использованием контроллеров внешних устройств.
В результате развития аппаратной части компьютера появляются контролеры внешнего устройства. Он упростил жизнь ЦП. Все равно поток команд идет через ЦП, *контролер взял некоторые функции*:
1.Обнаружение ошибок
2. Обеспечение более высокоуровнего интерфейса по управлению ВУ
3. Позволяет использовать команды типа «вывести головку на нужный сектор», «… на нужный цилиндр»

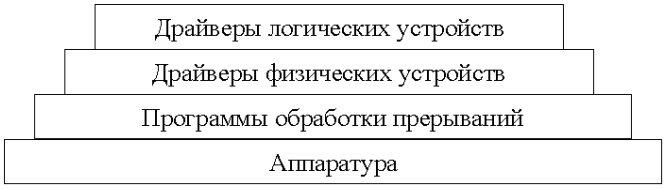
4. Появилось разделение функций синхронизации. ЦП подавал сигнал и ждал.
В результате развития аппаратных прерываний появилась возможность использовать **асинхронный** режим работы.

Асинхронное управление внешними устройствами с использованием контроллеров внешних устройств.

Появление **контролеров прямого доступа** позволяет вывести поток данных, который появляется при обмене с ВУ из ЦП. Это имеет смысл для блок-ориентированных устройств, подразумевающих большой поток информации. Поток управляющей информации остается в ведении ЦП.

Управление внешними устройствами **с использованием процессора или канала ввода/вывода**.
Наличие **процессоров ввода-вывода** позволяет обеспечить высокоуровневый интерфейс для ЦП при управлении внешними устройствами. ЦП предоставляются различные макрокоманды. (например «записать на диск … начиная с …места»)

Программное управление внешними устройствами



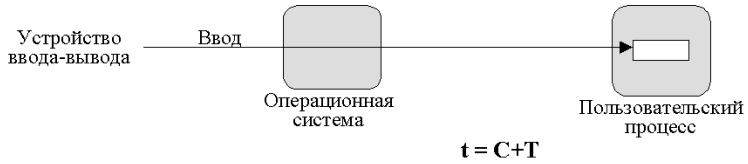
<u>Цели, которые стоят перед программным обеспечением</u>:
1. унификация программных интерфейсов доступа к внешним устройствам (унификация именования, абстрагирование от свойств
конкретных устройств);
2. обеспечение конкретной модели синхронизации при выполнении обмена (синхронный, асинхронный обмен);
3. обработка возникающих ошибок (индикация ошибки, локализация ошибки, попытка исправления ситуации);
корректно обработать эту ситуацию, минимизировать негативные последствия.
4. буферизация обмена – в системе очень многоуровневая, применяется на всех этапах:
- развитые канала ввода-вывода могут иметь встроенный КЭШ, который управляется внутри этих каналов. Эта функция
остается на уровне ОС, этот КЭШ ОС полностью программноориентирован.
5. обеспечение стратегии доступа к устройству (распределенный доступ, монопольный доступ);

6. планирование выполнения операций обмена – возникает, когда возникает конкуренция за доступ к ресурсу.

Билет 53. Управление внешними устройствами. Буферизация обмена. Планирование дисковых обменов, основные алгоритмы.

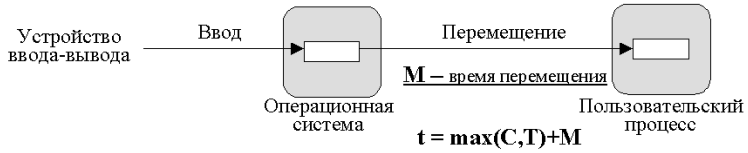
Буферизация обмена

T – время обмена;
C – время выполнения программы между обменами
t – общее время выполнения программы
Схемы буферизации ввода-вывода
а) Без буферизации



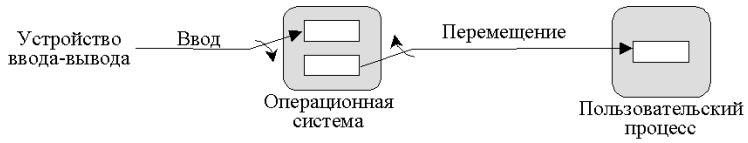
Если обмен проходит без буферизации, то совокупное время выполнения программы будет складываться из времени обмена и времени выполнения программы между обменами.

б) Одинарная буферизация



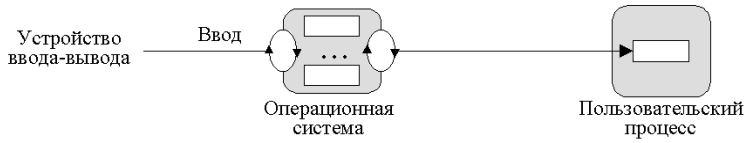
При использовании одиночной буферизации подавляется заказ на обмен с ОП, и процесс может в этом случае не ожидать. Целесообразно использовать, когда идет интенсивный поток заказов на обмен.

в) Двойная буферизация



Модель использования двойной буферизации следующая: в один буфер помещаются данные по обмену, в другой ОС готовит данные за предыдущий обмен.

г) Циклическая буферизация



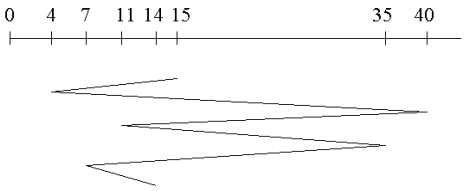
Какую схему выбрать зависит от интенсивности буферизации и особенности действий

Планирование дисковых обменов

Возможна ситуация, когда поток заказов на обмен > пропускной способности системы в некоторые моменты.
Тогда есть несколько вариантов действий:
1.Принимаем решения о порядке обработки запросов
2. начинаем учитывать приоритеты
3. осуществляем случайный выбор.
Проблема: Обмены могут быть зависимы друг от друга. В таком случае некоторые варианты не подходят.
Пусть наш диск может сразу переходить i-ой дорожки на j-ую без начального позиционирования.
Рассмотрим модельную ситуацию:
головка HDD позиционирована на дорожке 15
Очередь запросов к дорожкам: 4, 40, 11, 35, 7, 14
Варианты решения
1. простейшая модель – случайная выборка из очереди
2.

FIFO

Путь головки	L
15 → 4	11
4 → 40	36
40 → 11	29
11 → 35	24
35 → 7	28
7 → 14	7
общ. 135 средн. 22,5	



Общее время выполнения – 135ед.
Среднее время выполнения – 21.5 ед.

3.SSTF

SSTF

Путь головки	L
15 → 14	1
14 → 11	3
11 → 7	4
7 → 4	3
4 → 35	31
35 → 40	5
общ. 47 средн. 7,83	

Приоритет имеет обмен, для которого потребуется наименьшее время. «Жадный» алгоритм на каждом шаге пытается получить максимальный эффект. Общая нагрузка на систему с точки зрения обмена сокращается в 3 раза. Возможно «залипание» головки в том случае, если обмен идет интенсивно с одними и теми же дорожками. Некоторые процессы будут отделены.

4.LIFO

LIFO

Путь головки	L
15 → 14	1
14 → 7	7
7 → 35	28
35 → 11	24
11 → 40	29
40 → 4	36
общ. 126 средн. 20,83	

Смысл – попытка развязать последовательность обмена, связанную с новыми источниками.

Приоритетный алгоритм (RPI) – это алгоритм, когда последовательность обменов (очередь) имеет характеристику приоритетов. При использовании приоритетных алгоритмов может возникать проблема голодания или дискриминации. Проблема дискриминации возникает при непрерывном поступлении более приоритетных запросов на обмен, в это время как менее приоритетные запросы простаивают.

SCAN

Путь головки	L
15 → 35	20
35 → 40	5
40 → 14	26
14 → 11	3
11 → 7	4
7 → 4	3
общ. 61 средн. 10,16	

Находясь в начальной позиции сначала двигаемся в одну сторону до конца, затем в другую до конца.
Для " набора запросов
перемещений £ 2 x число дорожек

C-SCAN

Путь головки	L
15 → 4	11
4 → 7	3
7 → 11	4
11 → 14	3
14 → 35	21
35 → 40	5
общ. 47 средн. 7,83	

Выходим на минимальную (максимальную дорожку, а затем движемся в одну сторону). Пройдем не более двух маршрутов.

N-step-SCAN

Разделение очереди на подочереди длины £ N запросов каждая (из соображений FIFO). Последовательная обработка очередей. Обрабатываемая очередь не обновляется. Обновление очередей, отличных от обрабатываемой.
Этот алгоритм срывает головку с залипания.

Распространенный пример: 2 очереди, одна обрабатывается, другая собирает вновь поступающие запросы.

Билет 54 .RAID системы.

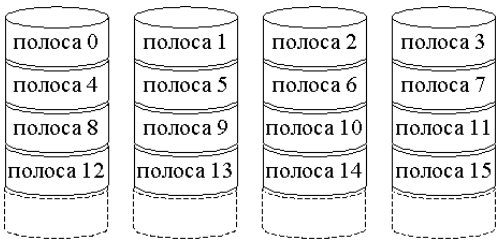
Существуют проблемы с организацией больших потоков данных.
В общем случае для дисковых систем имеют место как минимум две проблемы:

1. **Эффективность**. Допустим, в системе присутствуют все уровни КЭШ, но производительности не хватает, так как обмены, которые производятся на дисковых устройствах, медленные.
2. **Надежность.** Является одним из основных качеств любого программного решения. Соответственно есть необходимость создания надежных дисковых систем.

Все это обусловило появление так называемых RAID систем. Вначале RAID переводили как избыточный массив недорогих дисков. Со временем понятие RAID системы изменилось и на сегодняшний день оно переводится как избыточный массив независимых дисков.
Итак, RAID система представляет собой набор независимых дисков, которые рассматриваются ОС как единое дисковое устройство, где данные представляются в виде последовательности записей, которые называются полосы. /*Полосы цилиндрически распределены по дисковому устройству. */
Рассмотрим модели организации многодисковых систем, которые относятся к классу RAID.

Семь уровней RAID систем.

RAID 0 (без избыточности)
Не является настоящим **RAID** уровнем, поскольку не использует избыточность для повышения эффективности.
Пользовательские и системные данные распределяются по всем дискам массива. Это лучше, чем использовать один большой диск, так как появляется вероятность того, что два



различных блока памяти, к которым поступили два различных запроса ввода\вывода, размещены на различных дисках, вследствие чего эти два запроса могут обрабатываться параллельно.

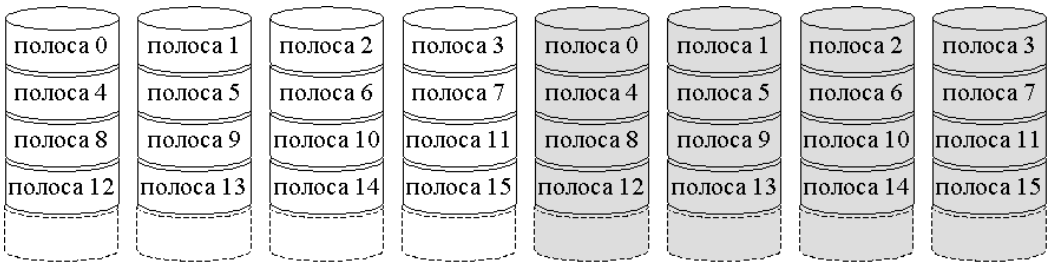
Все пользовательские и системные данные рассматриваются как хранящиеся на одном логическом диске. Диск делится на полосы, которые могут быть физическими блоками, селекторами или другими единицами хранения. Полосы циклически размещаются на последовательных дисках массива. В n-дисковом массиве первые n полос располагаются как первые полосы каждого из n дисков; вторые n- как вторые полосы каждого из n дисков и т.д.

«+» Если один запрос ввода\вывода обращается к множеству логически последовательных полос, то параллельно может быть обработано до n полос. Уменьшается время обработки.

RAID 1 (зеркалирование Предполагает наличие массивов устройств. 1ая группа – циклическое распределение устройств по уровням 2ая группа-копия первой. Запись идет параллельно и независимо)

«+»

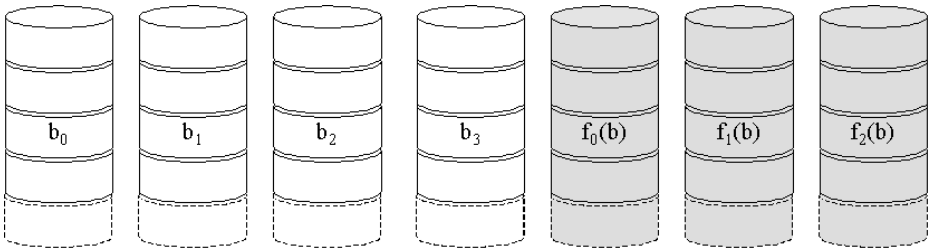
1. Запрос на чтение может быть обслужен любым из двух дисков, содержащих необходимые данные; для обслуживания выбирается диск, у которого минимальное время поиска.
2. Для запроса на запись необходимо обновление обеих полос, что может быть выполнено в параллельном режиме. Поэтому скорость записи определяется самой медленной из них (т.е. той, для которой время поиска оказывается большим). Однако никаких дополнительных расходов на запись не требуется.
3. Простота восстановления данных в случае сбоя



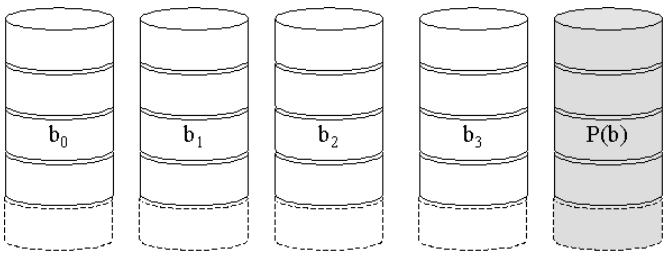
RAID первого уровня это достаточно дорогостоящая конструкция, потому что получается двойное резервирование, но тем не менее эта система наиболее просто организована.

RAID 2 избыточность с кодами Хэмминга (Hamming, исправляет одинарные и выявляет двойные ошибки) Также используется разделение на полосы. Полосы оказываются очень малыми; нередко они соответствуют одному байту или слову. Обмен с синхронизацией головок чтения записи. Часть дисковых устройств

предназначены для хранения содержательной части информации. Существует несколько дисковых устройств, в которых реализованы коды Хемминга.
При считывании осуществляется одновременный доступ ко всем дискам. Данные запроса и код коррекции ошибок передаются контролеру массива. При наличии однобитовой ошибки контролер способен быстро ее откорректировать, так что доступ для чтения в этой схеме не замедляется.
При записи происходит одновременное обращение ко всем дискам массива.
. Имеют место 2 проблемы:
1.Соответственно избыточность меньше, чем у RAID 1, но все равно она присутствует.
2. Есть зависимые обмены, т.е. обмены, которые организованы на специализированных движениях головок. И соответственно информация сильно распределена по RAID массиву. Т.е. последовательная информация за счет маленького размера полосок распределена. Т.е. одновременно происходит обращение ко всей цепочке. Т.е. нет независимых обменов в каждом дисковом устройстве.

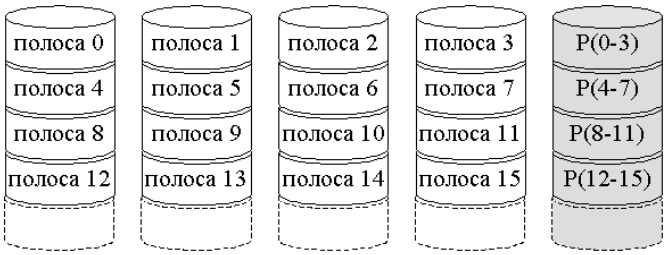


RAID 3 (четность с чередующимися битами) 4 диска содержательные – для размещения логических данных. 5ый – контрольная избыточная информация.
Суть: Если представить, что модель RAID состоит из 5 дисков. В этих 5 дисках 4 диска содержательные, т.е. для размещения логического диска с соответствующими полосками. 5-й диск – это контрольная избыточная информация. Содержимое пятого диска выражается по формулам через содержимое первых 4.То есть определенный разряд 5-го диска представляется как «исключающее или» для соответствующих ему содержательных разрядов. В случае гибели какого-нибудь из устройств утверждается, что информацию на этом устройстве можно восстановить по второй, приведенной ниже, формуле. Т.е. имеет место избыточность, которая с одной стороны дает синхронизированный параллельный доступ, а с другой имеется функция, которая восстанавливает информацию в случае гибели устройства.



Пример: 4 диска данных, один – четности:
Потеря данных на первом диске
X4(i)=X3(i)XOR X2(i)XOR X1(i)XOR X0(i)
X1(i)=X4(i)XOR X3(i)XOR X2(i)XOR X0(i)

RAID 4



Он не синхронизированный, т.е. в этом плане он аппаратно организован проще, чем предыдущие. Схема примерно та же самая: имеется 4 устройства для логического диска, на которых располагаются полосы, и 5-е устройство, в котором находятся контрольные суммы. Контрольная сумма вычисляется по той же самой формуле, что и в RAID 3. И здесь есть проблема работы в случае независимого обмена.

Пример: 4 диска данных, один – четности:

При независимом обмене происходит обновление следующим образом: предположим, что обновление произошло на первом диске.

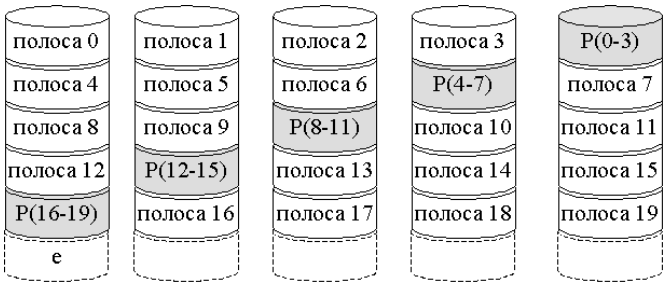
X4(i)=X3(i)XOR X2(i)XOR X1(i)XOR X0(i)

все разряды на 4-м будут обновлены по следующей формуле:

X4new(i)=X4(i)XOR X1(i)XOR X1new(i)

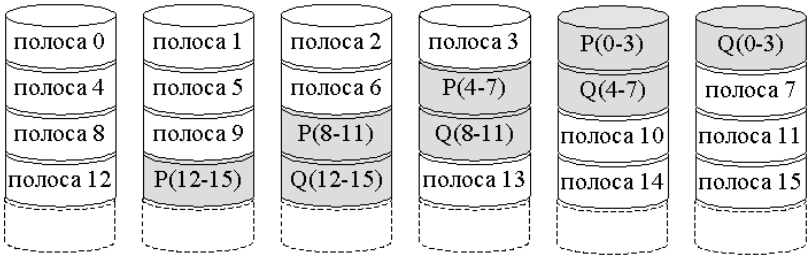
Восстановление информации проходит по предыдущей схеме (это схема обновления, потому что обмены могут быть независимыми, т.е. обмен может происходить только по одной полоске, но для этого необходимо скорректировать содержимое контрольной полоски и использовать ее для восстановления).

RAID 5 (распределенная четность – циклическое распределение «четности»)



RAID 5 - это использование циклического распределения контрольного диска. Суть: в RAID 3 и RAID 4 есть некоторая диспропорция в распределении потока обмена, т.е. сильно нагружено последнее устройство (это плохо тем, что рано или поздно это устройство выйдет из строя первым), на котором находится контрольная сумма. Т.о. контрольный диск циклически распределен по всем устройствам, т.е. вся работа равномерно распределяется.

RAID 6 (двойная избыточность – циклическое распределение четности с использованием двух схем контроля: N+2 дисков)



RAID 6 – это двойная избыточность. Делается еще одно дополнительное устройство для хранения избыточной информации.

Какие-то из RAID массивов можно реализовать чисто программно. Какие-то из них можно реализовать только аппаратно. Какие-то из них можно реализовать в зависимости от решения. Это все относится к проблеме управления внешними устройствами: качеством и свойством работы внешних устройств в системе.

Уровни RAID (сравнение)

Категория	Уровень	Описание	Скорость обработки запросов	Скорость передачи данных	Типичное применение
Расщепление	0	Без избыточности	Большие полосы: отлично	Малые полосы: отлично	Приложения с некритическими данными, требующие высокой производительности
Зеркалирование	1	Зеркалирование	Хорошо/ удовлетворительно	удовлетворительно/ удовлетворительно	Системные диски, важные файлы
Параллельный доступ	2	Избыточность с кодами Хэмминга	Плохо	Отлично	
	3	Четность с чередующимися битами	Плохо	Отлично	Приложения с большими запросами ввода/вывода (графич. редакторы, САПР)

Независимый доступ	4	Четность с чередующимися блоками	Отлично/ удовлетворительно	Удовлетворительно/ плохо	
	5	Распределенная четность с чередующимися блоками	Отлично/ удовлетворительно	Удовлетворительно/ плохо	Высокая скорость запросов, интенсивное чтение, поиск данных
	6	Двойная распределенная четность с чередующимися блоками	Отлично/плохо	Удовлетворительно/ плохо	Приложения, требующие исключительно высокой надежности

Билет 55 ОС Unix: Работа с внешними устройствами

Файлы устройств, драйверы

Особенность UNIX- все устройства обслуживаются в системе виде файлов.

С точки зрения внутренней организации системы, как и в подавляющем большинстве других операционных систем, работа с внешними устройствами осуществляется посредством использования иерархии драйверов, которые позволяют организовывать взаимодействие ядра ОС с конкретными устройствами.

В системе Unix существует единый интерфейс организации взаимодействия с внешними устройствами, для этих целей используются ***специальные файлы устройств***, размещенные в каталоге **/dev**. Файл устройства позволяет ассоциировать некоторое имя (имя файла устройства) с драйвером того или иного устройства. Следует отметить, что здесь мы несколько замещаем понятие устройства понятием драйвер устройства, так как несмотря на то, что мы используем термин ***специальные файлы устройств***, на практике, мы используем ассоциированный с данным специальным файлом драйвер устройства, и таких драйверов у одного устройства может быть произвольное число. Возможно, более удачным было бы использовать специальный файл-драйвер устройства.

В системе существуют два типа специальных файлов устройств:

- **файлы байт-ориентированных устройств** (драйверы обеспечивают возможность побайтного обмена данными и, обычно, не используют централизованной внутрисистемной кэш-буферизации);
- **файлы блок-ориентированных устройств** (обмен с данными устройствами осуществляется фиксированными блоками данных, обмен осуществляется с использованием специального внутрисистемного буферного кэша).

Следует отметить, файловая система может быть создана только на блок-ориентированных устройствах.

В общем случае тип файла определяется свойствами конкретного устройства и организацией драйвера. Конкретное физическое устройство может иметь, как байт-ориентированные драйверы драйверы, так и блок-ориентированные. Например, если рассмотреть физическое устройство *Оперативная память*, для него можно реализовать, как байт-ориентированный интерфейс обмена (и соответствующий байт-ориентированный драйвер), так и блок-ориентированный.

Содержимое файлов устройств размещается исключительно в соответствующем индексном дескрипторе, структура которого для файлов данного типа, отличается от структуры индексных дескрипторов других типов файлов.

Итак индексный дескриптор файла устройства содержит:

- тип файла устройства – байт-ориентированный или блок-ориентированный;
- «старший номер» (major number) устройства - номер драйвера в соответствующей таблице драйверов устройств;
- «младший номер» (minor number) устройства – служебная информация, передающаяся драйверу устройства.

Система поддерживает две таблицы драйверов устройств.

bdevsw – таблица драйверов блок-ориентированных устройств.

cdevsw - таблица байт-ориентированных устройств. Выбор конкретной таблицы определяется типом файла устройства. Соответственно, поле *старший номер* определяет строку таблицы с которой ассоциирован драйвер устройства. Драйверу

устройства может быть передана дополнительная информация через поле *младший номер* это может быть, например, номер конкретного однотипного устройства или некоторая информация, определяющая дополнительные функции драйвера

Каждая запись этих таблиц содержит так называемый коммутатор устройства – структуру, в которой размещены указатели на соответствующие точки входа (функции) драйвера. Таким образом, в системе определяется базовый уровень взаимодействия с драйвером устройства (конкретный состав точек входа определяется конкретной версией системы). В случае, если конкретный драйвер устройства не поддерживает работу с той или иной точкой входа, на ее место устанавливается специальная ссылка-заглушка на точку ядра.
В качестве примера, рассмотрим типовой набор точек входа в драйвер (b - префикс точки входа, характеризующий конкретный драйвер):

- **bopen()** открытие устройства, обеспечивается инициализация устройства и внутренних структур данных драйвера;
- **bclose()** закрытие драйвера устройства, например в том случае, если ни один из процессов не работает с драйвером;
- **bread()** чтение данных;
- **bwrite()** запись данных;
- **bioctl()** управление устройством, задание режимов работы драйвера, определение набора внутренних операций/команд драйвера;
- **bintr()** – обработка прерывания, вызывается ядром при возникновении прерывания в устройстве с которым ассоциирован драйвер;
- **bstrategy()** управление стратегией организации блок-ориентированного обмена (некоторые функции оптимизации организации обмена, обработка специальных ситуаций, связанных с функционированием конкретного устройства и т.п.).

Так в некоторых реализациях системы возможно отсутствие точек входа чтения и записи для блок-ориентированных устройств. В этом случае блок-ориентированный обмен реализуются путем передачи управления на точу **bstrategy().**

В системе возможно обращение к функциям драйвера в следующих ситуациях:

1. старт системы, определение ядром состава доступных устройств.
2. обработка запроса ввода/вывода (запрос может быть инициирован, любыми процессами, в том числе и ядром);
3. обработка прерывания, связанного с данным устройством, в этом случае ядро вызывает специальную функцию драйвера;
4. выполнение специальных команд управления (например, остановка устройства, приведение устройства в некоторое начальное состояние и т.п.).

Существует два, традиционных способа включения драйверов новых устройств в систему:

- **путем «жесткого», статического встраивания** драйвера в код ядра, требующего перекомпиляцию исходных текстов ядра или пересборку объектных модулей ядра.
- **за счет динамического включения** драйвера в систему.

Динамическое включение драйверов в систему предполагает выполнение следующей последовательности действий:

- загрузка и динамическое связывание драйвера с кодом ядра (выполняется специальным загрузчиком);
- инициализация драйвера и соответствующего ему устройства (создание специальных структур данных драйвера,
  формирование данных коммутатора устройства, связывание обработчика прерываний ядра с данным драйвером).

Для обеспечения динамического включения/выключения драйверов предоставляется набор системных вызовов, обеспечивающий установку и удаление драйверов в систему.

Билет 56. Внешние устройства в ОС UNIX. Системная организация обмена с файлами. Буферизация обменов с блокоориентированными устройствами.

На практике, наиболее часто мы имеем дело с обменами, связанными с доступом к содержимому обыкновенных файлов. Рассмотрим обобщенную схему организации обмена данными с файлами, т.е. внутреннюю организацию программ и данных, обеспечивающих доступ к содержимому файловой системы (файловая система может быть создана исключительно на блок-ориентированных устройствах).
Рассмотрим ряд информационных структур и таблиц, используемых системой для организации интерфейса работы с файлами.

Для организации интерфейса работы с файлами ОС использует информационные структуры и таблицы двух типов:
- ассоциированные с процессом;
- ассоциированные с ядром операционной системой.

Таблица индексных дескрипторов открытых файлов.

Для каждого открытого в рамках системы файла формируется запись в таблице ТИДОФ, содержащая:

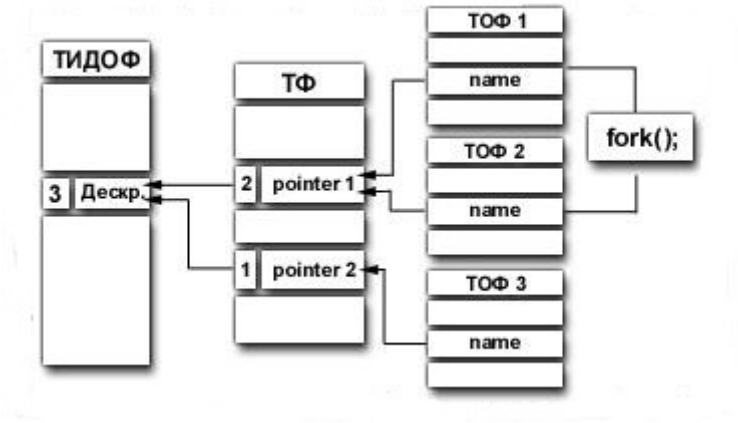
- копия индексного дескриптора (ИД) открытого файла;
- кратность - счетчик открытых в системе файлов, связанных с данным ИД.

Вся работа с содержимым открытых файлов происходит посредством использования копии ИД, размещенной в таблице ТИДОФ. Данная *таблица размещается в памяти ядра ОС*. Если один и тот же файл открыт неоднократно, то запись в ТИДОФ создается одна, но каждое дополнительное открытие этого файла увеличивает счетчик на единицу

Таблица файлов.
Таблица файлов содержит сведения о всех файловых дескрипторах открытых в системе файлов. Каждая запись ТФ соответствует открытому в системе файлу или точнее используемому файловому дескриптору (ФД). Каждая запись ТФ содержит указатели чтения/записи из/в файл. Рассмотрим правила установления соответствия между открытыми в процессах файлами и записями ТФ. При каждом новом обращении к функции открытия файла в таблице процессов образуется новая запись, таким образом если неоднократно в одном или нескольких процессах открывается один и тот же файл, то в каждом случае будет определяться свой независимый от других файловый дескриптор, в том числе со своим указателем чтения/записи. Если файловый дескриптор в процессе образуется за счет наследования, то в этом случае новые записи в ТФ не образуются, а происходит увеличение счетчика «наследственности» в записи, соответствующей файлу, открытому в прародителе. *Таблица размещается в памяти ОС.*

Таблица открытых файлов.

С каждым процессом связана таблица открытых файлов (ТОФ). Номер записи в данной таблице есть номер ФД, который может использоваться в процессе. Каждая строка этой таблицы имеет ссылку на соответствующую строку ТФ. Первые три строки этой таблицы используются для файловых дескрипторов стандартных устройств/файлов ввода вывода.
Для иллюстрации работы с данными таблицами рассмотрим следующий **пример**.



Пусть в системе сформирован процесс **№1**, в нем открыт файл с именем **name** (для простоты будем считать, то это единственное открытие файла с данным именем в данный момент времени), в таблице ТОФ№1 этого процесса будет образована соответствующая запись, которая будет ссылаться на запись в ТФ, которая, в свою очередь, ссылается на таблицу ТИДОФ. Счетчик наследственности ТФ и счетчик кратности ТИДОФ будут равны единице.
Далее, формируется процесс №2, который в свою очередь открывает файл с именем **name**, в результате чего в ТФ будет образована новая запись, которая будет ссылаться на запись ТИДОФ, соответствующую индексному дескриптору файла name, счетчик кратности этой записи увеличится на единицу.
Процесс №1 выполняет системный вызов **fork()** в результате чего образуется процесс №3 с открытым (унаследованным) файлом **name**. В таблице ТОФ№3 будет размещена копия таблицы ТОФ№2, счетчик наследственности соответствующей записи ТФ и счетчик кратности в записи ТИДОФ увеличатся на единицу.

Буферизация при блок-ориентированном обмене

Особенностью работы с блок-ориентированными устройствами является **возможность организации буферизации при обмене**. Суть заключается в следующем. В RAM организуется пул буферов, где каждый буфер имеет размер в один блок. Каждый из этих блоков может быть ассоциирован с драйвером одного из физических блок-ориентированных устройств.
«+» оптимизация и минимизация обмена с реальными устройствами.
«-» Существенная критичность к несанкционированному выключению машины
«-»проблемы разорванности во времени операции записи (поработал, ушел, а данные еще не записались.)
Рассмотрим, как выполняется последовательность действий при исполнении заказа на чтение блока. Будем считать, что поступил заказ на чтение N-ого блока из устройства с номером M.
1. Среди буферов буферного пула осуществляется поиск заданного блока, т.е. если обнаружен буфер, содержащий N-ый блок M-ого устройства, то фиксируем номер этого буфера. В этом случае, обращение к реальному физическому

устройству не происходит, а операция чтения информации является представлением информации из найденного буфера. Переходим на шаг 4.
2. Если поиск заданного буфера неудачен, то в буферном пуле осуществляется поиск буфера для чтения и размещения данного блока. Если есть свободный буфер (реально, эта ситуация возможна только при старте системы), то фиксируем его номер и переходим к шагу 3. Если свободного буфера не нашли, то мы выбираем буфер, к которому не было обращений самое долгое время. В случае если в буфере имеется установленный признак произведенной записи информации в буфер, то происходит реальная запись размещенного в буфере блока на физической устройство. Затем фиксируем его номер и также переходим к пункту 3.
3. Осуществляется чтение N-ого блока устройства M в найденный буфер.
4. Происходит обнуление счетчика времени в данном буфере и увеличение на единицу счетчиков в других буферах.
5.Передаем в качестве результата чтения содержимое данного буфера.

Вы видите, что здесь есть оптимизация, связанная с минимизацией реальных обращений к физическому устройству. Это достаточно полезно при работе системы. Запись блоков осуществляется по аналогичной схеме. Таким образом, организована буферизация при низкоуровневом вводе/выводе. Преимущества очевидны. *Недостатком является то*, что система в этом случае является критичной к несанкционированным отключениям питания, т. е. ситуация, когда буфера системы не выгружены, а происходит нештатное прекращение выполнения программ операционной системы, что может привести к потере информации.
Второй недостаток заключается в том, что за счет буферизации разорваны во времени факт обращения к системе за обменом и реальный обмен. Этот недостаток проявляется в случае, если при реальном физическом обмене происходит сбой. Т. е. необходимо, предположим, записать блок, он записывается в буфер, и получен ответ от системы, что обмен закончился успешно, но когда система реально запишет этот блок на ВЗУ, неизвестно. При этом может возникнуть нештатная ситуация, связанная с тем, что запись может не пройти, предположим, из-за дефектов носителя. Получается ситуация, при которой обращение к системе за функцией обмена для процесса прошло успешно (процесс получил ответ, что все записано), а, на самом деле, обмен не прошел.
Таким образом, эта система рассчитана на надежную аппаратуру и на корректные профессиональные условия эксплуатации.
Для борьбы с вероятностью потери информации при появлении нештатных ситуаций, система достаточно «умна», и действует верно.
А именно, *в системе имеется некоторый параметр, который может оперативно меняться, который определяет периоды времени, через которые осуществляется сброс системных данных*.
Второе - *имеется команда, которая может быть доступна пользователю*, - команда SYNC. По этой команде осуществляется сброс данных на диск.
И третье - *система обладает некоторой избыточностью*, позволяющей в случае потери информации, произвести набор действий, которые информацию восстановят или спорные блоки, которые не удалось идентифицировать по принадлежности к файлу, будут записаны в определенное место файловой системы. В этом месте их можно попытаться проанализировать и восстановить вручную, либо что-то потерять.

Билет 57. Управление оперативной памятью

Основные задачи:

1.**Контроль состояния каждой единицы памяти (свободна/распределена).** Система должна обладать информацией о том, какая единица памяти свободна, какая занята, кем и почему. Соответственно эта функция совместно обеспечивается как аппаратурой компьютера, так и программным обеспечением ОС. ОС создает для этих целей специальные таблицы.

2.**Стратегия распределения памяти**. Надо выбрать правила, по которым принимать решения : когда кому и сколько памяти должно быть выделено.

3.**Выделение памяти**. Принятие решения о выделении конкретного объема памяти для потребителя.

4.**Стратегия освобождения памяти (**процесс освобождает, ОС "забирает" окончательно или временно). Одна из самых важных функций. Выбор стратегии, на основании которой система принимает решения о том, что память надо отобрать на время (при появлении более приоритетного процесса) или навсегда.

Поговорим о стратегиях и методах управления. Рассмотрим целый сектор различных методов. Некоторые старые стратегии управления памятью используются и сейчас, например в мобильных телефонах.

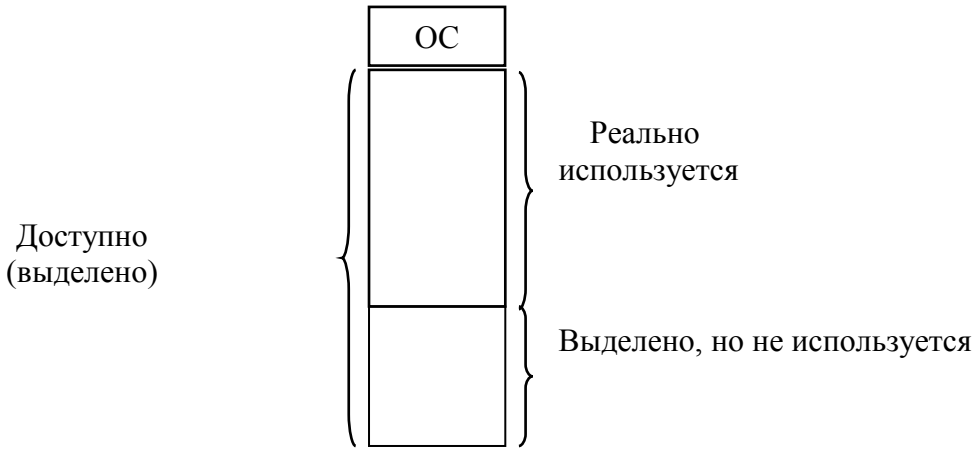
Стратегии и методы управления:

Одиночное непрерывное распределение.
Распределение разделами.
Распределение перемещаемыми разделами.
Страничное распределение.
Сегментное распределение.
Сегменто-страничное распределение.

План рассмотрения стратегий управления:

Основные концепции.
Необходимые аппаратные средства. (необходимое аппаратное обеспечение)
Основные алгоритмы.
Достоинства, недостатки.

Одиночное непрерывное распределение



ОП делится на 2 области. В одной находится ОС, другая предназначена для задач пользователя. (предполагается однопроцессная система.)

Необходимые аппаратные средства:

Регистр границ + режим ОС / режим пользователя. (В регистре границ находится граница между ОС и пользовательской частью ОП)

Если ЦП в режиме пользователя попытается обратиться в область ОС, то возникает прерывание.

В режиме ОС мы можем обращаться в любую точку ОП, если мы находимся в пользовательском режиме, то запрошенный адрес сравнивается с содержимым регистра границ, и , если он окажется меньше, т.е. Мы хотим обратиться в часть ОП, занятую под ОС, то выдается прерывание.

Алгоритм – процесс заканчивается, мы меняем на следующий.

Достоинства: простота.

Недостатки:

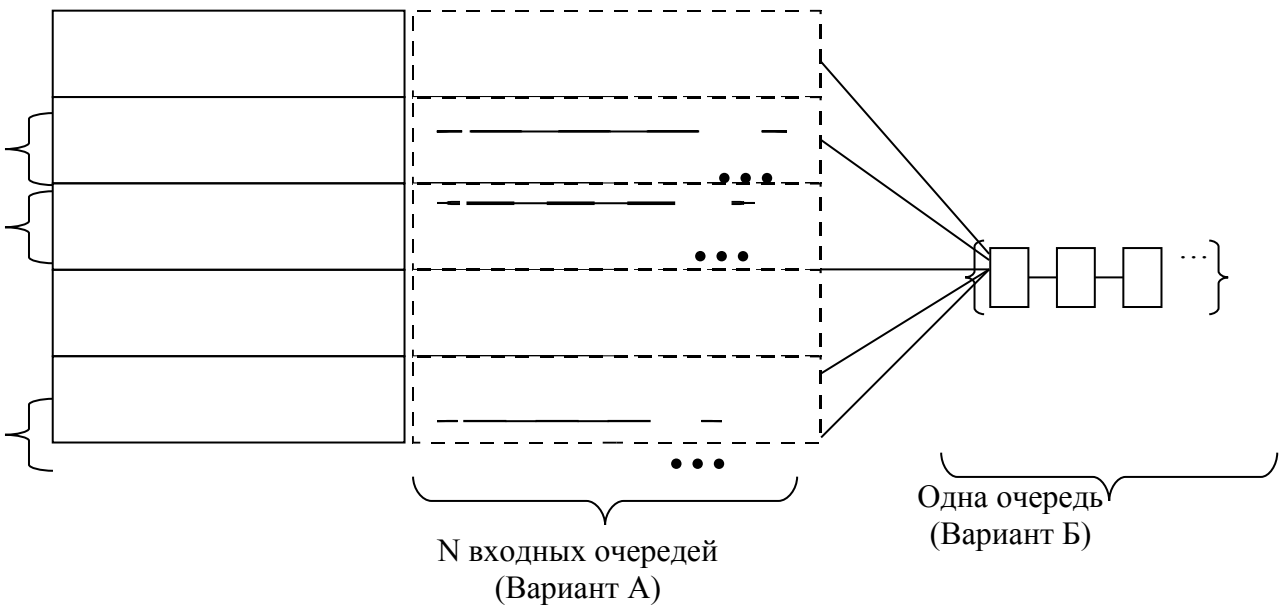
Они следуют из организации.

1.Часть памяти просто не используется. (Внешняя фрагментация)

2.Процессом/заданием память занимается все время выполнения. Внутренняя фрагментация заключается в том, что вся область памяти, которую процесс занимает, занимается процессом на всё время его выполнения. Это означает, что достаточно большие области памяти, которые заняты процессом, не используются., т.к. обычно управление достаточно локализовано. Т.е. неэффективность работы с памятью.

3.Ограничение на размеры процесса. Т.е. загрузить в эту систему процесс, превосходящий область памяти, мы не можем.

Распределение неперемещаемыми разделами



Суть: Есть ОС и оставшаяся физическая память. Оставшуюся физическую память делим на конкретное количество разделов. В каждом может быть свое задание и свой процесс. Внутри каждого раздела все аналогично рассмотренному выше примеру.

Необходимые аппаратные средства:

Необходимо наличие 2 регистров границ, т.к. необходимо обеспечить корректность как по отношению к другим пользовательским разделам, так и по отношению к ОС.
Недостатки:
а. перегрузка регистра границ при каждой смене контекста;
б. сложности при использовании каналов/процессоров ввода/вывода. Если процесс попытается читать не из своей области, то это тяжело отловить
1.Ключи защиты (PSW). Каждый раздел имеет свой ключ защиты, который проверяется при всех операциях чтения\записи . Это решает проблему б).
Алгоритмы: *Модель статического определения разделов*
. Сортировка входной очереди процессов по отдельным очередям к разделам. Вся очередь процессов разбивается на к очередей, с каждой из которых связан свой раздел. Процесс размещается в разделе минимального размера, достаточного для размещения данного процесса. В случае отсутствия процессов в каких-то под очередях – неэффективность использования памяти.
Недостаток: Может возникнуть ситуация, когда очередь больших процессов пуста, а в очереди маленьких процессов очень много процессов. А перегрузить мы не сможем.

Алгоритмы: *Модель статического определения разделов*
Б. Одна входная очередь процессов.
1. Освобождение раздела поиск (в начале очереди) первого процесса, который может разместиться в разделе.
Проблема: большие разделы маленькие процессы. Это несправедливо по отношению к большим процессам.

2. Освобождение раздела поиск процесса максимального размера, не превосходящего размер раздела.
Проблема: дискриминация “маленьких” процессов.

3. Оптимизация варианта 2. Каждый процесс имеет счетчик дискриминации. Если значение счетчика процесса ³ K, то обход его в очереди невозможен.

Достоинства:
Простое средство организации мультипрограммирования.
Простые средства аппаратной поддержки.
Простые алгоритмы.
Недостатки:
Внешняя Фрагментация.
Ограничение размерами физической памяти как внутри одного раздела, так и в целом

Весь процесс размещается в памяти – возможно неэффективное использование и внутренняя фрагментация.

Распределение перемещаемыми разделами

Система имеет фиксированное количество разделов. Через некоторое время ее использования начинается внешняя фрагментация.

Решение: перемещение разделов и освобождение одного большого куска. Но это требует очень больших затрат.

Необходимые аппаратные средства:

1.Регистры границ + регистр базы

2.Ключи + регистр базы

Алгоритмы: Аналогично предыдущему

Достоинства:

Потенциальная ликвидация внешней фрагментации

Недостатки:

Внутренняя фрагментация

Ограничение размером физической памяти

Затраты на перекомпоновку. Операция освобождения одного большого куска ОП очень тяжела.

Билет 58 Управление оперативной памятью. Страничное распределение.
Страничное распределение

Посредством аппаратных и программных решений, например, таблицы страниц, возможно отображать физические страницы. Содержимое таблицы определяет соответствие виртуальной памяти физической для выполняющейся в данный момент программы/процесса. Соответствие определяется следующим образом: i-я строка таблицы соответствует i-й виртуальной странице.
При замене процесса таблицу надо менять.
Таблица страниц – отображение номеров виртуальных страниц на номера физических.

Проблемы:

1. Размер таблицы страниц (количество 4кб страниц при 32-х разрядной адресации – 1000000. Таблица должна иметь миллион строк, а таблицу надо перегружать каждый раз при смене контекстов Любой процесс имеет собственную таблицу страниц).
2. Скорость отображения. Эта проблема, фактически следует из проблемы 1.

Возможные аппаратные средства:

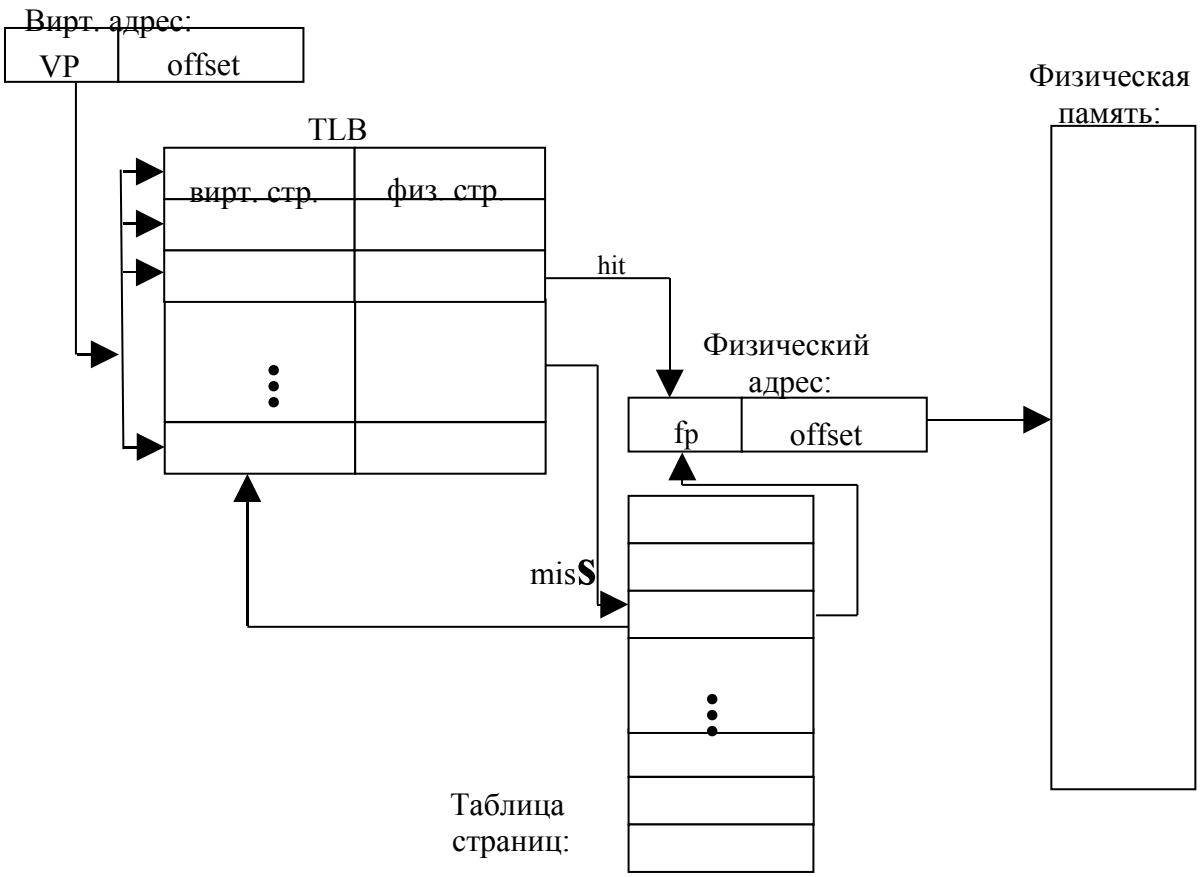
1.Полностью аппаратная таблица страниц, которая будет находится в виде сверхоперативной памяти. Все преобразования будут проходить очень быстро (Проблемы :стоимость, полная перегрузка при смене контекстов, +: скорость преобразования).
2.Регистр начала таблицы страниц в памяти. Будет многократное увеличение количества обращений к памяти. (простота, управление смены контекстов, медленное преобразование). Альтернативное решение - организация таблицы страниц на ОП. В этом случае нам нужен аппаратный регистр начала таблицы, и переключение с контекста на контекст будет осуществляться очень хорошо и быстро, просто я буду менять содержимое регистра начала таблицы. При этом мы получим многократное увеличение количества обращений в память. И как минимум мы потеряем 100% эффективность. Понятно что, часть проблем будут минимизированы за счет работы КЭШ, но все равно это будет неэффективно. Но зато это просто и дешево.

3.Гибридные решения. Т.е. Те, которые имеют и программную и аппаратную составляющую.

Решение проблем, связанных с размерами таблицы страниц – иерархическая организация таблицы страниц.
Предположим, что таблицы страниц индексируются по номерам соответствующих виртуальных страниц. Содержимое каждой записи – информация о соответствующей виртуальной странице.
Поля:
α – присутствие/отсутствие. Если этот признак установлен, то это означает, что в поле «номер физической станицы» находится та самая физическая страница, к которой мы обращаемся. Если отсутствует, то возможны 2 варианта: либо эта

страничка запрещена для данного процесса, либо она разрешена, но сама страница в это время откачена во внешнюю память. Но в любом случае, если есть элемент отсутствия, то при обращении к этой строчке происходит прерывание.
β – поле защиты (чтение, чтение/запись, выполнение). Когда процессор доходит до таблицы страниц, он уже знает, с какой целью он получает этот адрес. Либо этот адрес есть операнд, куда мы хотим записать, либо этот адрес есть операнд, из которого мы хотим считать информацию, либо этот адрес есть операнд команды, которую я хочу выбрать и выполнить (goto адрес). Соответственно это поле обеспечивает защиту. Т.е. в зависимости от того, с какой целью процессор обращается к этой строчке, и содержимого этой строчки (а содержимое могут быть коды, которые разрешают чтение, или чтение/запись, или выполнение, или запрещают их – так же как в ФС), то при нарушении происходит прерывание.
γ – признак изменения (модификации). Если мы в эту страничку писали, то этот признак будет установлен. Этот признак устанавливается обычно аппаратно – автоматически. Снимается он либо аппаратно, либо программно ОС.
δ – обращение (чтение, запись, выполнение). Когда мы обратились либо за чтением, либо за записью и т.д.
ε – признак блокировки кэширования. Я заказал обмен: прочесть информацию с внешнего устройства на какую-то страницу, в конечном итоге физическую станицу. А на самом деле N страниц у меня находится в КЭШе. Как разрешить эту коллизию? Внешнее устройство кинет информацию в физическую память, а на самом деле я работаю с КЭШем, а потом из КЭШа я это переобновлю и все потеряется. Для того, чтобы можно было синхронизовать эту вещь, используется блокировка кэширования. Здесь, кроме управления оперативной памятью в контексте того, о чем говорим, мы еще добавляем некоторую информацию и в темы, связанные с управлением вводом/выводом и в темы, связанные с кэшированием.
Для разрешения всех коллизий, связанных со скоростью, размерами и прочим, используются гибридные решения. И, в частности, одно из решений основывается на TLB буферах.

TLB (Translation Lookaside Buffer) – Буфер быстрого преобразования адресов. В процессоре есть буфер (не большой), который используется в качестве КЭШ таблицы страниц.
Структура буфера: Каждая запись содержит 2 поля - №виртуальной страницы и № физической страницы. TLB буфер – буфер оперативной памяти. Поиск идет параллельно: за одну операцию просматривается наличие всей таблицы.
Мы мимеем виртуальный адрес, в котором традиционно есть поле: «виртуальная страница» и есть поле «смещение». Процессор выбирает поле «виртуальная страница» и обращается к TLB буферу. Если мы фиксируем факт попадания, то в этом случае автоматически происходит замена поля виртуальной страницы на содержимое поля физической страницы – так мы получили физический адрес со всеми вытекающими параметрами, которые могут находиться в TLB. Если мы фиксируем промах, то в этом случае происходит прерывание, управление передается ОС. И ОС уже программно находит необходимую строчку и обновляет TLB буфер и соответственно дообрабатывает команду преобразования виртуального в физический.



Иерархическая организация таблицы страниц

Проблема – размер таблицы страниц.

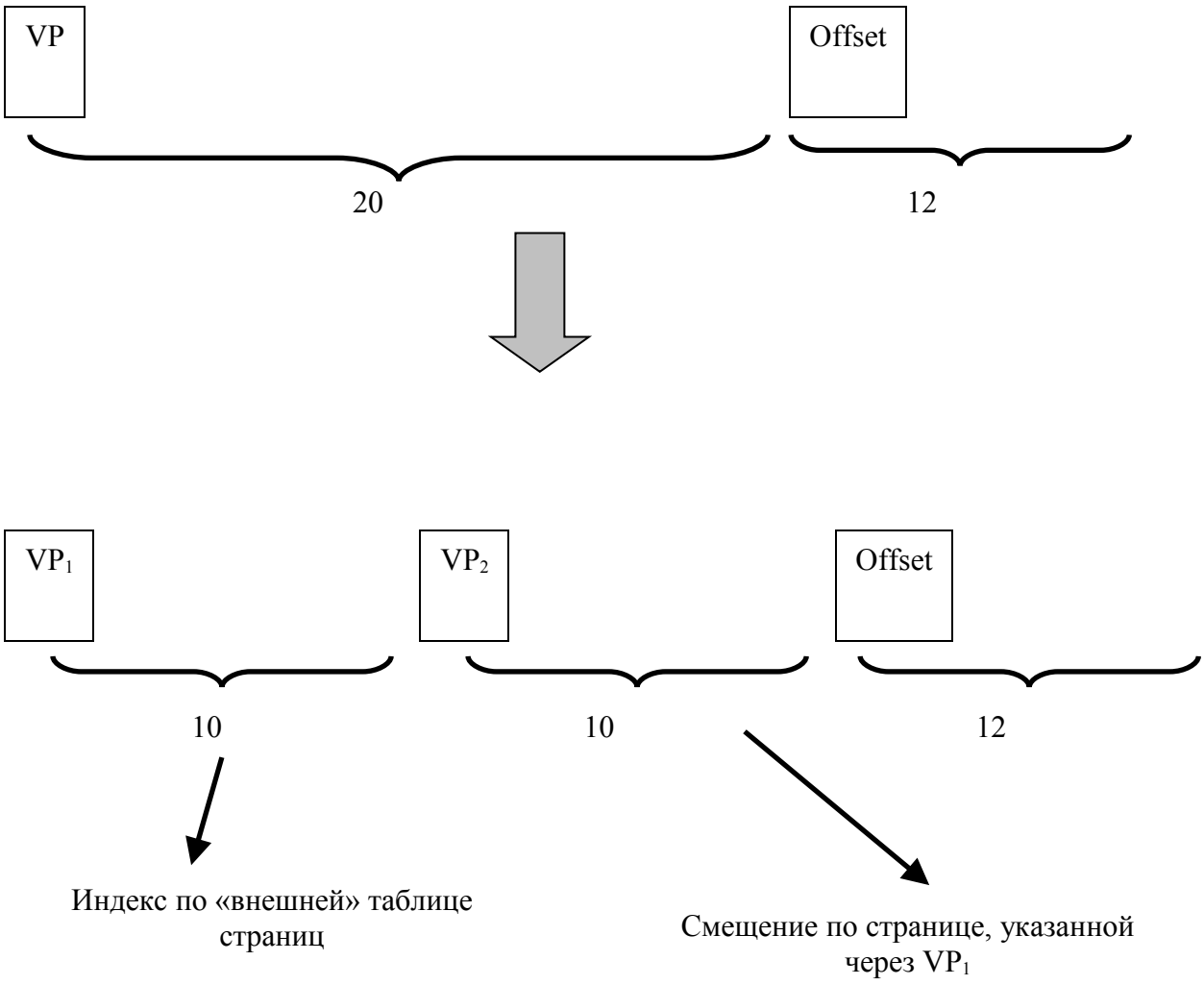
Объем виртуальной памяти современногокомпьютера - 2^{32},…2^{64}
Пример: $\mathbf{V_{вирт.}} = 2^{32}$

$$\mathbf{V_{стр.}} = 2^{12} \quad (4Kb)$$

Количество виртуальных страниц – 2^{20} (много)

Решение – использование многоуровневых таблиц страниц (2^x, 3^x, 4^x)
Современные системы используют многоуровневую организацию таблицы страниц.

Двухуровневая организация



Система разделяет VP на 2 подполя: VP1 - индекс по внешней таблице страниц, а VP2 – смещение по странице, на которую указывает VP1. >4 уровней иерархии считается не целесообразно.

Многоуровневая организация

Суть многоуровневости достаточно простая: если мы имеем виртуальный адрес следующей структуры (на слайде): смещение 4кб и 20-ти разрядный адрес, то система разделяет поле виртуальной странички на два подполя. 1-е подполе – это индекс по внешней таблице страниц, через этот индекс мы попадаем на страничку, в которой находится продолжение описания этой таблицы; 2-е поле – это смещение

по этой странице. Т.е. мы имеем внешнюю таблицу, по VP1 мы индексируемся и соответственно по содержимому этой таблицы попадаем на некоторую страницу, в которой находится часть таблицы страниц 2-го уровня. И по VP2 мы проходим смещение по этой странице и в соответствующем элементе получаем номер физической страницы. Этих уровней может быть 2, 3, 4. Больше 4-х считается нецелесообразным. Для 64-х разрядных машин таких уровней если их реализоввывать должно быть не менее 7, что совсем нецелесообразно.

Использование хэштаблиц

ХЭШ функции изначально использовались при организации таблицы имен.

ХЭШ – функция берет номер виртуальной страницы и по этому номеру виртуальной страницы имеется некоторая функция, которая определяет номер записи хэш-таблицы. С этой записью связан список виртуальных страниц с их физическими страницами, которые имеют одинаковое значение хэш-функции. Это означает, что при преобразовании мы берем виртуальную страницу и фактически автоматически попадаем на этот самый список. Дальше по этому списку мы можем дойти до искомой страницы и получаем физическую страницу. Если в списке нет, то это означает, что и странички такой нет.

Инвертированные таблицы страниц

Используется в более развитых системах, системах аппаратно поддерживающих pid обрабатываемого процесса.

Каждая строка таблицы соответствует конкретной физической странице.

Проблема – поиск по таблице

Замещение страниц

Проблема загрузки «новой» страницы в память, если свободных мест в памяти нет. Необходимо выбрать страницу для удаления из памяти (с учетом ее модификации пр.)

Алгоритм NRU (Not Recently Used – не использовавшийся в последнее время)

Используются биты статуса страницы. R – обращение, M – модификация. Устанавливаются аппаратно при обращении или модификации.

Алгоритм

1.При запуске процесса M и R для всех страниц процесса обнуляются
2.По таймеру происходит обнуление всех битов R
3.При возникновении страничного прерывания ОС делит все страниц на классы:
•Класс 0: R=0; M=0; - не читался и не изменялся.
•Класс 1: R=0; M=1;
•Класс 2: R=1; M=0;
•Класс 3: R=1; M=1;
4.Случайная выборка страницы для удаления в непустом классе с минимальным номером

Стратегия: лучше выгрузить измененную страницу, к которой не было обращений как минимум в течение 1 «тика» таймера, чем часто используемую страницу

ОС фиксирует время размещения страницы. Наиболее старую страницу удаляем, но это может быть неправильно, т.к. старая может часто использоваться, а новая - редко. Поэтому используется модификация этотого алгоритма. R – бит обращения.
1.Выбирается самая «старая страница». Если R=0, то она заменяется

2.Если R=1, то R – обнуляется, обновляется время загрузки страницы в память (т.е. переносится в конец очереди). На п.1

Алгоритм FIFO

«Первым прибыл – первым удален» - простейший вариант FIFO. Для каждой страничке, которая была помещена в память, ОС фиксирует время ее размещения. Соответственно после этого наиболее старую страницу ОС удаляет. Это не очень справедливо (проблемы «справедливости»). Потому что в этом случае старая страница может активно использоваться и быть удалена. Поэтому реально используются модификации алгоритма FIFO.

*Модификация алгоритма (**алгоритм вторая попытка**):*

1.Выбирается самая «старая страница». Если R=0, то она заменяется

2.Если R=1 (к ней обращения идут), то R – обнуляется, обновляется время загрузки страницы в память (т.е. считается, что она была загружена в момент обнуления признака чтения, т.е. фактически она переносится в конец очереди). На п.1 (начинаем смотреть следующую).

Алгоритм «Часы»

Алгоритм аналогичен предыдущему, только все страницы связаны в кольцевой список. Существует указатель (стрелка) на текущую страницу.

Алгоритм LRU (Least Recently Used – «менее недавно» - наиболее давно используемая страница)

Пусть в памяти N – страниц. Составляется битовая матрица NxN (изначально все биты обнулены). При каждом обращении к iой странице происходит присваивание 1 всем битам iой строки и обнуление всех битов iго столбца. Строка с наименьшим 2ным числом соответствует искомой странице.

Алгоритм NFU (Not Frequently Used – редко использовавшаяся страница)

Развитие предыдущего алгоритма.

Для каждой физической страницы заводится программный счетчик, который изначально обнулен. По таймеру к счетчикам прибавляется признак доступа.

В момент принятия решения выбирается страница с минимальным значением счетчика.

Это все решается программно.

Недостаток – если процесс поработал и «сидит без дела», то удалить его не удастся, а он не работает.

Модификация:

1.Значение счетчика сдвигается на 1 разряд вправо.

2.Значение R добавляется в крайний левый разряд счетчика.

Достоинства страничной памяти:

- нет проблемы внешней фрагментации
- никак не ограничены размерами физической памяти, т.е. мы часть страниц можем всегда держать во вне и через прерывания их закачивать, когда они нам нужны

Недостатки:

- проблема принятие решений об организации таблицы страниц
- при страничной организации памяти адресное пространство представляет одну модель от 0 до N. Т.е. мы работаем с одним пространством адресации в этом процессе. В некоторых ситуациях это бывает не очень удобно.

Билет 59. Управление ОП. Сегментное распределение.

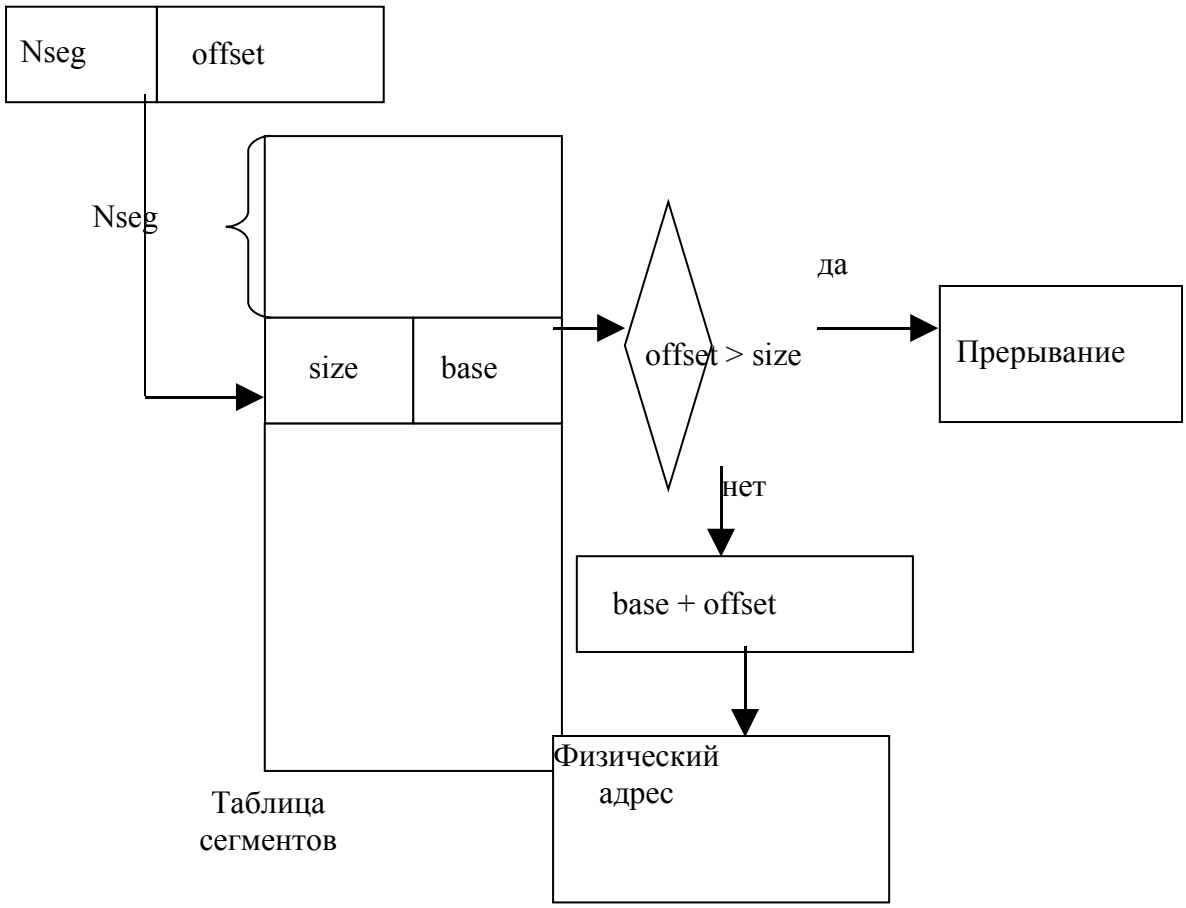
Сегментная организация памяти

Основные концепции:

•Виртуальное адресное пространство представляется в виде совокупности сегментов
•Каждый сегмент имеет свою виртуальную адресацию (от 0 до N-1)
•Виртуальный адрес: <номер_сегмента, смещение>

Необходимые аппаратные средства для организации сегментной памяти достаточно концептуально просты. Это таблица сегментов, по которой при вычислении физического адреса из виртуального мы можем индексироваться по номеру сегмента. Соответственно каждая запись таблицы сегментов содержит размер сегмента и адрес начала сегмента.

«+» простота реализации
«+» размер таблицы сегментов может быть много меньше размера таблицы страниц
«-» наличие внешней фрагментации
«-»сегмент рассматривается как единое целое



Преобразование происходит достаточно просто: мы индексируемся по таблице, получаем запись, после этого сравниваем смещение с размером сегмента:
если смещение выходит за пределы размера – происходит прерывание,
иначе мы значению базы прибавляем смещение и получаем физический адрес.

Упрощенная модель Intel.

Виртуальный адрес содержит 2 поля: **селектор и смещение**. Селектор содержит информацию о **номере сегмента**, о **локализации.**
Поле Локализация это таблицы локальных дескрипторов (сегменты доступные для данного процесса) LDT (Local Descriptor Table) и Таблица глобальных дескрипторов (разделяемые между процессами сегменты) GDT (Global Descriptor Table).
По LDT и GDT и виртуальному адресу мы вычисляем линейный адрес. Линейный адрес представляется в виде двухуровневой страничной организации. По этим параметрам мы вычисляем физический адрес.

Сегменто - страничная организация памяти

Виртуальный адрес содержит 2 поля: селектор и смещение. Селектор содержит информацию о номере сегмента, о локализации (есть 2 таблицы сегментов, который используются – это сегменты, которые доступны только для данного процесса, или сегменты, которые могут использоваться в разных процессах).
При использовании алгоритма сегментно\страничной организации к плюсам страничной прибавляются плюсы сегментной.