Прерывания:

1. Системные вызовы (программные прерывания). Пример: ввод, вывод.
2. Исключения: возникают в процессе выполнения программы
    a. Устранимые - страничное прерывание
    b. Неустранимое - неправильный адрес, деление на 0
3. Аппаратные прерывания
    a. От системного таймера
    b. От устройств ввода/вывода - по завершению операций ввода/ вывода
    c. От действий оператора

Для ввода/вывода используются системные вызовы - иначе система станет уязвимой. Прерывание от устройств ввода/вывода возникает после завершения операции ввода/вывода.

Принципы ЭВМ по Фон-Нейману:

- принцип хранимой программы (располагается в последовательных адресах): записывать в одну и ту же память и команды и данные, обращение выполнять по адресу. Т.о. при считывании команды содержимое счетчика команд инкрементируется на размер команды. Она всегда содержит адрес следующей команды.

1-е поколение: все вручную, не имело ОС.
2-е поколение: транзисторы. Диод(твердотельный переключатель) создан в 48-м году - новая элементная база. Началось производство серийных машин. Операторы эксплуатируют машины. Появилась необходимость написать ОС - она заменяла оператора. Появились интегральные схемы.
3-е поколение: интегральные схемы, появление понятия архитектуры вычислительной машины. В нее заложен принцип распараллеливания функций. Это выполнялось с помощью каналов. Увеличивается эффективность работы процессора, который простаивал, ожидая ответа от внешних устройств. Теперь эту функцию выполняют каналы. В IBM 360,370 канальная архитектура. У современных машин шинная архитектура. Внешними устройствами управляет контроллер. Адаптер располагается на материнской плате. Они получают от процессора команду, после чего управляют устройством. Для информирования процессора о завершении работы используются аппаратные прерывания. Возникает идея виртуальной памяти.
4-е поколение: 70-е годы IBM 370, создание машин PDP(Programmed Data Processor - компьютер). Пишется для этих машин ОС Unix. Ставится вопрос о непосредственной работе с ПО. Появляются терминалы. Идея виртуальной памяти (много программ, нужно много ОП).
Терминал - монитор + клавиатура.
Телетайп - объединение пишущей машинки и вывода ОЦПУ.
Система должна обеспечивать гарантированное время ответа. Оно не может превышать 3 секунды.Чтобы обеспечить это, решили работать с удаленным терминалом. Время стали квантовать. Системы стали называть системами распределения времени. Время выделяется запущенным программам квантами.
Была система MULTIX - система разделения времени. Язык Си, написали сами для себя. (Си - язык системного, профессионального программирования.) Ядро этой системы было открытым.
BSD(Berkeley Software Distribution)
Университет Беркли начал разрабатывать(дорабатывать) ядро - Unix BSD, которую можно было поставить на 386 машину. Возникла система System 5, MacOS и BSD. В результате развития ОС с открытым ядром появились 2 коммерчески значимые.
Отсутствовала переносимость (на system 5, MacOS).
Программисты выступили инициаторами создания стандарта POSIX - перечислены до 1000 системных вызовов, которые должна поддерживать ОС Unix для переносимости программ. Этому помог институт стандартов.
Так, учреждения могли покупать ОС только по стандарту POSIX. Это послужило стимулом для широкого распространения стандарта.

Классификация ОС

- однопрограммной пакетной обработки
- мультипрограммной пакетной обработки
- системы разделения времени
- системы реального времени

DOS - однопрограммная интерактивная ОС.

Система реального времени (реальное время - техническое понятие)
Ответ вычислительной системы должен формироваться за заданный интервал времени. Он меняется в зависимости от того, чем управляет ОС - внешним объектом, внешней системой, внешним процессом. Обычно системы реального времени - это системы специального назначения.
Пример: автопилот самолета - сложная система из подсистем (критических и не очень), состоит из датчиков. В зависимости от важности подсистемы определяется время ответа подсистемы. Подача топлива - критическая, температура - не очень. Программа подает команды на соответствующие устройства.

Системное ПО:
- ОС и утилиты ОС
- системы программирования (трансляторы, редакторы связей, загрузчики)

Управления процессорами
Компьютер - программно управляемое устройство, причем часть времени его работой управляет ОС, а часть времени - исполняемая программа.
ОС (Оксфордский словарь по вычислительной технике) - комплект программ, которые совместно управляют ресурсами вычислительной системы и процессами, использующими эти ресурсы при вычислениях (профессиональный взгляд на ОС). //для пользователя ОС - интерфейс//
Ресурс - любой из компонентов вычислительной системы и предоставляемые ею возможностей:
Ресурсы:
1) процессорное время
2) Объем ОП
3) устройства ввода/вывода
4) таймер
5) данные
6) ключи защиты
7) реентерабельные коды ОС(код чистой процедуры) - это код или программа, которая не модифицирует саму себя, при этом множество процессов могут одновременно использовать эту процедуру, при этом находится в разных ее точках. - код чистой процедуры - из чистых процедур выносятся все переменные (в ядре вся информация находится в системных таблицах). Unix код всегда реентерабельный.

Иерархическая и виртуальные машины
Иерархическая машина (Медника-Донавана) построена относительно процессов.
Процесс - программа в стадии выполнения.
Была картинка - теперь просто текст - иерархия (1 - самый низ):
1. Аппаратура
2. Управление процессами(нижний уровень)
3. Планирование процессов и управление памятью
4. Управление процессами(верхний уровень) - создание и удаление процессов, обмен сообщениями
5. Управление устройствами
6. Управление данными (файловая подсистема)

Диспетчер выполняет задачу распределение времени на процессы.
Планировщик отвечает за “очередь” процессов.
Процесс выполняется только тогда, когда он получает процессорное время(диспетчер). Разработана иерархическая структура для возможности разработки ОС несколькими разработчиками.

Интерфейсы:
- непрозрачный - можно обращаться только на низлежащий уровень
- полупрозрачный - можно обращаться через уровень (на некоторые)
- прозрачный - можно обращаться с любого на любой уровень

Каждый уровень предоставляет последующему некоторую виртуальную машину.

Особенности разработки ОС

1) Определение абстракций - основных понятий, которыми будет оперировать ОС. К абстракции относятся процессы, потоки, файлы, семафоры и т.д.
2) Предоставляемые примитивные операции. Поскольку абстракции в системе предоставляются конкретными структурами данных, то для управления для управления этими структурами необходимо предоставить в распоряжение пользователя соответствующие примитивные операции: создание, уничтожение, изменение. Например, для файлов: чтение, запись, создание, удаление. Примитивные(низкоуровневые) операции реализуются в виде системных вызовов (API функции).
3) защита системы от пользователей, пользователей от друг друга и обеспечение совместного использование данных и ресурсов системы, изоляция отказов
4) управление аппаратурой

Сложность проектирования ОС обуславливается:

1. большой объем кода (unix > 1 млн строк, windows 2000 - 29 млн строк)
2. подсистемы ОС взаимодействуют друг с другом
3. параллелизм - параллельное выполнение
4. хакеры
5. совместное использование данных и ресурсов системы
6. высокая степень универсальности
7. переносимость - на разных платформах
8. совместимость с предыдущей версией для ПО

Структура ядра ОС Unix BSD

<table>
<tr><td colspan="5"></td><td></td><td></td><td colspan="2"></td></tr>
<tr><td colspan="2">управление терминалом</td><td colspan="2">(3)
3</td><td>3</td><td>(4)??</td><td>сокеты(2)</td><td rowspan="2">обработка сигналов</td><td rowspan="2">создание и удаление процессов</td></tr>
<tr><td rowspan="2">необработанный телетайп</td><td>обработанный телетайп</td><td colspan="2">файловая система</td><td colspan="2">виртуальная память</td><td>сетевые протоколы</td></tr>
<tr><td>дисциплины связи</td><td colspan="2">буферный кэш(1)</td><td colspan="2">страничный кэш(7)</td><td>маршрутизация</td><td colspan="2">планирование процессов(6)</td></tr>
<tr><td colspan="5">Драйверы</td><td rowspan="2">сетевых устройств</td><td colspan="2" rowspan="2">диспетчеризация процессов(5)</td></tr>
<tr><td colspan="3">символьных устройств
(мышь, клавиатура)</td><td colspan="2">блочных устройств
(диски)</td></tr>
</table>

(1) Буферный кэш предназначен для передачи данных(работа с дисками)
(2) привязаны к сетевым устройствам - универсальное средство взаимодействия процессов
(3) символьный уровень - именование пользователем файлов
(4) отображение данных, страничные прерывания????
(5) выделение процессорного времени процессу
(6) в какой последовательности выполняются процессы
(7) виртуальная память

Диаграмма состояний процесса (общая)

Диаграмма состояний процесса (это основная абстракция современных ОС). Процесс является единицей декомпозиции системы, ему выделяются ресурсы.
Этапы жизненного цикла процесса:

1) Создание(порождение процесса). - это значит его идентифицировать. Процесс описывается структурой, содержащей информацию, необходимую системе для управления процессом. В системе существует 1 главная таблица процессов. Когда он создается, говорят, что ему выделяется строка в таблице процессов.

Номер строки является идентификатором процесса. Процессы Unix идентифицируются целым числом. Далее система вызывает менеджер памяти, процессу выделяется физическая память (в соответствии с архитектурой Неймана). Процессор выполняет только тот процесс, который хранится в памяти (принцип хранимой программы). В результате процесс получает все необходимые ресурсы, чтобы он мог начать выполнение и переходит в состояние готовности(2).

2) Готовность. В мультипроцессных ОС в памяти одновременно находится большое количество процессов, т.е. находятся в состоянии готовности. Они образуют очередь процессов. В состоянии готовности все процессы имеют все необходимые ресурсы для их выполнения, кроме последнего - процессорного времени. Когда процесс получает процессорное время, оно переходит в стадию выполнения(3).
3) Выполнение. Из этого состояния процесс может перейти в 2 состояния - состояние завершения(4) и блокировки(5)(sleep в Unix).
4) Завершения (освобождение ресурсов)
5) Блокировка (ожидание/сон sleep) Из состояния блокировки процесс возвращается в очередь готовых процессов (3) (после получения дополнительных ресурсов).

Планирование - постановления процессов в очередь.
Диспетчеризация - выделение процессорного времени.

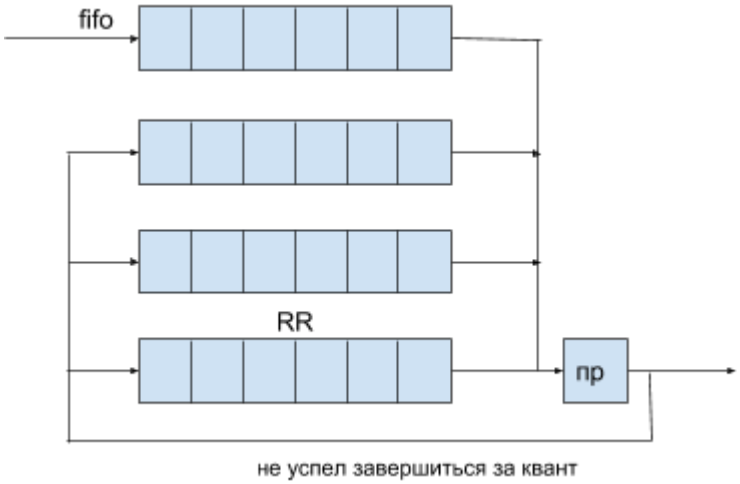
Алгоритмы планирования в системах пакетной обработки.

1. FIFO. Выполняется от начала до конца. В очередь добавляются по времени создания.
2. SJF - Shortest Job First - указывалось максимальное время выполнения и максимальный объем памяти. Преимущество отдается заданиям с минимальным временем (для того, чтобы было выполнено больше заданий). Недостаток - постоянно откладываются задания с большим временем работы - проблема бесконечного откладывания.
3. SRT - Shortest Remaining Time - наименьшее оставшееся время - выполняющийся процесс мог быть вытесненным, если поступил процесс с меньшим оценочным временем выполнения, чем время, которое осталось процессу до его завершения. Оценочное время процессов уменьшается на время выполнения. Проблема бесконечного откладывания также остается.
4. HRRN - Highest Response Ratio Next (наибольшее относительное время ответа). Приоритет вычисляется по формуле p = (tw + ts) / ts, где tw - время ожидания, ts - запрошенное время.

Алгоритмы планирования в интерактивных системах и системах разделения времени

1. RR (циклический алгоритм планирования)

2. Адаптивный алгоритм (многоуровневые очереди). В первую очередь попадают новые процессы или с устройств ввода-вывода. Для 1-й очереди величина кванта выбирается таким образом, чтобы большинство процессов успела завершиться или выдать запрос на ввод-вывод. Если за выделенный квант процесс не успел завершиться или выдать запрос на вв/в, то он попадает в более низко приоритетную очередь. Если он требует большое число квантов, он оказывается в самой низкой приоритетной очереди, в которой крутиться холостой процесс. В последнюю очередь попадают вычислительные процессы.



Это концептуальная модель алгоритма. В Windows и Linux также учитывается и время ожидания в очереди. Высокоприоритетные процессы ставят в начало очередей. Процессы в состоянии готовности образуют двусвязные списки. В Unix приоритеты вычисляются по формуле: p_usrpri = PUSER + (p_cpu/4) + (2*p_nice), где PUSER — базовый приоритет в режиме задачи, равный 50.
В Windows сканируются 12 процессов, если какой-либо процесс ждет более 2х секунд, то его приоритет повышается.
Если приходит более высокоуровневый процесс - он вытесняет текущий, который отправляется в очередь готовых. На этапе завершения освобождаются ресурсы.

Алгоритмы планирования бывают:

- с переключением и без переключения
- с приоритетами и без приоритетов
- с вытеснением и без вытеснения.

Приоритеты бывают:

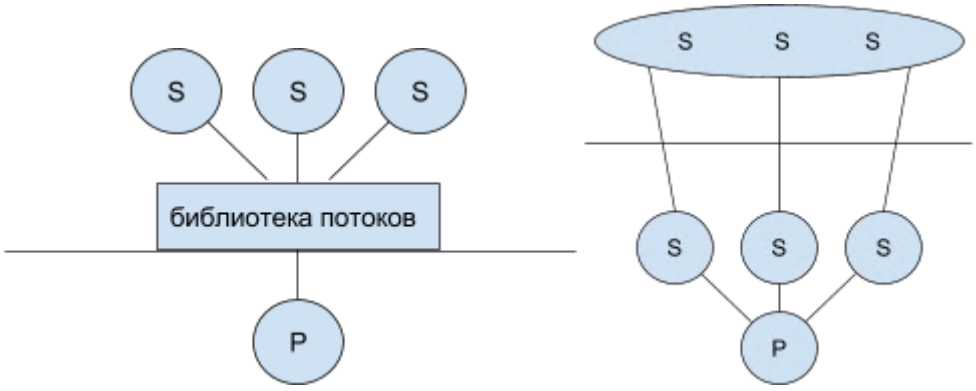
- статические - не изменяются в процессе выполнения (имеют процессы реального времени)
- динамические - изменяются в процессе выполнения (могут иметь прикладные процессы)

Контекст, переключение контекста

Вытеснить\переключить процесс - дело непростое. Система должна запоминать информацию о выполнении процесса для продолжения(возобновления) выполнения работы процесса. Эта информация называется **контекстом процесса**.
Контекст бывает **аппаратный** (содержимое регистров процессора, поддерживается аппаратно PUSHA (в стек ядра сохраняется аппаратный контекст) и POPA) и **полный** (аппаратный + информация о выделении процессу ресурсов (память, семафоры, разделяемая память, сокеты) Память описывается в специальных таблицах. в Unix - карты трансляции адресов.
При переключении процессов сохраняется полный контекст. В интерактивных системах очень много переключений. На переключения контекста требуется много времени. Чтобы сократить время придумали потоки.

Потоки

Поток - это часть кода программы , которая может выполняться параллельно с другими частями кода программы. У потока нет собственного адресного пространства. Он выполняется в адресном пространстве процесса. Владельцем ресурсов остается процесс. Есть потоки уровня пользователя и ядра.
П**отоки уровня пользователя** (о них системе ничего не известно. Для их управления нужна специальная библиотека уровня пользователя.)
Библиотеки предоставляют функции: создания, удаления, обмена сообщениями, функции планирования и переключения контекста. При запросе сервиса системы блокируется весь процесс.



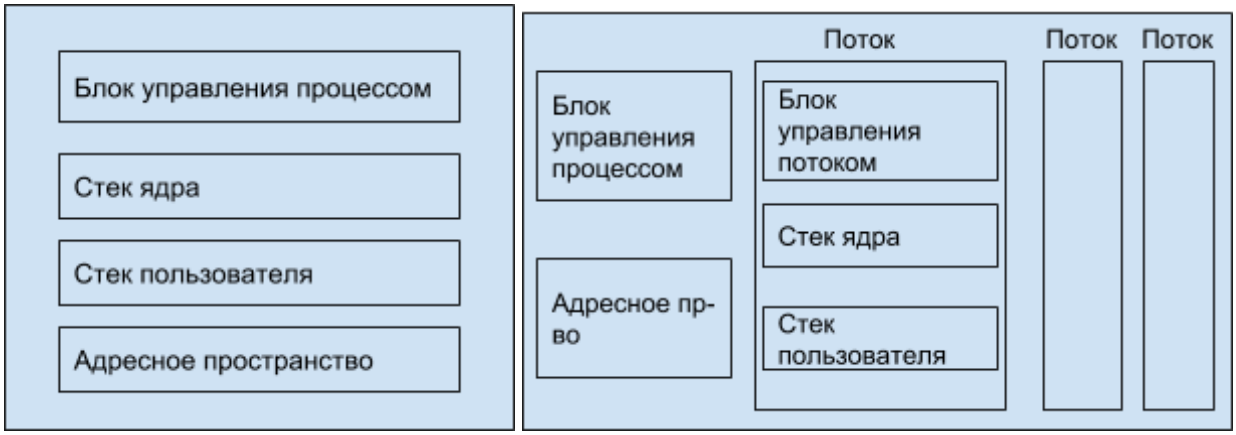
Потоки уровня ядра. Потоки уровня ядра описаны в системе. Ядро знает о них. Единицей диспетчеризации становится поток. Поток имеет аппаратный контекст, т.е. владеет счетчиком команд, т.е. процессорное время выделяется потоку.

Когда переключаются потоки одного процесса, переключается только аппаратный контекст, а когда разных - то полный контекст. Система не учитывает этого. Потоки выстраиваются по приоритетам процесса. В наших системах есть потоки ядра. В структуре, описывающей процесс, есть структура, описывающая поток.
Процессорное время (квант) выделяется потоку. Система не контролирует, поток какого процесса выполняется следующим. Процесс является владельцем ресурсов, а поток только запрашивает ресурсы. Поток имеет приоритет процесса. Поток можно определять как динамический объект, представимы в процессе точкой входа и последовательностью команд, отсчитывающихся от точки входа.

Переключение контекста и что-то о процессах.
Когда процесс запрашивает ресурсы:
Все типы прерываний переводят систему в режим ядра. Процессы постоянно переключаются в режим ядра и обратно. Часть времени процесс находится в режиме задачи - выполняет свой код, а часть - в режиме ядра - выполняет реентерабельный код ОС.
Пример. программа выполняется команда за командой, обращается к read/write - обращение к внешнему устройству. Система не позволяет обратиться напрямую (иначе система будет беззащитной - это открывает доступ к любым системным данным).
В Unix все файл. Система работает с помощью одних и тех же системных вызовов.
Обращение к устройству выполняется только в режиме ядра. При этом переключается аппаратный контекст. Процесс переводится в состояние блокировки.
Аппаратное прерывание происходит асинхронно. Системные вызовы и исключения - синхронные (возникают в процессе выполнения программы).

Полный контекст - информация о ресурсах, выделенных процессу. Есть дополнительный регистры, в частности, в них записывается начальный адрес таблицы сегментов или таблицы страниц.
Переключение полного контекста - затратное действие.

Однопоточная и многопоточная модель процесса
Однопоточная модель: Дескриптор процесса = блок управления процесса. / **Многопоточная модель**



Под адресным пространством понимается адресное пространство приложения(программы). Так как виртуальное адресное пространство содержит оба пространства: адресное пространство приложения и адресное пространство системы. Именно программа делится на потоки. Для единообразного управления если в процессе не запускаются потоки, то считается, что был запущен 1 глобальный поток.
В наших системах нет отдельного стека прерываний. Чтобы обрабатывать события для перехода процессора в режим ядра любой процесс, любой поток должен иметь стек ядра. Стек пользователя создает пользователь (он может быть не один). Нельзя один стек использовать и как стек ядра и как стек пользователя (проблема - размер , который определит пользователь может быть недостаточен).

Переключение в режим ядра.
Существует 3 типа событий, переключающих систему в режим ядра.

1) системные вызовы или программные прерывания(software interrupts)
2) исключения(exceptions)
3) аппаратные прерывания (interrupts)

Диаграмма состояний процесса в ОС Unix
(какие действия выполняются в режиме ядра, а какие в пользовательском режиме)
Любой процесс может создать любое количество процессов.

Ни одна переменная не может использоваться до ее определения, тип до объявления, функция до описания.
Система должна идентифицировать процесс - дать место в таблице. Любая таблица - массив структур. Если в системе достаточно памяти, то она выделяется процессу, он загружается и выполняется.
В Unix говорят, что процесс часть времени выполняется в режиме задачи и выполняется собственный код, а часть - в режиме ядра, тогда он выполняет реентерабельный код ОС.

Управление памятью
Горизонтальное управление - задача управления конкретным уровнем (диски, оперативная память, регистры(в порядке убывания расстояния) - между ними есть кэши и используют буферизацию).
Вертикальное управление(управление иерархией памяти)(близость к процессору) - передача информации с уровня на уровень.
Адресное пространство диска делится на 2 части:

1. Большая - для управления ФС - файловой подсистемой
2. Меньшая - для swap и paging

Быстродействие ОП более чем на порядок меньше быстродействия ЦПУ. Внешним ЗУ управляет файловая система. Файловая система обеспечивает доступ к хранимой информации. Для сокращения времени доступа используется буферизация.
Иерархия памяти:

1) Внешнее ЗУ
2) Оперативная память
3) КЭШ 1-го уровня
4) Процессор

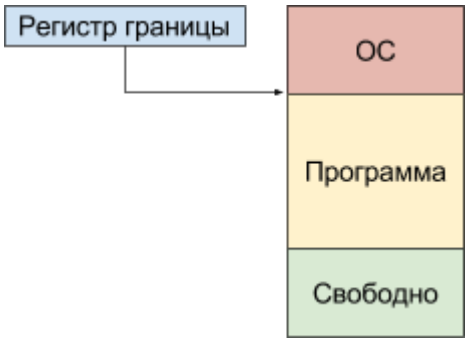
В процессорах intel выделяют кэш команд, данных и TLB - специализированный кэш центрального процессора, используемый для ускорения трансляции адреса виртуальной памяти в адрес физической памяти. Кэш находится в том же кристалле, что и процессор.

По Фон-Нейману Процессор может выполнять только то, что загружено в ОП.

Связанное и несвязанное распределение памяти
Связанное - ситуация в которой программа полностью помещается в память и занимает непрерывное адресное пространство.

ОС однопрограммной пакетной обработки
Все адресное пространство ОЗУ отдано единственной программе (на самом деле программе и ОС).

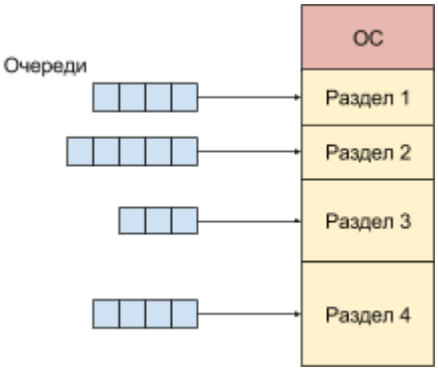
Одиночное непрерывное распределение



Регистр границы находится в процессоре, защищает ОС от приложения. Позже, с увеличением объема оперативной памяти, появились мультипрограммные ОС.

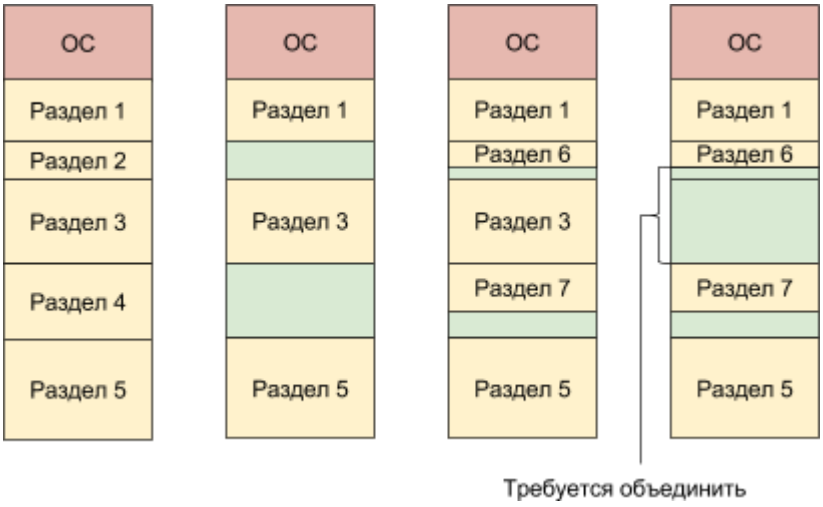
Мультипрограммные ОС
Основная цель – сократить время переключения с одной программы на другую, что повышает производительность системы.

Распределение памяти статическими разделами(исторически первый вариант)



Разделы определяются до начала работы системы на основании собранной статистике о наиболее вероятных размерах программ (малый, средний, большой). К разделам выстраиваются очереди процессов. Возникают ситуации, когда какой-то раздел не используется. Можно создать общую очередь, но при загрузке в большой раздел маленький - остается много памяти.
Проблема в том, что очереди заполняются не равномерно, к большим разделам маленькая очередь, а в другие могут быть большие.
Перешли к распределению динамическими разделами.

Распределение памяти динамическими разделами



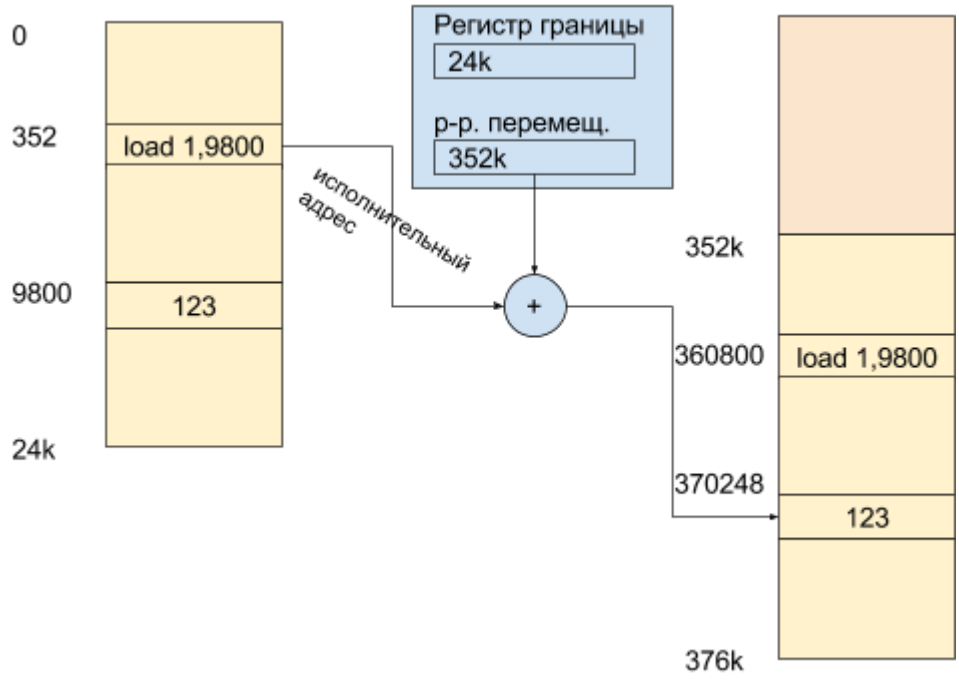
Разделы определяются загрузкой программ. В результате постоянной загрузки и выгрузки возникают небольшие участки памяти объем которых слишком мал для того, чтобы в них что-то загрузить. Это явление называется фрагментацией памяти (до 30%).

Стратегии выбора раздела для загрузки программы

1) Первый подходящий (по размеру)
2) Самый тесный (для загрузки ищется наиболее близкий по размеру)
3) Самый широкий

Для реализации 2 и 3 необходимо упорядочивать разделы по размеру.

Появилась идея логического адресного пространства: любая программа считает, что она начинается с 0 адреса. Это дает возможность загрузки с любого адреса, но для этого нужны дополнительно преобразования.(для перемещения программ).



В программе находятся не адреса, а смещения.
Смещение мы берем из программы. Надо защищать адресное пространство процессов друг от друга - контролировать выход процесса за собственное адресное пространство. В регистре границы записывается размер программы. Регистр перемещения - в котором лежит начальный адрес. Преобразование может выполняется несколько раз за команду!
Для управления памятью нужны 2 таблицы:

Таблица выделенных разделов				Таблица свободных областей			
№	Размер	Адрес	состояние	№	Размер	Адрес	состояние
1	8k	312k	распределен	1	32k	352k	доступен
2	32k	320k	распределен	2	320k	504k	доступен
3	-	-	пуст	3	-	-	пуст
4	120k	384k	распределен	4	-	-	пуст
5	-	-	пуст	5	-	-	пуст

Компоновка - затратное действие по редактированию таблиц. Способы:

1. Сразу после освобождения
2. Не найден свободный раздел нужного раздела (реже перекомпоновка, но дольше!).

Идея несвязанного распределения памяти
Предполагает деление программы на сегменты и т.к. есть таблицы с помощью которых мы можем управлять преобразованиями, то мы можем загружать сегменты в разные разделы.
Размер сегмента = размер кода.
Сегмент - единица логического деления памяти (ничем не отличается от разделов)
Другим способом является распределение памяти **страницами**.
Адресное пространство процесса делится на страницы и адресное пространство физической памяти делится на страницы такого же размера. Нужно иметь возможность обращаться каждому отдельному сегменту/странице с помощью системных таблиц. Нужно описывать адресное пространство каждого процесса.
Разбиение по сегментам - логическое деление. Разбиение по страницам - физическое деление.

Виртуальная память

Виртуальная память - кажущаяся, возможная. Это память, объем которой превосходит объем физического адресного пространства.

Это способ управления памятью, когда часть процесса в ОП, а часть на диске.

Схемы управления виртуальной памятью:

1) Страницами по запросу
2) Сегментами по запросу
3) Сегментами, поделенными на страницы по запросу

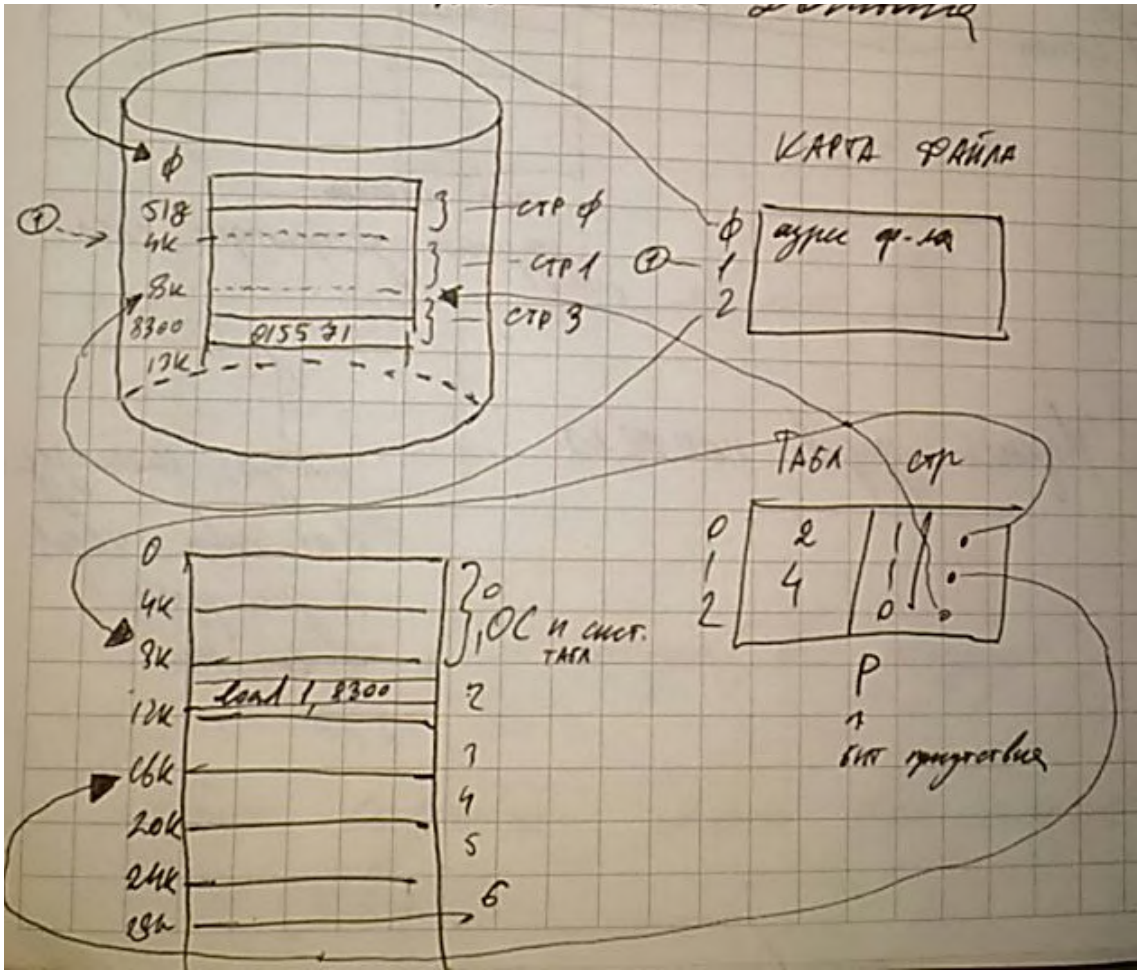
Часть памяти, выделенной процессу виртуализирована, часть находится в реальной памяти.

По запросам – загрузка в память осуществляется по запросу. Запрос возникает, когда программа обращается к команде или данным, отсутствующей в памяти. При обращении к команде, которой нет в памяти - возникает страничное прерывание (или прерывание при отсутствии сегмента памяти).

Исключения:

1. Страничное прерывание - страничная неудача
2. Прерывание по сегменту

Страничное преобразование

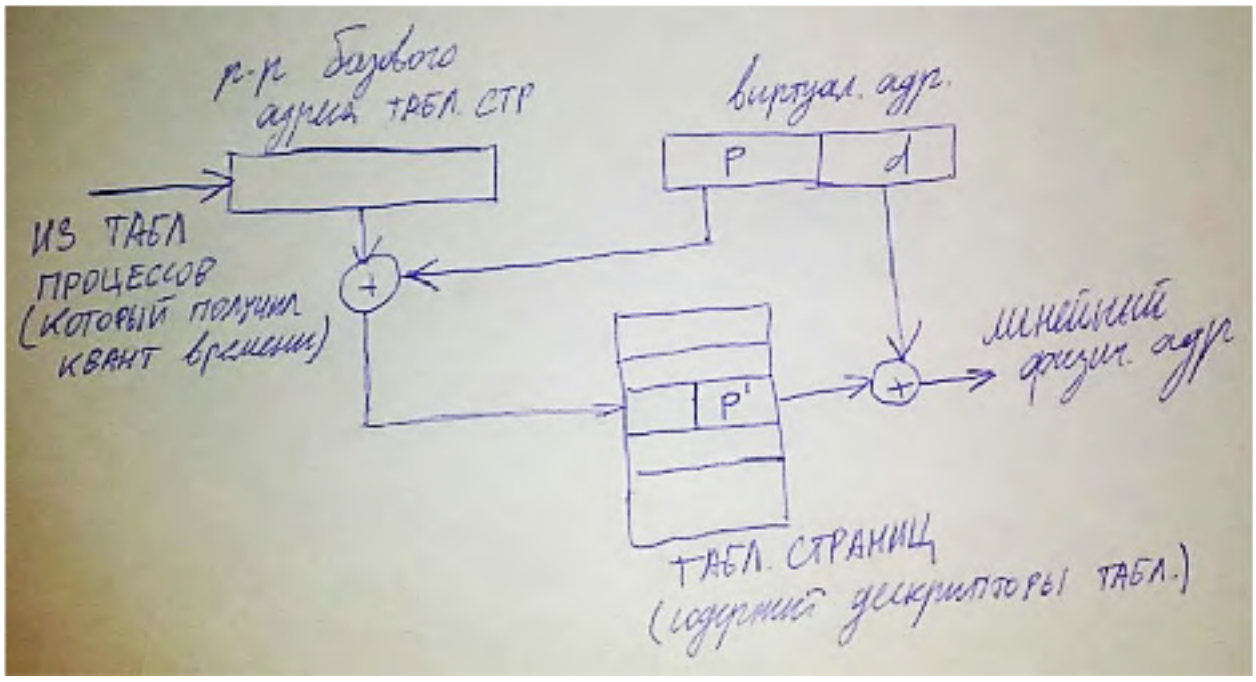


В результате обработки страничного прерывания загружается страница. Пока загрузка не произошла - процесс блокирован.

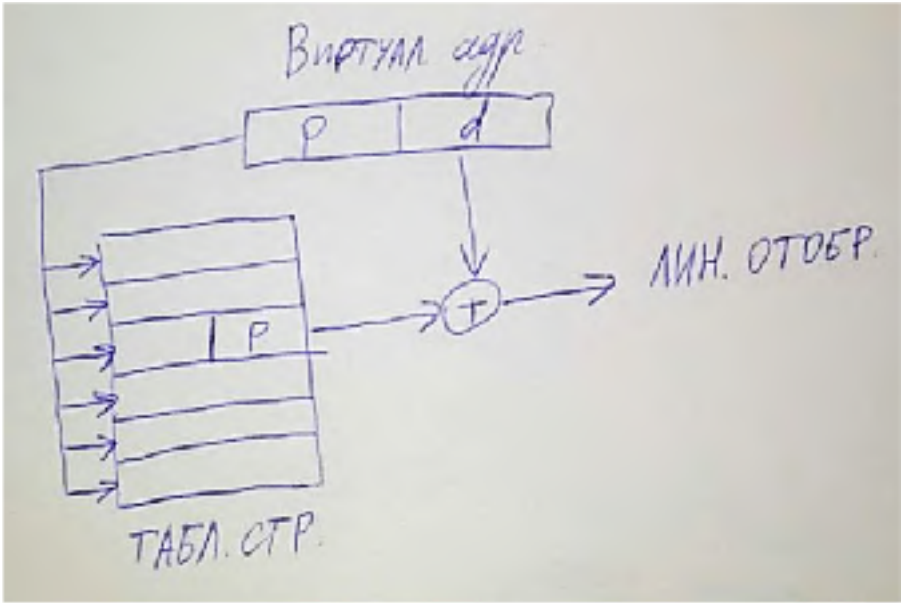
1. **Управление памятью страницами по запросу**

Прямое отображение

В процессоре должен быть регистр базового адреса таблицы страниц. Виртуальный адрес делится на две части (p, d). p - индекс страницы, d - смещение страницы. Таблица страниц расположена в физической памяти, таких таблиц столько, сколько процессов. Таблицы содержат дескрипторы страниц адресного пространства процесса.



Циклы обращения к памяти - последовательность действий по обращению к физической памяти.
Зачем: позволяет повысить уровень мультизадачности так как большее число программ может одновременно находиться в памяти. Однако возрастают затраты на вычисление физ. адреса.
только актуальные части??(актуальные страницы)

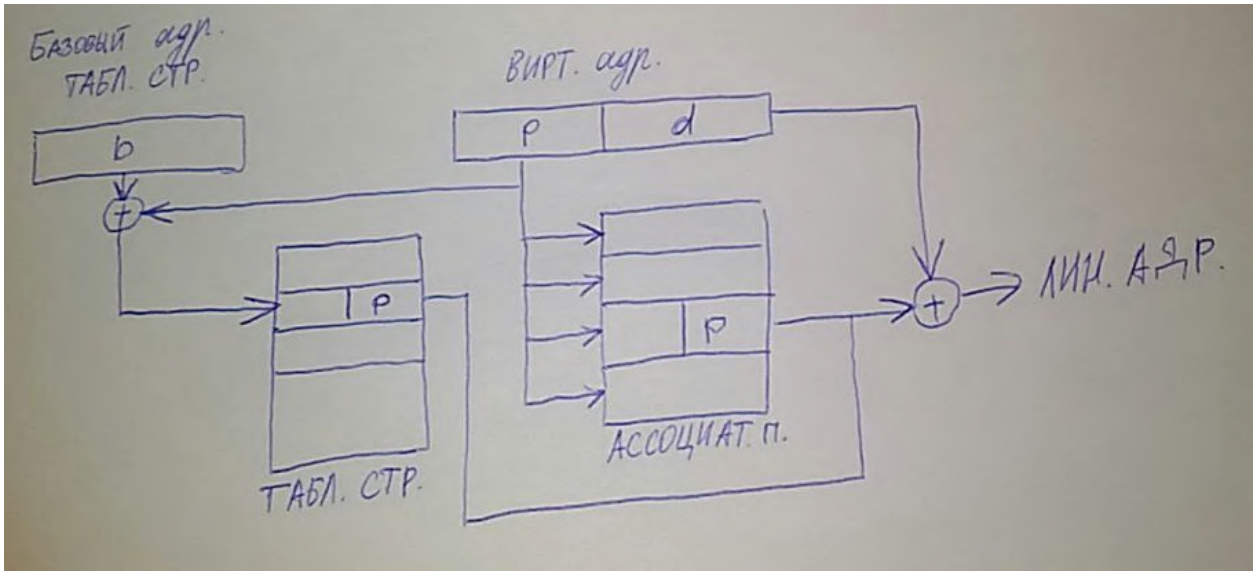


Ассоциативное отображение

// для сокращения времени обращения к памяти
Выполняется за 1 такт!
Это ассоциативная память, она дорогая, регистровая.

По ключу (w страницы), все разряды проявляются????
Много соединений, много компорандов!
Очень дорогая! Параллельное ассоциативное запоминающее устройство. В чистом виде не используется, а только в след. виде.

Ассоциативное прямое отображение



Кеш небольшого размера!

В процессоре есть регистр базового адреса таблицы страниц. В нем содержится начальный адрес таблицы страниц в оперативной памяти. Страница сначала ищется в ассоциативном кэше. Если в кэше нет, то происходит обращение к физической памяти. Замещение происходит тех страниц, к котором обращение было давно.

Даже при небольшом размере кэша, учитывая утверждение:

Если обращение было к странице, то следующее обращения будут к этой же странице В кеш будет обеспечивать показатели аналогии кешу полностью ассоциативно (до 90%). Это следует из свойства локальности программ.

Процесс во время своего существования но обращается ко всем своим страницам (фирма IBM показала это уже).

Обращение всегда происходит к подмножеству, которое меняется со временем.

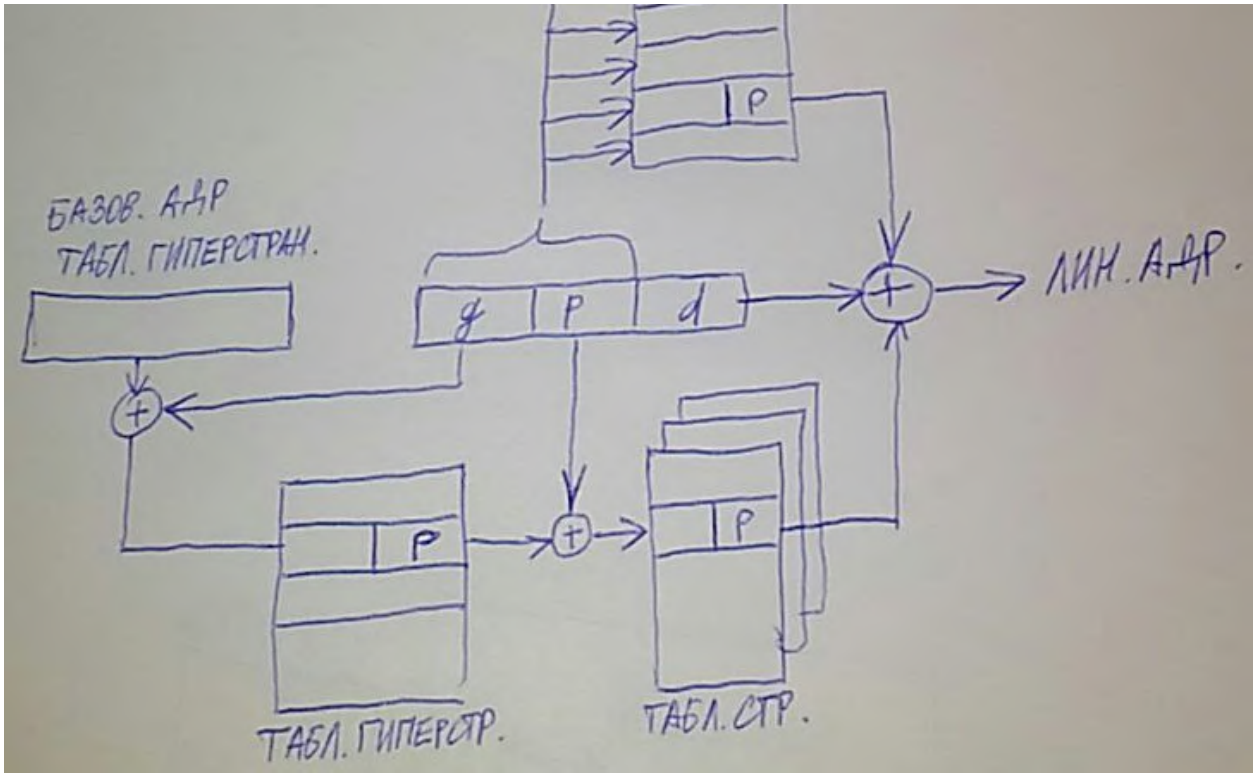
Чаще идет следование, а не обращение! Данные хранятся по последовательным адресам.

Двухуровневое страничное преобразование

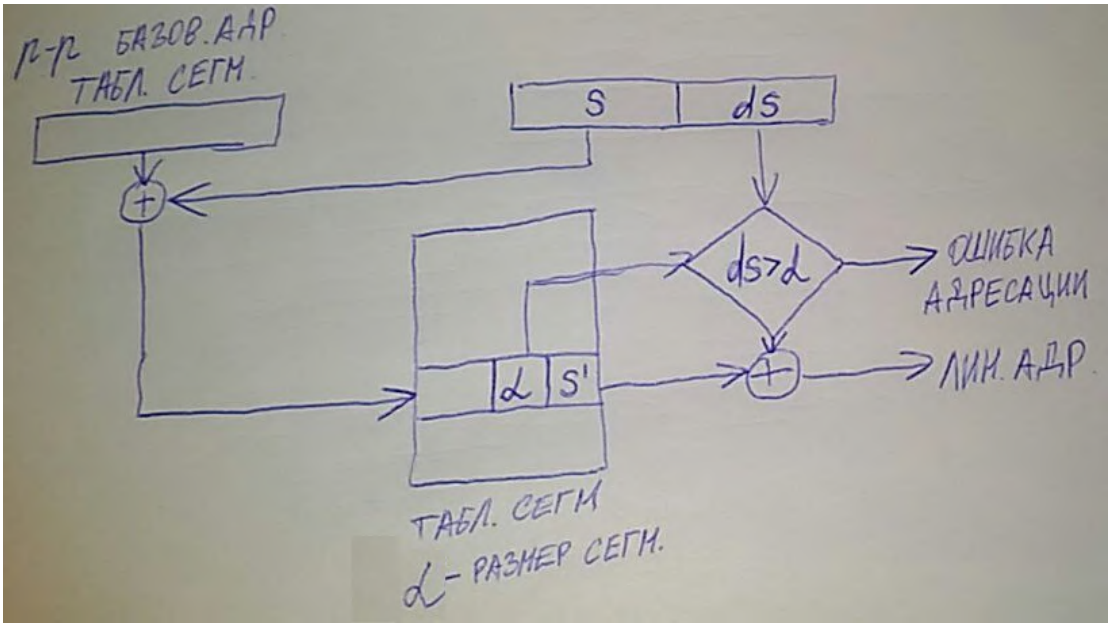
Появилось в IBM 360/67, 370

Рост ОП и Процессоров приводит к тому, что “растут” программы.

С ростом программ увеличиваются размеры страниц. Таблицы загружают память данными о страницах. Таблицы должны находиться в области данных ядра системы. Решение:



????для чего или что это? я полагаю gpd. Смещение, номер страницы, номер гиперстарницы.
Каждый процесс имеет таблицу страниц и таблицу гиперстраниц.
Это сделано для того, чтобы в памяти хранились только актуальные таблицы страниц.
Вводится понятие гиперстраницы. Виртуальный адрес делится на 3 части. Сохраняется стремление использовать ассоциативный буфер. Адресное пространство процесса делится на гиперстраницы, которые делятся на страницы. В результате таблиц страниц будет много, а таблица гиперстраниц – одна. Из таблицы гиперстраниц получаем базовый адрес таблицы страниц. P – смещение таблицы страниц. Если необходимой страницы нет в кэше, то обращаемся к ??? и получаем адрес страницы физической памяти, который можем использовать для ??? линейного адреса. Мы можем выгружать те таблицы страниц, которые в данный момент не используются (к которым не было обращения).



2. Сегментами по запросу

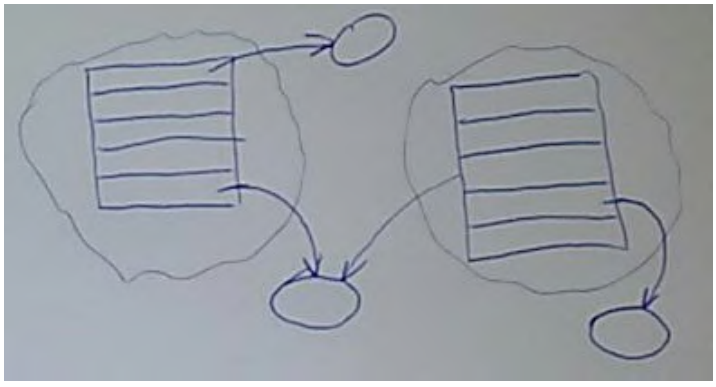
Сегмент - единица логического деления, размер определяется объемом программы(кода).

Система контролирует выход за пределы адресного пространства процесса. Ни один процесс не может вторгнуться в адресное пространство другого!
Таблицы дескрипторов сегментов могут быть **организованы 3 способами**:

1. Единая таблица (содержит все дескрипторы)

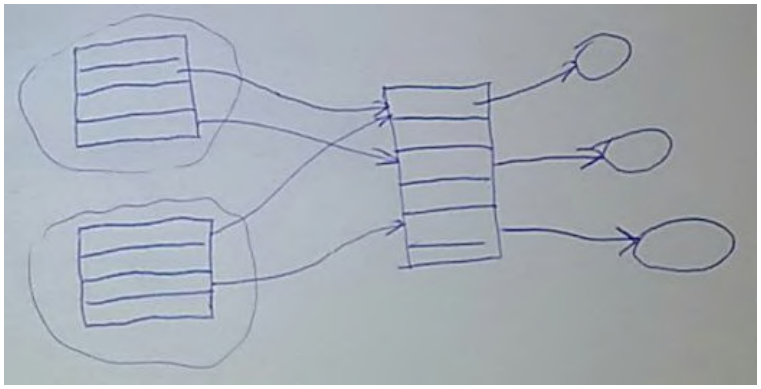
Каждый дескриптор содержит список прав доступа, который определяет доступ всех процессов к данному сегменту

2. Локальная таблица



Каждый процесс имеет свою таблицу дескрипторов. Если сегмент разделяется исполняемыми? процессами, то в каждой таблице будет дескриптор этого сегмента.

3. Локальная и глобальная таблица



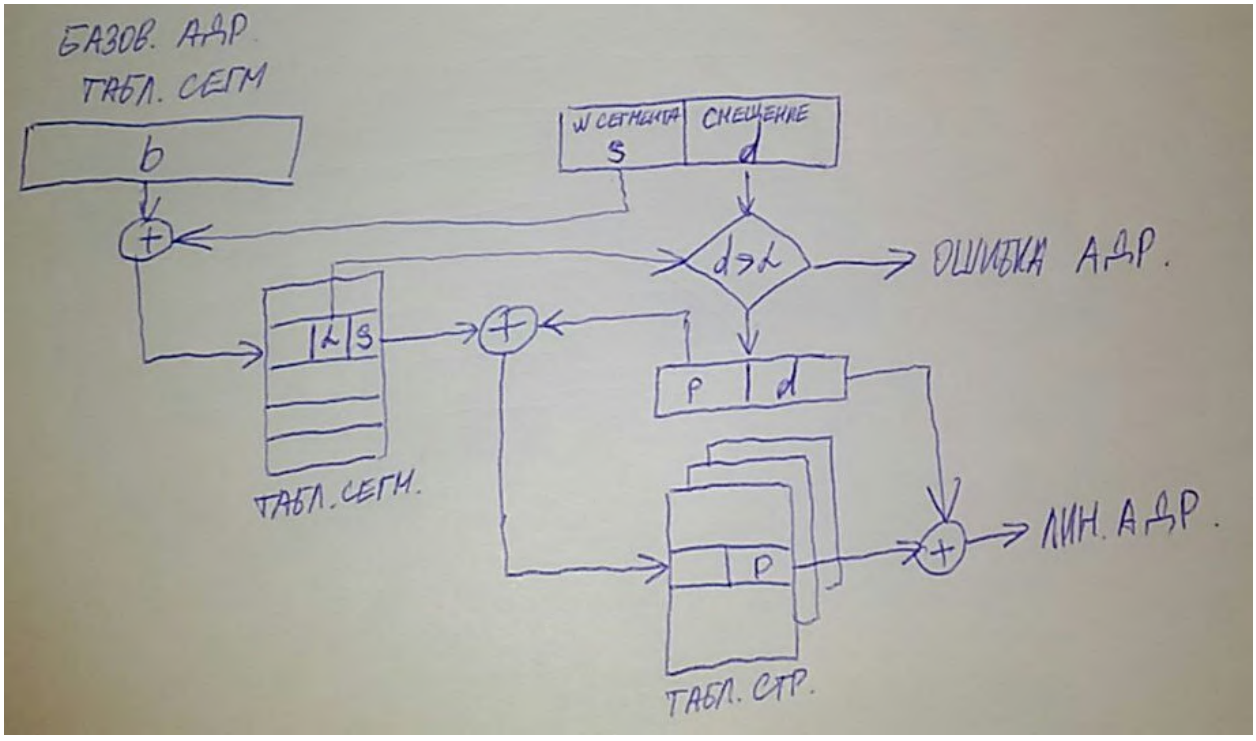
Смешанная организация - более гибкая и логичная. Каждый сегмент имеет глобальный дескриптор, который содержит всю информацию о физическом расположении сегмента (базовый адрес, размер, атрибуты(тип, бит ??, бит обращения)). Локальный дескриптор содержит информацию относящуюся к конкретному адресному пространству в т.ч. права доступа.

Теневые регистры

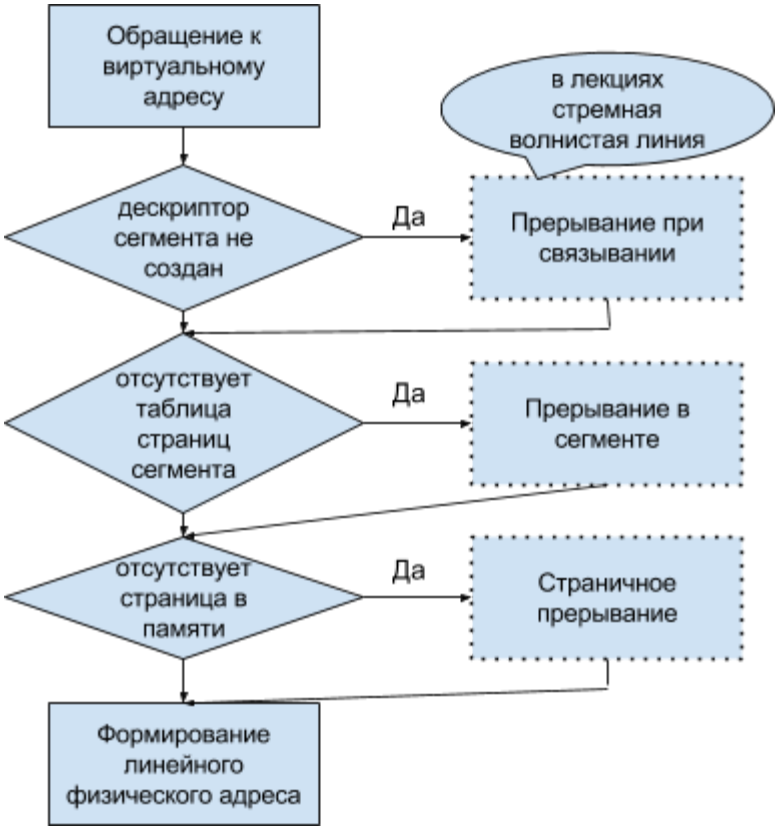
Сопоставлены в сегментными регистрами, когда обращен к сегментному регистру, то происходит заполнение, к ним нет доступа. Нужны для уменьшения задержек. Содержат информацию о сегменте.

Swapping - когда подгружается страница, то размер виртуальный и физический одинаков.

3. Управление памяти сегментами поделенными на страницы по запросу.



У процесса одна таблица сегментов и много таблиц страниц. Этот способ - попытка совместить достоинства двух предыдущих.



Отложенное(динамическое) связывание

Предполагается, что первоначально в память загружается только основная программа. Если встречается обращение к другим подпрограммам, то только в этот момент требуемые сегменты будут обработаны и в итоге соответствующий код будет загружен в память.

Прерывание при связывании - сегменту выделяется дескриптор сегмента (он же заполняется).

В результате обработки прерывания в сегменте создается таблицы страниц и заносится в дескриптор сегмента.

В результате обработки страничного прерывания страница загружается в физическую память.

В системе может создаться ситуация, когда свободных физических страниц нет. Тогда система должна выгрузить страницу, и на ее место загрузить нужную. Вопрос: какую страницу выбрать?

Page replacement(PR) - замещение страниц Рассмотрим несколько алгоритмов.

1) Выталкивание случайной страницы - с равной вероятностью может быть выгружена любая страница. Может быть выкинута активно используемая страница, а значит, ее обратно нужно будет загруж. Алг. малозатратн.
2) Алгоритм FIFO(очередь). **Или** каждой странице присваивается временная метка. Она обновляется при каждой загрузке новой страницы (тогда выталкивается страница с минимальной временной меткой). **Или** создается связный список (новые страницы ставятся в конец очереди, со временем они перемещаются в конец. Та, что стояла в конце - перемещается в начало, и потом будет вытолкнута). Т.о. не будет вытолкнута только что загруженная страница, но может быть вытолкнута часто используемая страница.

Аномалия FIFO: //При увеличении объема памяти число страничных прерываний должно сокращаться.

Примеры. объем - 3 страницы и 4 страницы. Будем исследовать одну и ту же траекторию страниц - номера страниц процесса, которые последовательно загружаются в память. //+ - загрузка новой страницы, (*) - вытеснение страницы

4	3	2	1	4	3	5	4	3	2	1	5
4	3+	2+	1+	4+	3+	5+	5	5	2+	1+	1
	4	3	2	1	4	3	3	3	5	2	2
		(4)	(3)	(2)	(1)	4	4	4	3	5	5

f = 9/12 = 75%

4	3	2	1	4	3	5	4	3	2	1	5
4	3+	2+	1+	1	1	5+	4+	3+	2+	1+	5+
	4	3	2	2	2	1	5	4	3	2	1
		4	3	3	3	2	1	5	4	3	2
			4	4	(4)	(3)	(2)	(1)	(5)	(4)	(3)

f = 10/12≈83%

3) Алгоритм LRU (list resently used) (LRUPR) - наименее используемая в последнее время. Алгоритм строится на эвристическом правиле: если к некоторой странице было обращение, то наиболее вероятно, что следующее обращение будет к этой же странице - свойство локальности программ.

Используем алгоритм для той же траектории. Реализуем алгоритм с помощью временных меток. Обращение перемещает страницу в конец списка //заметим, что 1-е 3 строки совпадают

4	3	2	1	4	3	5	4	3	2	1	5
4	3+	2+	1+	4+	3+	5+	4	3	2+	1+	5+
	4	3	2	1	4	3	5	4	3	2	1
		(4)	(3)	(2)	(1)	4	3	(5)	(4)	(3)	2

f = 10/12≈83%

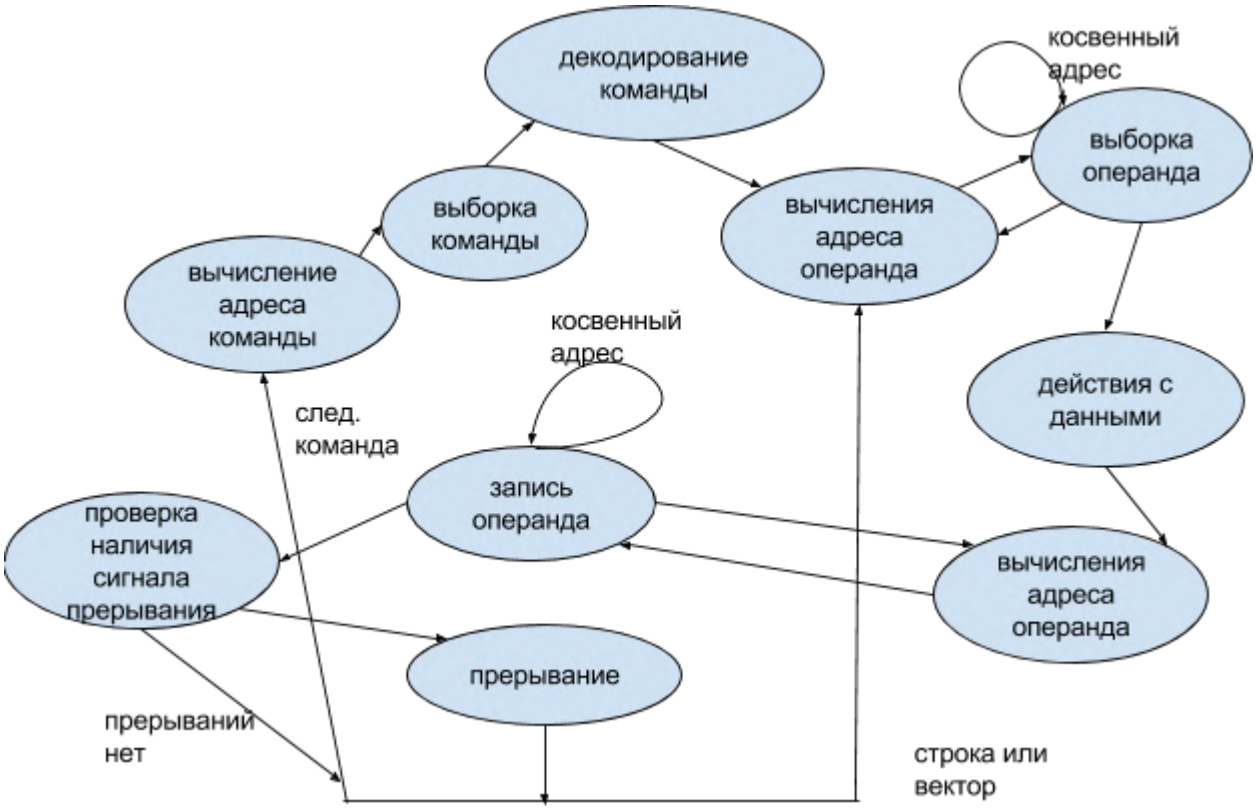
4	3	2	1	4	3	5	4	3	2	1	5
4	3+	2+	1+	4	3	5+	4	3	2	1	5+
	4	3	2	1	4	3	5	4	3	2	1
		4	3	2	1	4	3	5	4	3	2
			4	3	(2)	1	1	(1)	(5)	(4)	3

f = 8/12 Алгоритм LRU относится к стековым алгоритмам - не приводит к увеличению числа страничных прерываний. Алгоритм обладает свойством включения. В чистом виде алгоритм не используется(затратный - обращение к странице идет каждую команду

Выполнение команды

1. выборка команды
2. декодирование команды
3. выполнения команды

Диаграмма состояний цикла выполнения команды.



//есть команды регистр-регистр - тогда только выборка. Не бывает команд память-память.

4) LFU (list frequently used) наименее часто используемые страницы. Контролируется частота использования страниц. Недостаток: наименее интенсивно используемой может оказаться новая страница.
5) NUR (not used recently) - страницы, не используемые в последнее время. Для реализации данного алгоритма используется бит обращения. Этот алгоритм является аппроксимацией алгоритма LRU. Каждой физической странице нужно приписывать бит обращения. Периодически все биты обращения сбрасываются в ноль. При обращении бит устанавливается в 1. Для вытеснения ищется любая недостоверная таблица у которой сброшен бит обращения.

стремная картинка - //только что загружена 5-я страницы в 1-й фрейм. есть еще указатель удаления (он указывает на 1-й фрейм)

№ (фреймы)	Страница	Бит обращения
0	4	0
1	5	1
2	1	1
3	2	0

бит обращения (access bit)	бит модификации (dirty bit)
0	0
0	1
1	0
1	1

Кроме бита обращения используется бит модификации, который отражает, была модифицирована страница, или нет. Если страница не была модифицирована, то ее точная копия есть на диске => не надо выполнять копирование. Выгодно вытеснять немодифицированные страницы.

//ситуация (0,1) - норм, т.к. модификация могла быть выполнена до сбросов битов обращений

Локальное и глобальное вытеснение

Определяем, какую страницу выбираем для вытеснения: или любую (принадлежащую любому процессу) - **глобальное**, или страницу, принадлежащую тому процессу, который запросил новую страницу - **локальное**. Страничное прерывание - запрос дополнительного ресурса памяти. Приводит к блокировки процесса. Значит, нужно стремиться сократить блокировки. В системе есть поток(страничный демон), анализирующий … страницы и вытесняет те, к которым дольше всего не было обращения.

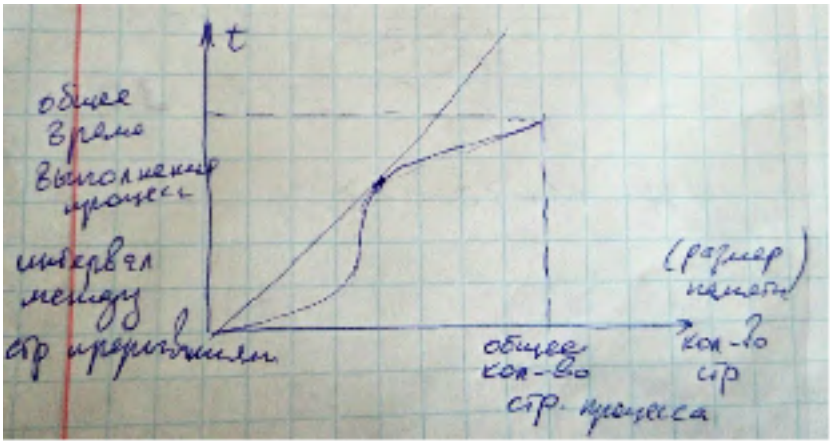
Теория рабочего множества

Связана с производительностью выполнения программ. Управления виртуальной памятью требует дополнительных ресурсов системы. Обработка страничных прерываний - затратное действие, связанное с обращением в MMU (manager memory). Зависимость страничных прерываний от объема памяти является обобщенной мерой страничного поведения программ. Поведение программы является нестабильным и неравномерным по времени: обращается к одному множеству своих страниц, потом к другому, но никогда не обращается ко всем своим страницам сразу (свойство локальности).

1968 г. Деннинг предложил в качестве локальной меры производительности взять число страниц, к которым обращается программа за интервал времени Δt. Обозначим w(t, Δt), w - множество страниц, называется рабочим множеством.

Размер рабочего множества является монотонной функцией от Δt, т.е. при увеличении интервала, число страниц будет стремиться к некоторому пределу. Именно этот предел определяет количество страниц, необходимых процессу для эффективного выполнения (без страничных прерываний). Если процессу удалось загрузить в память все необходимые ему в текущий момент времени страницы, то какое-то время он будет выполняться без страничных прерываний. Важно обеспечить такую возможность. Если процесс не может загрузить все страницы, то возможна пробуксовка (trashing) - подкачка одних и тех же страниц. Важно, чтобы процесс мог загрузить в память все свое рабочее множество. С течением времени он переходит к другому множеству. по оси x - время t.

В t_o выделяется начальное количество страниц (страницы сегмента данных, стека). После этого подкачиваются страницы, пока в памяти не окажется 1-е рабочее множество. Тогда процесс будет выполняться без страничных прерываний.



Исследования страничного поведения процесса в 60-70-е годы.

Большой вклад сделала IBM. Рассмотрим некоторые результаты исследований.

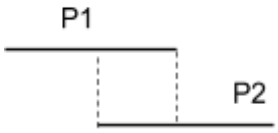
График зависимости длительности перехода между прерываниями - кривая времени жизни. Интервал между обращением и отсутствием страницы называется временем жизни. Перегиб происходит из-за того, что в некоторый момент времени в памяти оказывается все рабочее множество.

Число удачных обращений к странице называют hit rate. Чем больше процент удачных, тем более эффективно выполняется программа. hit rate - доля обращений, не потребовавших замены страницы. Рассмотрим соответствующий график:

Размер страницы фиксирован. на графике буква n = w

График отображает факт: если объем доступной памяти падает, то существует точка, начиная с которой количество страничных прерываний растет экспоненциально. Это связано с невозможностью процесса загрузить нужные страницы в память (из-за trashing)

Взаимодействие параллельных процессов.
Рассмотрим уровни исполнения

(1) Последовательно выполнение	(2) Квазипараллельное выполнение. Команды процессов выполняются последовательно. С точки зрения пользователя процессы выполняются параллельно. Программы P1 и P2 должны находиться в физической памяти.	(3) Параллельность (реальная параллельность). Выполнение процессов совпадает по времени на уровне команд.
		

Проблемы, связанные с параллельностью выполнения процессов одинаковы и в системах с (2) и с (3).
Процессоров меньше чем процессов, поэтому часть процессов выполняется квазипараллельно.
Проблемы. Пример 1. Ситуация. 2 филиала банка в конце рабочего дня должны перевести на 1 и тот же счет какие-то суммы.
P1: … S = S + V_1 … P2: … S = S + V_2 ….
Пусть первым начал 2-й филиал, взял S и V_2, сложил, и квант закончился. Далее выполнялся 1-й филиал. сделал тоже самое и отправил результат в S.Потом 2-й филиал продолжил выполнение и записал результат заново, т.е. потеряна сумма V_1.
Пример 2. Рассмотрим выполнение действия на 2-х процессорах в многопроцессной системе. Пусть P1 выполняется на П1, а P2 на П2. У каждого П свой eax

P1	eax	myvar=0	eax	P2
mov eax, myvar	0	0		
	0	0	0	mov eax, myvar
	0	0	1	inc eax
	0	1	1	mov myvar, eax
inc eax	1	1	1	
mov myvar, eax	1	1	1	
Рассмотрим выполнение на 1 процессоре.				
P1	eax	myvar=0	eax	P2
Выполняется P1				
mov eax, myvar	0	0		
P1 вытеснен, выполняется P2				
контекст P1 сохранен		0	0	mov eax, myvar
		0	1	inc eax
		1	1	mov myvar, eax
P2 вытеснен, выполняется P1				
inc eax	1	1		контекст сохранен
mov myvar, eax	1	1		

Происходит потеря данных. myvar называется разделяемой переменной
Необходимо обеспечить монопольный доступ процесса к разделяемой переменной. Когда процесс выполняет действия над разделяемой переменной никакой процесс не должен выполнять никакую работу над ней.
Монопольный доступ обеспечивается методом взаимоисключения, т.е. если процесс находится в своей критической секции, то никакой другой процесс не может находится в этой же критической секции по той же разделяемой переменной.
Ситуацию часто называют гонки (Rase Condition)
Пример: 2 последовательности неделимых действий(атомарностей)
P: a b c
Q: d e f
a, b, c, d, e, f - атомарные действия.
Если P и Q выполняются последовательно, то получим abcdef.
При псевдопараллельном выполнении P и Q получим разные комбинации: abcdef, abdcef, abdecf … defabc.

Т.о. результат псевдопараллельного выполнения может отличаться от последовательного выполнения, если оба процесса модифицируют одни и теж же переменные. Будем говорить, что набор называется детерминированным, если при псевдопараллельном выполнении для одних и тех же данных он дает одинаковые данные (результат).
В противном случае, он называется недетерминированным.
Условия Берстайна Если R(p) - набор входных переменных. W(p) - набор выходных значений.
Для двух активностей P и Q: $W(P) \cap W(Q) = \emptyset$, $W(P) \cap R(Q) = \emptyset$, $R(P) \cap W(Q) = \emptyset$ => выполнение P и Q детерминировано. Если условие не выполняется, то однозначно установить, детерминированы или недетерминированы нельзя.

Чисто программный способ реализации взаимоисключения

Метод взаимоисключения обеспечивает монопольный доступ к взаимозаменяемым ресурсам.
Mutual exclusion (mutex - взаимное исключение). Решается с помощью флага.

<table>
<tr><td colspan="2">fl1, fl2:logical</td><td colspan="2">fl1, fl2:logical; QVE:int; //Решение - математик Деккар</td></tr>
<tr><td>P1:
while(1) do
{
  while(fl2) do;
  fl1 = 1; (*)
  CR1
  fl1 = 0;
  PR1;
}</td><td>P2:
while(1) do
{
  while(fl1) do;
  fl2 = 1; (*)
  CR2
  fl2 = 0;
  PR2;
}</td><td>P1:
while(1) do {
  fl1 = 1;
  while(fl2) {
  if(QVE==2) then {
  fl1 = 0;
  while(QVE==2) do;
  fl1 = 1;
  }
  }
  CR1; fl1 = 0; QVE = 2; PR1;
}</td><td>P2:
while(1) do {
  fl2 = 1;
  while(fl1) {
  if(QVE==1) then {
  fl2 = 0;
  while(QVE==1) do;
  fl2 = 1;
  }
  }
  CR2; fl2 = 0; QVE = 1; PR2; }</td></tr>
<tr><td colspan="2">fl1 = 0; fl2 = 0;
par begin
  P1; P2;
par end</td><td colspan="2">fl1 = 0; fl2 = 0; QVE = 1;
par begin
  P1; P2;
par end</td></tr>
</table>

Данный подход не обеспечивает гарантированного взаимоисключения.
(*) - если fl поменять с while местами, то получим бесконечные циклы - тупик т.к. установлены флаги двух процессов сразу.
while - активное ожидание на процессоре.
Решение - математик Деккар - решил только для 2-х процессов. Используются 2 флага и переменная “чья очередь”

Петтерсон решение:

<table>
<tr><td colspan="2">fl1, fl2:logical
QVE:int</td><td rowspan="3">P1 после установки своего флага fl1 не сможет войти в свой критический участок, если P2 уже находится в своем критическом участке. т.к. fl2 = 1 и для P1 это закрывает возможность входа.
Есть 3 ситуации:
1) P2 не хочет входить в свой критический участок - невозможно, т.к. при этом выполнилось бы присваивание fl2 = 0
2) P2 постоянно ожидает входа в критический участок, если QUE == 2, то P2 может войти в свой критический участок
3) P2 монополизировал вход в крит участок - невозможно, так как P2 перед попыткой входа должен дать P1 возможность войти в его критический участок, устанавливая QVE = 1.</td></tr>
<tr><td>P1:
…
while(1) do
{
  fl1 = 1;
  QVE = 2;
  while(fl2&&QVE==2) do;
  CR1
  fl1 = 0;
  PR1;
}</td><td>P2:
…
while(1) do
{
  fl2 = 1;
  QVE = 2;
  while(fl2&&QVE==2) do;
  CR2
  fl2 = 0;
  PR2;
}</td></tr>
<tr><td colspan="2">fl1 = 0; fl2 = 0;
par begin
  P1; P2;
par end</td></tr>
</table>

Аппаратная реализация -
команда test_and_set - это атомарная команда, (появилась начиная с IBM 370) которая:

1) Читает значение переменной b
1) Копирует его в a
2) Для b устанавливает значение источника.

На командах test_and_set построены списки блокировки (spin_lock - активное ожидание, крутится в блокировке).
Части кода ядра системы не могут блокироваться, поэтому нужно активное ожидание.

fl, c1, c3 :logical

P1:
```
while(1) do
{
   c1 = 1;
   while(c1) do;
   {
      test_and_set(c1, fl)
   }
   CR1; fl = 0; PR1; }
```

P2:
```
while(1) do
{
   c2 = 1;
   while(c2) do;
   {
      test_and_set(c2, fl)
   }
   CR2; fl = 0; PR2; }
```

```
fl = 0;
par begin
   P1; P2;
par end
```

fl = true когда любой из процессов находится в своем критическом участке.
Пусть 1 процесс хочет войти в крит. участок, а второй процесс уже в нем.
1 Процесс устанавливает c1 = 1 и входит в цикл проверки переменной fl команды test_and_set. Поскольку 2 процесс уже в своем крит участке, то значение fl = 1. Когда test_and_set обнаруживает этот факт и устанавливает c1 = 1. Т.о. 1 процесс находится в своем цикле активного ожидания до тех пор, пока 2 процесс не выйдет из своего крит. участка. Этот способ не исключает бесконечного откладывания, но считается, что его вероятность очень мала. Когда процесс выходит из своего крит участка и устанавливает флаг fl в 0, то скорее всего 2 процесс сможет перехватить инициативу и установить fl в 1.

while … { test_and_set() }
Команда test_and_set в цикле также называется циклической блокировкой (spinlock)

```
void spin_lock(spin_lock_t *c)
{
   while(test_and_set(c)!= 0)
   {
      \\ресурс занят
   }
}
```

```
void spin_unlock(spin_lock_t *c)
{
   c = 0;
}
```

Реализация test_and_set во многих архитектурах связана с блокировкой шины памяти.

Однако данный цикл проверки может привести к занятию памяти одной нитью, что влечет существенное снижение производительности системы. Это решается путем использования двух вложенных циклов.

```
void spin_lock(spin_lock_t *c)
{
   while(test_and_set(c)!= 0)
   {
      while(c != 0) \\ не связано с блок-й
шины пымяти
      \\ресурс занят
   }
}
```

```
void spin_unlock(spin_lock_t *c)
{
   c = 0;
}
```

spin_lock чаще всего используется в ядре, так как многие части ядра не могут блокироваться.
Активное ожидание(на процессоре) - зачастую негативный факт.
Поэтому Дейкстра,работая над проблемной монопольного доступа и взаимоисключения,в 1965 году в своих работах

предложил семафоры.

Классические семафоры.

Семафор - неотрицательная защищенная переменная S >= 0, на которой определены 2 операции $P(S)$ - пропустить, и $V(S)$ - освободить. Переменная защищена, т.к. м.б. изменена только этими двумя неделимыми операциями.

Бинарный семафор - семафор, который может принимать лишь два значения - 0\1

Считающий семафор - семафор, который может принимать любые неотрицательные целые значения.

P(s): s = s - 1 Если s = 0, то процесс блокируется на семафоре (процесс переводится в состояние ожидания освобождения нужного ему ресурса).

V(s): s = s + 1 Если s = 0, то активизируется некоторый процесс, ожидает освобождения ресурса (если таковой конечно был).

Эти операции атомарны.

Семафоры исключают активное ожидание, поскольку если процесс не может захватить семафор, то он переводится в состояние ожидания, причем активизирует его другой процесс, вызвавший V(s).

Семафоры реализуются на уровне ядра, это системный вызов. Так как только ядро может блокировать и разблокировать процессы. С каждым семафором может быть связана очередь процессов.

+ Нет активного ожидания
- Требуют перехода в режим ядра(а это переключение контекста).

P1: … P(s) CR1 V(s)	P2: … P(s) CR2 V(s)	Семафоры являются высокоуровневыми действиями в ОС, они выполняют сложные действия, но в основе лежат машинные команды, очень часто это test_and_set.

Считающие семафоры (на примере задачи “производство-потребление”)

Задача производство-потребление: Имеется ограниченный буфер, Producter может только класть данные в буфер, Consumer(потребитель) только забирать.

```
Program semaphore
N, se, sf, sb :int;
```

```
Producter:
while(1) do {
   \\создать данные
   P(se)
   P(sb)
   N+=1;
   V(sb)
   V(sf) }
```

```
Consumer:
while(1) do {
   P(sf)
   P(sb)
   N-=1; \\ взять
   V(sb)
   V(se)
}
```

```
begin
   se = N; sf = 0; sb = 1;
   parbegin:
      Producter;
      Consumer;
   parend
end
```

В начальный момент буфер пуст, поэтому sf = 0, а se = N.

Producter может класть, если se != 0;

Consumer может брать, если sf != 0;

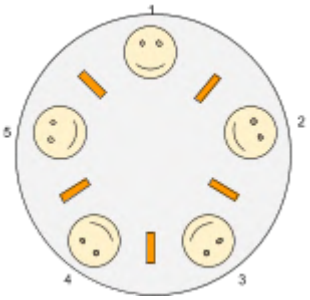
Producter будет блокирован если буфер полон.

Consumer будет блокирован если буфер пуст.

Producter положив данные увеличивает sf.

Consumer взяв данные увеличивает se.

Бинарный семафор регулирует доступ к разделяемой переменной N. Т.о. se, sf считающие семафоры, содержат количество пустых и полных ячеек буфера соответственно.



Обедающие философы

В современных системах используются множественные семафоры - наборы считающих семафоров. Они определяются как массивы семафоров.

Рассмотрим наборы считающих семафоров на примере задачи “обедающие философы” (предложена Дейкстрой)

```
var
        forks:array [1..5] of semaphore;
        i:integer;
        {
        i := 5;
        do {
                forks[i] = 1;
                i--;
        } while (i != 0);
cobegin
        1: {left = 1, right = 2; ...}
…
        5. {left = 5; right = 1;
                do {
                <размышляет>
                P(forks[left], forks[right]);

                <ест>
                V(forks[left], forks[right]);
                } forever;
        }
```

Основная особенность наборов считающих семафоров заключается в том, что одной неделимой операцией можно проверить часть или все семафоры набора.

Задача: на круглом столе расставлены тарелки, по одной на каждого философа. В центре стоит большое блюдо. На столе 5 вилок, по одной между каждыми тарелками.

Философ либо размышляет, либо ест. Чтобы начать есть, необходимо выполнить ряд условий. Предполагается, что любой философ должен положить спагетти на тарелки с помощью 2-х вилок и не выпуская вилок начать есть. Закончив есть, философ кладет вилки справа и слева и опять размышляет и т.д.

Существует три способа действия философа:

1) Каждый из философов пытается взять себе 2 вилки одновременно. Если это ему удается, он начинает есть.
2) Каждый из философов пытается взять сначала правую вилку. Взяв правую, пытается взять левую.
3) Каждый из философов пытается взять сначала правую вилку. Взяв правую, пытается взять левую. И если левую взять не может, то кладет правую

Возможны три сценария действия философов (эти ситуации – негативные в системе):

1) Бесконечное откладывание - голодание: в каждый момент времени могут есть только 2-е.
2) Тупик (deadlock)
3) Захват и освобождение одних и тех же ресурсов. (система работает, ни ничего не делает) Пример: подкачка одних и тех же страниц - trashing.

Задача - это модель,связанная с распределением ресурсов в системе.

Семафоры не являются единственным средством реализации взаимоисключений. Очень часто такие средства называются lock and unlock. lock связан с блокировкой процесса, а unlock - с разблокировкой процесса и получения им ожидаемого ресурса (иногда называется wait и signal, в современных системах используются mutex[mutual exclusion]).

Различия: семафор не имеет владельца, т.е. семафор может освободить процесс, который его не захватывал. Для него не обязательно парность захвата/освобождения.

P1: … P(s) … | P2: … V(s) ...

Mutex наоборот, имеет владельца. Освободить его может только процесс, который его захватил. Кроме того, семафоры могут быть удалены, т.е. могут перестать существовать. С mutex это невозможно.

Результат использования семафор привел к тому, что система взаимодействующих процессов попадала в тупик. Трудно уследить за большим числом семафоров.

Для того, чтобы использовать схему взаимоисключения, были использованы мониторы. //(Для структурирования систем взаимоисключения были предложены мониторы)

Мониторы

Идея монитора заключается в создании механизма, который соответствующим образом унифицировал бы взаимодействие параллельных процессов и обеспечивал монопольное использование разделяемых ресурсов.

Мониторы - средства структурирования ОС. Они могут входить в состав ЯП. (Например, монитор в качестве элемента) (какой-то Pascal, Ada). Но очень часто для обеспечения задачи распределения ресурсов вычислительной системы по каждому типу ресурса строится соответствующий планировщик. И именно такую схему декларирует монитор. В частности, в Linux ядре реализован монитор чтения/записи и ядро предоставляет соответствующие функции read lock, read unlock, write lock, write unlock.

Монитор обозначается ключевым словом monitor. Он представляет из себя конструкцию, состоящую из полей данных и набора подпрограмм, которые могут применять эти данные.(они появились раньше объектов). Монитор защищает свои данные - доступ осуществляется только с помощью функций монитора. Он гарантирует, что функции монитора может использовать только один процесс - он называется находящимся в мониторе. Если 1 процесс в мониторе, то остальные, обращающиеся к монитору, блокируются и ставятся к нему в очередь (она находится вне монитора). На мониторах определены 2 операции: wait() и signal(), которые изменяют переменную типа условие (conditional).

wait блокирует процесс, вызвавший монитор в том случае, если ресурс занят, а сигнал, который выполняется другим процессом, освободит процессы заблокированные в очереди к монитору.

Примеры мониторов

1. Простой монитор обеспечивает выделение единственного ресурса произвольному числу процессов.

<table>
<tr><td colspan="2">resource: monitor
var
busy: logical;
x : conditional;</td></tr>
<tr><td>procedure acquire; //приобретать
begin
if busy then wait(x)
busy = true;
end;</td><td>procedure release; //освобождать
begin
busy = false;
signal(x);
end</td></tr>
<tr><td colspan="2">begin
busy = false;
end;</td></tr>
</table>

Монитор обслуживает произвольное число процессов, ограниченное только возможной длинной очереди. Когда к монитору обращаются для захвата ресурса - вызывается acquire. Если busy = true, то по переменной типа условие выполняется команда wait(x). Это приводит к блокировке процесса, значение busy не меняется.

Если busy = false, то обратившийся к монитору процесс сразу получает доступ к ресурсу и устанавливает значение переменной busy в true, т.о. захватывает ресурс.

Если процесс хочет освободить ресурс, он вызывает release, busy становится false, вызывает signal(x) - разблокирует процесс, находящийся в очереди к монитору.

2. Монитор кольцевой буфер - решает задачу производство-потребление.

```
resource: monitor
var
   bcircle: array[0..n-1] of type;
   pos: 0..n; //текущая позиция
   j: 0..n-1;//заполняемая позиция
   k:0..n-1;.//освобождаемая позиция
   buffer_full, buffer_empty: conditional
```

```
   procedure producer(data: type)
      begin
         if pos == n  then wait(buffer_empty)
         bcircle[j] = data;
         pos := pos + 1;
         j := (j+1) mod n;
         signal(buffer_full);
      end;
```

```
   procedure consumer(var data: type)
      begin
         if pos == 0 then wait(buffer_full);
         data := bcircle[k];
         pos = pos -1;
         k := (k+1) mod n;
         signal(buffer_empty);
      end
```

```
   begin    pos := 0;  j := 0;   k := 0;  end;
```

wait - блокирует (без активного ожидания), signal - разблокирует.

Если n = pos producer блокируется на buffer_empty. Освободив ячейку, consumer вызывает системный вызов signal(buffer_empty);

Consumer блокирован, если буфер пуст - возможно, если producer отстает от consumer.Тогда consumer будет блокирован на buffer_full. Его разблокирует producer, который положит данные и вызовет signal(buffer_full).

Монитор "читатели-писатели" Классический вид, предложенный Хоаром.

4 функции: start_read, stop_read, start_write, stop_write.

Применяется в ядре LINUX ReadLock, ReadUnlock, WriteLock, WriteUnlock. Данная задача характерна для ядра и часто встречается.

Бытовая интерпретация – система продажи билетов на самолеты, когда бронируются места, конкретный день и рейс. Состоит из 2 этапов: 1) чтение информации, указываем вылет прилет, дату . . . 2) бронирование. Место – критическая переменная. Когда бронируем – мы выступаем как писатель.

В мониторе читатель-писатель предполагается наличие разных типов процессов. Читатели могут читать информацию параллельно. А вносить изменения могут только процессы-писатели в режиме монопольного доступа.

Блокировать всю базу нет смысла - блокируется конкретное поле (используется в режиме монопольного доступа).

Существуют ядерные потоки, которые читают, и потоки-писатели, которые могут изменять системные таблицы.

```
resource: monitor
var
   nz: integer; \\ читатели
   wrt: logical; \\ писатель
   c_read, c_write: conditional;
```

```
procedure start_read()
   begin
      if wrt or turn(c_write) then wait(c_read);
      nz := nz + 1; // увеличивыем число чит.
      signal(c_read);
   end;

procedure stop_read()
   begin
      nz := nz - 1;
      if nz == 0 then signal(c_write);
   end;
```

```
procedure start_write()
   begin
      if nz > 0 or wrz then // есть читатель или писатель
           wait(c_write); // блокирует
      wrt = true; // есть активный писатель
   end;
procedure stop_write()
   begin
      wrt = false;
      if turn(c_read) then
         signal(c_read);
      else:
         signal(c_write);
   end;
```

```
begin
   nz = 0;
   wrt = false;
end;
```

Когда число читателей = 0, процесс писатель получает возможность писать, если нет другого писателя. Если есть читатели или другой процесс писатель, то процесс блокируется. на c_write.

если условие не выполняется, то wrz = true; Процесс, который хочет начать читать, должен вызвать start_read. Он сможет начать, если нет активного писателя и [ждущих писателей(с)Свят](очередь читателей пуста (с) Наташа), иначе ожидает на c_read; Когда смог - счетчик активных читателей +1 и вызывается сигнал можно читать (Цепная реакция запуска читателей).

stop_write - завершение записи. Если очередь читателей не пуста, то вызывается сигнал можно читать(c_read), а если очередь пуста, то - можно писать(c_write).

stop_read - окончание чтения, счетчик активных читателей -1, если счетчик = 0, то сигнал по переменной можно писать - c_write.

В данном мониторе исключается бесконечное откладывание читателей благодаря тому, что процесс писатель по окончании записи проверяет очередь ждущих читателей. Прежде чем начать работу, процесс читатель проверяет нет ли ждущих писателей => исключается бесконечное откладывание писателей

Синхронизация и взаимоисключение

Задача производство-потребление - Синхронизация (один процесс должен ждать, что другой придет в некоторую точку). Читатели-писатели - Взаимоисключение (ни читатель, ни писатель не ждут определенных действий другой стороны).

Алгоритм (Лампорта - Lamport) “Булочная”

Решает задачу взаимоисключения для n процессов.

Идея алгоритма заимствована из принципов работы магазина. Каждому клиенту выдается листок с номером. Когда продавец освобождается, он обслуживает клиента с наименьшим номером. Если два клиента находятся в двери одновременно, то им выдаются одинаковые номера, но первым обслуживают клиента у которого меньше идентификатор.

Процесс, ожидающий возможности войти в критическую секцию выбирает номер, который должен быть больше чем все остальные номера, используемые на текущий момент. Это глобальный разделяемый массив текущих номеров, доступный каждому процессу. Входящий процесс проверяет всех и ждет каждого, у кого меньший номер.

Когда два процесса выполняют действия одновременно, то конфликт разрешается с помощью идентификаторов.

```
typedef char boolean …
shared boolean choosing[n];
shared int num[n]; …
for(j = 0; j < n; j++)
   num[j] = 0;  ...
/*choose a number*/ //начало крит секции
chosing[i] = TRUE; //i-й процесс выбирает номер
num[i] = max(num[0], …, num[n-1]) + 1;
choosing[i] = FALSE; //конец крит секции
/*for all other processes*/
        for (j = 0; j < n; j++) {
                /* wait if the process in currently choosing */
                while(choosing[i]) { /* nothing */ }
                /* wait if the process has a number and comes ahead of us*/
```

```
                if (num[j] > 0) && ((num[j] < num[i]) || (num[j] == num[i]) && (j < i))   {
                        while(num[j] > 0) { /* nothing */ }
                }
                /* critical section */
        }
```

у Лампорта

```
if ((num[i] != 0) && ((num[j], j) < (num[i], i)))
{ ….. }
```

Если процессы не пытаются войти в критический участок, то их num = 0. Если процесс выбирает номер, когда другой процесс пытается его найти, то этот другой процесс ждет до тех пор, пока номер не будет выбран.

Ада. Рандеву.

Выделяется метод **рандеву** (в языке Ада). Язык Ada является языком параллельного программирования. Создавался для военных целей, моделирования взаимодействия различных родов войск. В настоящее время используется для реализации банковских операций, управления движущимися объектами, телефонными компаниями. Язык создан в 1986 г в честь Ады Вонг Лавлейс.

В языке Ada работа с параллельными процессами осуществляется с помощью task-ов - модулей, которые могут выполняться параллельно. Задачи могут работать независимо друг от друга, но они также имеют средства взаимодействия друг с другом. Не регламентируется, как должны быть реализованы параллельные задачи. Ada интерпретирует задачи как логические объекты, которые ведут себя так, будто выполняются на разных машинах. Однако, задачи не могут быть интерпретированы по отдельности.

```
task <идентификатор задачи> is
        entry <идентификатор входа> (дискретный
        диапазон) (формальная часть);
                <другие объявления входа или фразы
        представления>
end <идентификатор задачи>;
task body <идентификатор задачи> is
        …

        select

                accept <идентификатор входа> (…)
                end <идентификатор входа>;
        or
                Accept <идентификатор входа> (…)
                End <идентификатор входа>;
        end select;
end <идентификатор задачи>;

accept:
accept <идентификатор входа> (выражение)
                                        формальная часть
do <последовательность операторов>
end;
```

В Аде основной единицей исполнения являются задачи - task. Связываются между собой при помощи кодов.

Если одна задача выдала сообщение, и оно было принято другой задачей, то обе задачи теряют свою независимость и устанавливают рандеву. До тех пор, пока оно действует, задачи синхронизированы.

Рандеву начинается с согласование фактических и формальных параметров, затем оно продолжается выполнением операторов, расположенных после accept между do и end.

Важнейший момент в рандеву – последовательность операторов выполняется от имени обеих задач, которые находятся в рандеву, т.е. результат выполнения последовательности операторов одновременно получают 2 параллельные задачи. В этом и есть особенность рандеву. Нет задержки получения результата, поэтому этот способ выделен как отдельный способ.

Ада - язык высокого уровня, рандеву реализовывается соответствующим транслятором. Результат выполнения опр. последовательности действий ОДНОВРЕМЕННО получают 2 процесса.

Синхронизация в распределенных системах

(Синхронизация взаимоисключения в распределенных системах)

Необходимо различать синхронизацию (связана с тем, что один процесс вынужден ждать, пока другой процесс выполнит работу, в которой заинтересован этот процесс. Производство-потребление – связана с синхронизацией – один и процессов будет блокирован, если другой работает «слишком быстро») и взаимоисключение.

Задача читатели-писатели – на взаимоисключение. Есть база, массив структур, процессы, которые могут только читать информацию и те, которые могут изменять значение полей в этом массиве структур. Читатель не ждет, пока писатель напишет что-то конкретное, а ждет возможности читать. Он не заинтересован в конкретном действии писателя.

В распределенных системах процессы могут взаимодействовать только путем передачи сообщений.

Процессы могут обратиться к адресному пространству ядра системы и получить нужную им информацию, т.е. существует адресное пространство, которые процессы могут разделять. В распределенных системах такое невозможно. У процессов, которые выполняются на разных машинах, нет общей памяти. При взаимодействии, сообщения между процессами оформляются в виде пакетов.

Пакет – конверт, на котором написан адрес, а внутри находятся данные.

Для того, чтобы сообщение мог получить адресат, на нем должен быть адрес и блок данных для передачи.

Есть задачи, связанные с распределением ресурсов.

Как люди синхронизируют свои действия в жизни? Договариваются по времени – действуют в соответствии с показаниям своих часов.

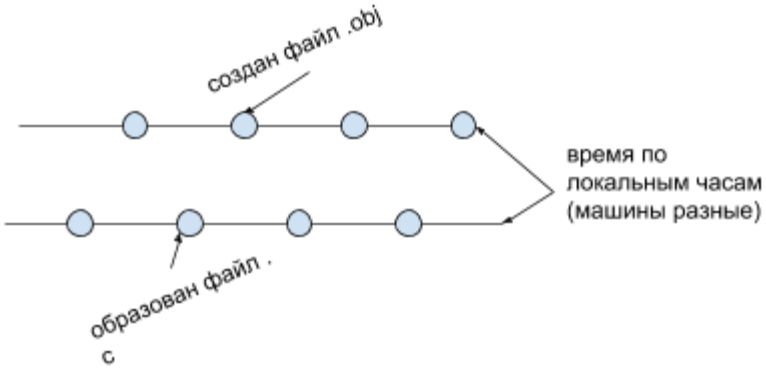
Синхронизация часов

Пример. Рассмотрим работу программы make Unix. Обычно большие программы разбиваются на несколько файлов. Изменения, вносимые в один файл предполагают компиляцию только этого файла.

Программа make после запуска проверяет время последней модификации всех исходных и объектных файлов. Если исходный файл имеет время последней модификации больше, чем время изменения соответствующего ему объектного файла, то программа считает, что исходный файл надо перекомпилировать.

Например.

файл	временная метка
<имя.c>	1224
<имя.obj>	1223



В итоге, мы видим, что по локальным часам время формирования файла с расширением .obj больше, чем исходника. Перекомпиляция не требуется.

Это связано с тем, что каждая машина имеет локальные часы. Это кварцевый генератор, который входит в состав любого компьютера. Они связаны с ЦМОС микросхемой и питаются от энергонезависимой аккумуляторной батареи. В микросхеме имеются часы реального времени. Точность генераторов ограничена. И современные микросхемы имеют точность 10^{-5}. Если таймер генерирует 60 прерываний в секунду, то за час будет сгенерировано 216 тыс. тиков. Это означает, что конкретный компьютер может выдать значение тиков в диапазоне от 215998 до 216002 тиков в час.

Каким образом это учитывать?

1. Подводить часы. Существует служба точного времени.

Для начала определим, что такое время – секунда. Момент подъема солнца на максимально возможную высоту называется солнечным переходом. Интервал между двумя последовательными солнечными переходами называется солнечным днем. Средняя солнечная секунда определена по результатам исследований – mean solar second. В 1948 году были изобретены атомные часы, и была определена атомная секунда – время, за которое атом цезия 133 совершает ровно 9192631770 распадов, которые можно зафиксировать прибором. Международное бюро

усредняет результаты, и выдает глобальное время - TAI International Atomic Time. В нашем мире средний солнечный день в настоящее время увеличивается, а секунда (международная атомная) остается той же, т.е. полдень наступает все раньше и раньше. Бюро решило эту проблему путем использования так называемой потерянной секунды. Всякий раз, когда разница между временем и солнечным временем превышает 800 мс, то вводится эта потерянная секунда. Поэтому весь мир перешел на Universal Coordinated Time UTC.

Есть компании, для которых измерение времени очень важно. ---бла бла про электричество---

Электрически компании положили в основу измерения времени для своих 60 Гц или 50 Гц часов систему UTC. Когда бюро объявляет потерянную секунду, они меняют частоту с 60 на 61 или с 50 на 51.

Институт стандартного времени имеет коротковолновую радиостанцию, которая находится в Fort Collins, имеет позывные WWV. Она широковещательно рассылает короткий импульс в начале каждой UTC секунды. Ее точность составляет +- 1мс.

Соответственно, если есть компании, которые заинтересованы в сигналах точного времени, то их надо получать. Речь идет о локальных сетях. Например, для управления в электрической компании. Оснастить каждый компьютер приемником сигнала точного времени дорого, поэтому приемником оснащается один компьютер, а остальные регулируют свое локальное время в соответствии с тем, который получил тот, у кого приемник.

В поисковике – алгоритмы реального времени – гуглим.

Алгоритмы, которые используются на каждом компьютере для того, чтобы время было установлено правильно:

1. Алгоритм Кристиана
2. Алгоритм Беркли
3. Усредняющие алгоритмы

На сигналы точного времени надо подписываться – платная услуга.

Алгоритм синхронизации логических часов Лампорта

Самый известный алгоритм, который не использует сигналы точного времени – алгоритм синхронизации логических часов Лампорта (мы рассматривали его алгоритм «булочная»). Рассмотрим его.

Важно: в самом общем виде процессы могут взаимодействовать путем передачи сообщений, причем должно выполняться отношение – “сделано до – сделано после”, т.е. имеется несколько компьютеров, которые взаимодействуют, на каждом есть собственный локальные часы. Когда отправляется сообщение, время посылки должно быть меньше чем время получения сообщение. Так ли это, если учитывать локальные часы?

Рассм. (примечание - стрелка идет от строки где написана буква, к след за ней)). Табл справа-алгоритм Лампорта

0		0		0		0		0	0
6	A	8		10		6	A	8	10
12	→	16		20		12	→	16	20
18		24	B	30		18		24	B 30
24		32	→	40		24		32	→ 40
30		40		50		30		40	50
36		48	C	60		36		48	C 60
42		56	←	70		42		61	← 70
48	D	64		80		48	D	69	80
54	←	72		90		70	←	77	90
60		80		100		76		85	100

Для A и B противоречий нет - $t_{отправки} < t_{получения}$. A для C, D - нет, что не согласовывается с правильным порядком событий.

Лампорт предложил следующий алгоритм: приписывать каждому сообщению время отправки по локальным часам процесса отправителя. Процесс-получатель сравнивает время, пришедшее в сообщении с собственным временем. И если собственное время меньше или равно времени, которое пришло в сообщении, то собственное время устанавливается на единицу больше полученного.

Посмотрим реализацию данного алгоритма. В результате такого действия в системе будет правильно идентифицироваться последовательность событий, т.е. будет выполняться отношение “случилось до – случилось после”. Т.е. будет обеспечен правильный порядок событий

Это самый распространенный алгоритм временной синхронизации по логическим часам.

Алгоритмы, обеспечивающие взаимоисключение в распределенных системах

В распределенных системах процессы могут разделять какие-то общие ресурсы, и доступ к этим ресурсам должен обеспечиваться как монопольный. Он обеспечивается методами взаимоисключений.

1. Централизованный алгоритм

Это наиболее очевидный способ взаимного исключения в распределенных системах, который коррелирует с методами, исп. на отдельно стоящих машинах.

В системе взаимодействующих процессов выбирается процесс-координатор – процесс, который выполняется на компьютере с наибольшим сетевым номером. Если какой-то процесс хочет войти в свою критическую секцию, он посылает координатору сообщение–запрос, в котором указывает критическую секцию, в которую он хочет войти, и ждет от координатора сообщение-разрешение

Координатор проверяет, не находится ли какой-либо процесс в этой критической секции. И если нет, посылает сообщение-разрешение. Если некоторый процесс уже находится в данной критической секции, то никакое сообщение процессу, который отправил запрос, не посылается, а его сообщение-запрос ставится в очередь к данной критической секции. И когда другой процесс выходит, то сообщение-разрешение посылается следующему процессу по алгоритму fifo.

Если координатор получил сразу несколько сообщений от разных процессов, которые хотят войти в одну и ту же секцию, то он смотрит, чье сообщение было отправлено раньше, и принимает решение.

Однако, возможна такая ситуация, что процесс-координатор прекратил свое существование – завершился по каким-то причинам. Если какой-либо из взаимодействующих процессов обнаруживает отсутствие координатора (по timeout – устанавливается время возможного ожидания – опасно, потому что есть непредсказуемые задержки, но другого способа нет), то он может инициировать выборы нового координатора. Для этого он посылает другим процессам специальное сообщение-запрос, в котором указывает свой сетевой номер. Если процесс, получивший сообщение о начале выборов нового координатора обнаруживает, что его номер больше, то он посылает назад сообщение подтверждения приёма, а сам инициирует новые выборы – выполняет рассылку сообщений о начале выборов. В результате цепочки действий в качестве координатора будет выбран процесс с наибольшим сетевым номером и работоспособность системы процессов будет восстановлена.

На каждом компьютере должно стоять и ПО обычного и ПО координатора.

Иногда этот алгоритм называют “алгоритмом забияки”.

2. Распределенный алгоритм

Имеется система взаимодействующих процессов, заинтересованных в совместных действиях – использовании совместных ресурсов. Когда процесс хочет войти в свою критическую секцию, он формирует сообщение, в котором указывает имя нужной критической секции, свой номер и текущее значение времени по своим локальным часам, и посылает это сообщение всем другим процессам в этой системе процессов. Т.е рассылает n-1 сообщение. При этом предполагается, что передача сообщения надежна, т.е. получение каждого сообщения сопровождается подтверждением получения. Процесс получивший сообщение, выполняет действия, которые зависят от того, в каком состоянии по отношению к указанной критической секции, он сам находится. Возможны 3 ситуации:

– процесс получатель не находится и не собирается входить в данную критическую секцию. В этом случае он посылает сообщение ответ-разрешение.

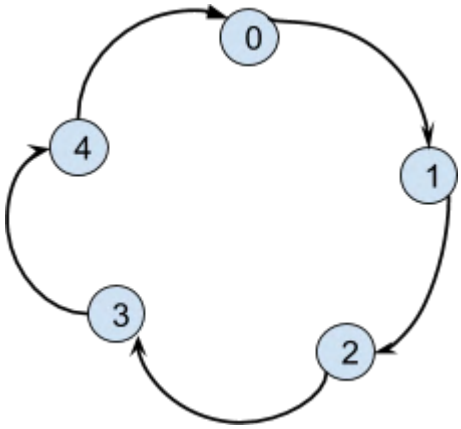
- процесс-получатель уже находится в данной критической секции. Он никакого сообщения не посылает, а ставит пришедший запрос в очередь.

- процесс-получатель сам хочет войти в ту же критическую секцию, но еще не сделал этого. В этом случае он сравнивает временную отметку поступившего сообщения с временной отметкой в собственном запросе, который он разослал другим процессам. Если время в полученном сообщении меньше, т.е. его собственный запрос возник позже, то он посылает сообщение-разрешение, в противном случае – ничего не посылает и ставит пришедший запрос в очередь.

В итоге, процесс может войти в свой критический участок, если он поучит n-1 сообщение-разрешение.

3. Token ring

Процессы образуют логическое кольцо. Каждый процесс знает номер своей позиции в этом логическом кольце, а также номер своего соседа.



Когда кольцо инициируется, процессу с номером 0 передается токен – специальное сообщение. Причем, для каждой критической секции (связана с конкретным ресурсом) будет свой токен. Этот токен циркулирует по кольцу. Переходит от n к n+1 процессу. Сообщение передается по типу P2P (точка в точке). Когда процесс получает токен от своего соседа, он анализирует, не требуется ли ему войти в данную критическую секцию, если требуется – он входит в нее, удерживая токен. Выйдя из критической секции, он посылает токен дальше. Если процесс, получивший токен не заинтересован во вхождении в секцию, то он сразу посылает токен по кольцу. Если ни один из процессов не желает войти в секцию, то токен будет циркулировать по цепочке быстро.

Если хотя бы один процесс прекратил свое существование, то логическое кольцо распадается.

Можно инициировать распределение нового кольца и запускать процесс сначала – это потеря проделанной работы и необходимо формировать точки отката.

Самым надежным является централизованный алгоритм с инициированием нового координатора. Без этого централизованный алгоритм плох.

Неделимые транзакции

Все рассмотренные средства относятся к средствам нижнего уровня. Они требуют определенных знаний, выполнения определенной последовательности действий, о которых пользователь должен иметь представления. Транзакции относятся к средствам более высокого уровня – освобождают программиста от необходимости вникать в тонкости системных вызовов, последовательность действий определенных вызовов.

Транзакция – последовательность операций над одним или несколькими объектами БД, файлами, записями и т.п., которые переводят систему из одного целостного состояния в другое целостное состояние.

Модель неделимой транзакции пришла из бизнеса. Представим себе переговорный процесс – две стороны договариваются, о каких-то поставках. В процессе переговоров условия договора меняются, они могут меняться и уточняться многократно, пока договор не подписан. Любая из сторон может отказаться от договора, но как только договор подписан, сделка должна быть выполнена. Компьютерная транзакция полностью аналогична.

Один процесс объявляет о желании начать транзакцию с одним или более процессами. Процессы могут некоторое время создавать и уничтожать некоторые объекты, выполнять некоторые операции. Затем инициатор объявляет о желании завершить транзакцию. Если все взаимодействующие процессы соглашаются, то результат транзакции фиксируется. Если хотя бы один процесс отказался, или хотя бы один процесс бы аварийно завершен, то все измененные объекты должны быть возвращены в исходное состояние, т.е. в состояние до начала транзакции. Такое свойство называется “все или ничего”. Для реализации транзакции система должна предоставить определенный набор системных вызовов. В самом общем виде это begin transaction, end transaction, abort transaction, read/write.

Транзакции обладают следующим свойством: *упорядоченностью, неделимостью, постоянством*.

Упорядоченность гарантирует, что если 2 или более транзакции выполняются в одно и тоже время, то конечный результат выглядит так, как если бы все транзакции выполнялись последовательно.

Неделимость означает, что во время выполнения транзакции никакой другой процесс не видит ее промежуточные результаты.

Постоянство означает, что после фиксации результатов транзакции никакой сбой не может отменить эти результаты.

Механизм транзакций

Существуют 2 основных подхода реализации механизма транзакций

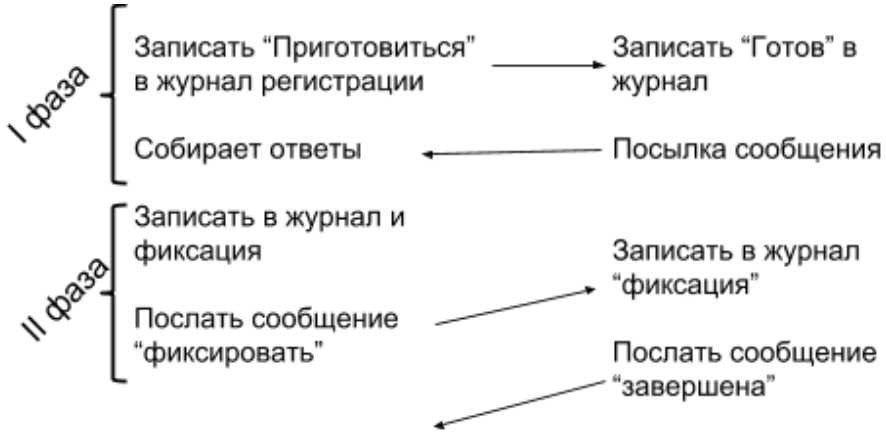
1. Каждый процесс, участвующий в транзакции, работает в индивидуальном рабочем пространстве. Это пространство содержит все файлы и другие объекты, к которым он имеет доступ во время транзакции. Все изменения выполняются только в этом рабочем пространстве. Такой подход требует больших накладных расходов – копирование большого количества данных в это индивидуальное рабочее пространство. Если транзакция прерывается, исходные файлы остаются без изменения, иначе происходит копирование объектов и файлов в систему.

2. Подход называется *списком намерений*.

Модифицируются не копии файлов, а сами файлы, но перед изменением любого файла, части файла – блока, производится запись в специальный журнал – журнал регистрации, который также является файлом. В этом журнале отмечается, какая транзакция делает изменения. Какой файл, или какая запись изменяется, а также старое и новое значение. Только после успешной записи в журнал могут быть сделаны изменения в самом файле или записи. Если транзакция фиксируется, об этом делается запись в журнале, но старые значения измененных данных все равно сохраняются. Если транзакции обрываются, то инфа из журнала регистрации служит для приведения всех файлов записей в исходное состояние. Такое действие называется откатом.

В настоящее время актуальны транзакции в распределенных системах(далее РС). В РС транзакции требуют взаимодействия процессов, которые выполняются на разных машинах. Соответственно у каждой машины имеется собственная память, на каждой машине имеется соответствующий набор файлов, объекты, бд.

Поэтому в РС используется специальный протокол: **протокол двухфазной фиксации транзакций**. Он заключается в следующем: Один из процессов выполняет функцию координатора. Координатор начинает транзакцию. При этом он делает запись о начале транзакции в своем журнале. После этого он рассылает подчиненным процессам, которые выполняют эту же транзакцию, сообщение – подготовиться к фиксации. Подчиненные процессы, получив это сообщение проверяют, готовы ли они к фиксации транзакции. И делают запись в своем журнале. После чего посылают координатору сообщение-ответ “готов к фиксации” и переходят в состояние ожидания сообщения от координатора – “фиксировать”. Если хотя бы один из подчиненных процессов не послал сообщение по каким-либо причинам , то координатор сообщает подчиненным процессам об откате. Если же все процессы готовы фиксировать транзакцию, то сообщение “фиксировать” приводит к тому, что каждый из участвующих в транзакции процессов фиксирует результаты транзакции. Схематически это выглядит след. Образом.



Механизм транзакций поддерживаются низкоуровневыми…???

Мы рассмотрели проблемы взаимодействующих процессов на отдельно стоящей машине и в РС. Вывод: единственным, по настоящему важным отличием РС от централизованных является способ взаимодействия процессов. При этом взаимодействие процессов, или межпроцессное взаимодействие, может осуществляться принципиально 2-мя способами:

1. с помощью использования разделяемой памяти.

2. путем передачи сообщений.

Производство-потребление в РС

Типичным примером является пример производство-потребление. Для системы с общей памятью – предполагается наличие буфера, в который производитель записывает, а потребитель имеет возможность из этого буфера данные прочитать. Такая же задача возможна и в РС, но взаимодействие возможно только путем передачи сообщений.

Концепция такой реализации. Обычно передача сообщений предполагает наличие соответствующих системных вызовов. Чаще всего это send и receive. Send(<кому>, <что>) ; receive(<от кого>, <что>);

producer:

```
        item: int;
        m : message;
        while(1)
        begin
                item = produce_item(); //создание записи
                receive(consumer, &m);//ждем от потребителя пустого сообщения
                build_message(&m, item);//создаем сообщение
                send(consumer, &m);//отправка сообщения
        end;
end;
```

consumer:

```
        item: int;
        m : message;
        for(i = 0; I < N; i++) //отослать N пустых сообщений
                send(producer, &m);
        while(1)
        begin
                receive(producer, &m);//получение сообщения сообщения
                item = extract_item(&m);//Извлечении данных
                send(producer, &m);//посылка пустого сообщения
                consumer_item(&m); //обработка данных
        end;
end;
```

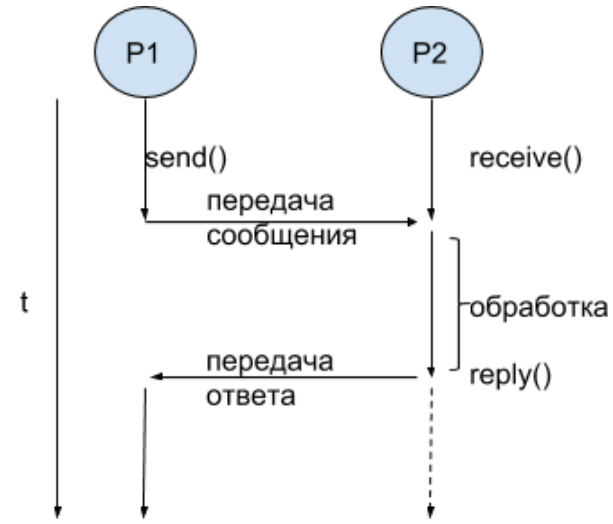
Потребитель начинает с того, что посылает N пустых сообщений производителю. Как только у производителя появляются данные, он берет пустое сообщение, заполняет его и отправляет. Если он работает быстрее потребителя, то все сообщения будут заполнены, производитель должен будет заблокироваться, ожидая когда потребитель прочитает и вернет пустой. Если потребитель быстрее, то все сообщения будут пустыми, он будет блокирован, пока производитель не пошлет сообщение.

Есть варианты посылок

1) каждому сообщению присваивается уникальный адрес, сообщение адресуется процессам.

2) использовать структуру, которая называется “почтовый ящик.” Она представляет собой буфер, для определенного количества сообщений. В этом случае адресуется почтовый ящик. При этом, если 2 взаимодействующих процесса, то должно быть 2 почтовых ящика – потребительский и производителя. Соответствующий производитель отправляет сообщение в почтовый ящик потребителя.

Три состояния блокировки процесса при передачи сообщения

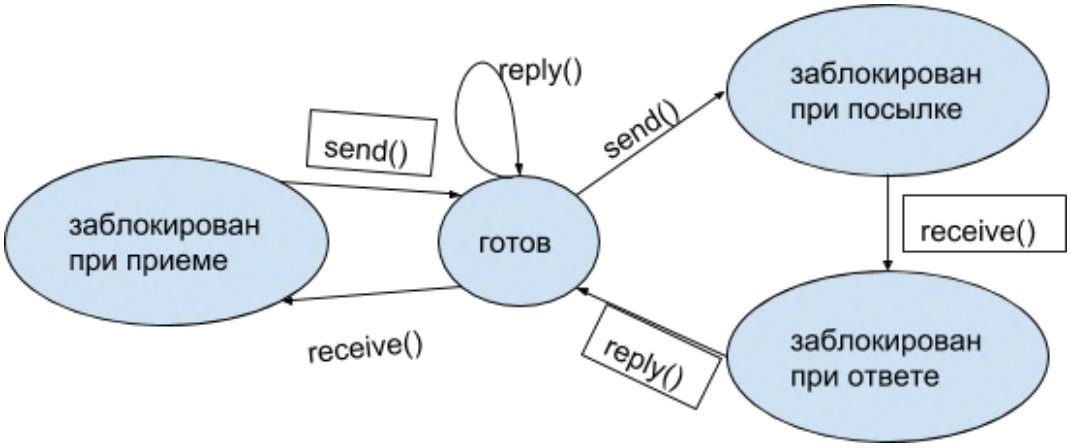


Если обратиться к реализации выше, можно составить диаграмму последовательности событий

reply – тот же send (просто подчеркивают, что это посылка ответа). Мы видим, что это синхронизация. 1-й процесс заинтересован в каких-то действиях от 2-го процесса. Чаще всего 1-й процесс ждет получения-сообщения ответа чтобы продолжить свое выполнение. Все это в самом общем виде отражается в следующей диаграмме.

//В рамочке системные вызовы, выполняемые другим процессом.

Процесс заблокирован при посылке, если он вызвал send, а процесс, которому адресовано сообщение, не готов его принять, т.е. не выполним системный вызов receive. Как только процесс выполнит receive, процесс, пославший сообщение, будет блокирован при ответе и будет находится в этом состоянии до тех пор, пока запрос не будет обработан и не будет вызван системный вызов reply.



Если процесс вызвал receive, а сообщение ему еще не послано, то процесс будет заблокирован при приеме.

При взаимодействии процессов в РС важную роль играют протоколы взаимодействия – соглашение, которое определяет, какие действия процессы выполняют при посылке или приеме сообщение. Например, после приема процесс должен послать сообщение-подтверждение приема.

Обычно взаимодействия процессов выполняется по модели клиент-сервер. В этой модели рассмотрим 2 группы процессов: серверы – процессы, которые предоставляют некоторый сервис, обслуживают другие процессы, и клиенты – процессы, которые нуждаются в некотором обслуживании, которое предоставляется сервером. В некотором смысле обмен сообщениями является универсальным способом взаимодействия – может быть и на одной машине. В частности, на одной машине может быть запущен и сервер и клиенты. Простейший протокол обмена содержит следующие пункты:

1. запрос. Клиент запрашивает сервер для обработки запроса
2. ответ. Сервер возвращает результат операции
3. подтверждение. Клиент подтверждает получение пакета от сервера.

Протокол может быть и расширенный. Для большей надежности обмена возможны следующие пункты:

4. Клиент запрашивает состояние сервера. Сервер доступен?
5. Сервер сообщает – “я доступен”. Данное сообщение показывает, что сервер доступен для обслуживания.
6. Если сервер не доступен – может быть отправлено сообщение – “перезвоните”. Такое происходит, если все ресурсы сервера исчерпаны в текущий момент времени.
7. адрес не верен. Процесс с заданным номером в системе отсутствует.

При этом все сообщения помещаются в так называемые пакеты. Пакет – на нем адрес, внутри содержимое.

При таком развитом протоколе обмена может существовать несколько типов пакетов. На данном примере – 5 типов пакетов. Основными являются первые 3 этапа взаимодействия, остальные повышают надежность обмена.

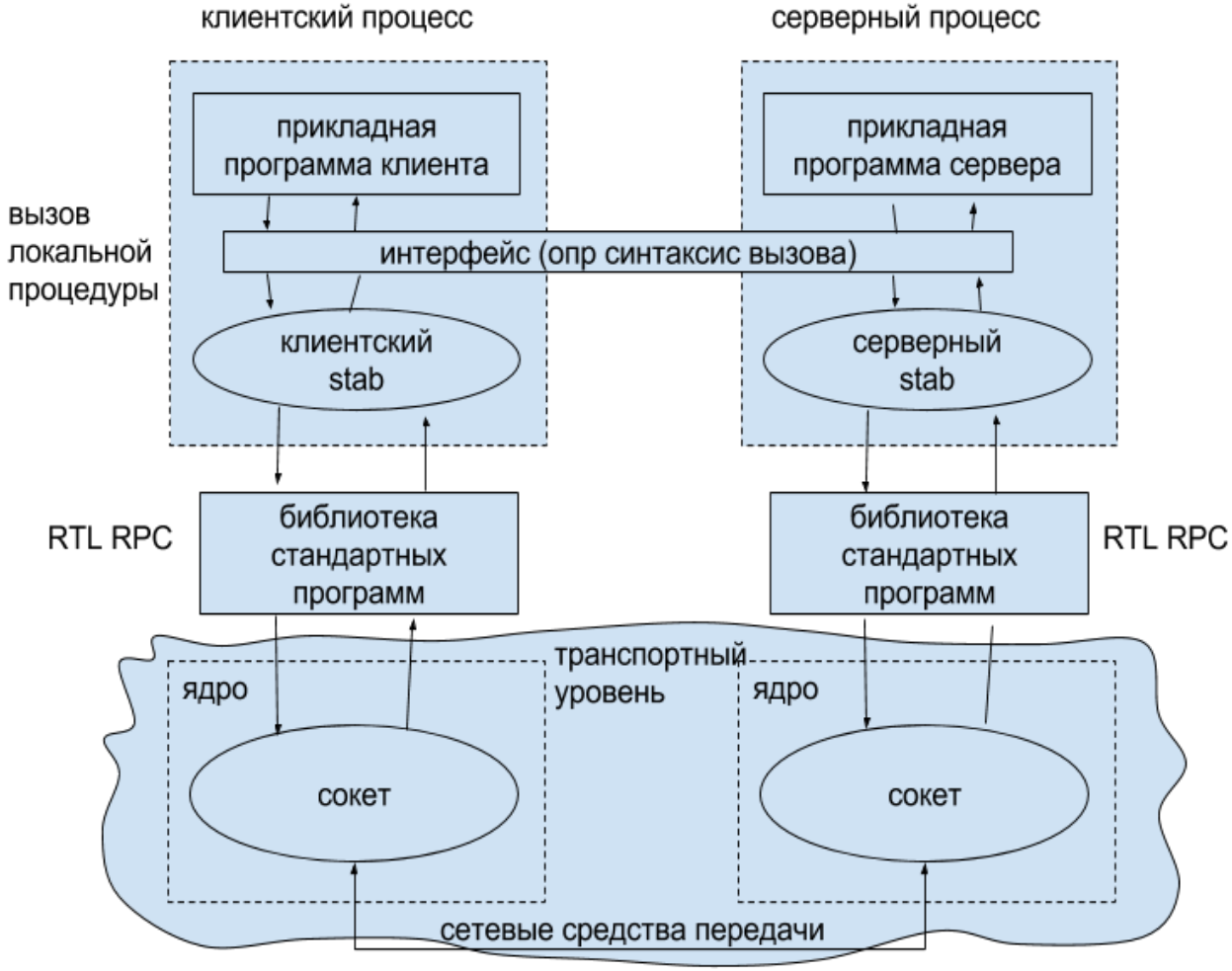
Соответственно надежный обмен приводит ко всем трем состояниям блокировки, причем оценить время блокировок невозможно – это вероятностные вещи. В результате предсказать задержки передачи сообщений в РС невозможно, и это затрудняет точную оценку состояния взаимодействующих процессов в РС. Несмотря на то, что компьютеры могут многое, взаимодействующие процессы не могут ждать бесконечно долго – устанавливается таймаут. Непредсказуемость задержек в результате блокировок делает установку таймаутов весьма проблематичной задачей.

RPC

Remote procedure call – вызов удаленных процедур. Это механизм, с помощью которого один процесс активизирует другой процесс на той же или удаленной машине для выполнения какой-либо функции от своего имени. RPC напоминает вызов локальной функции,а именно процесс вызывает функцию и передает ей данные, а затем ждет, когда функция вернет результат. Специфика RPC заключается в том, что эту функцию выполняет другой процесс, и возможно, на другой машине. Взаимодействие выполняется по модели клиент-сервер. При этом вызывающий RPC процесс - клиент, выполняющий - сервер.

RPC – базовое средство межпроцессного взаимодействия в ОС – низкоуровневое средство. Большинство ОС поддерживают RPC (и Windows, и Unix Linux). При этом RPC различных ОС совместимы, и это позволяет разрабатывать переносимые сетевые приложения.

Механизмы RPC. Как правило, показываются схематически.



Смысл RPC - осуществление взаимодействия с удаленной машиной также, как мы взаимодействуем с обычными подпрограммами, т.е. свести вызов удаленной процедуры к вызову очень похожему на вызов локальной процедуры. Для того чтобы это стало возможным, необходимо определить интерфейс процедуры. Это делается с помощью языка IDL(Interface Definition Language). В Майкрософт это MIDL. Затем выполняется трансляция программы.

Любая платформа, поддерживающая RPC содержит специальный интерфейсный транслятор, который в результате создает ***клиентский stab*** (клиентский переходник). Переходник, или stab, - программа, которая после трансляции присоединяется к программе клиента. В состав клиентского stab-а входят программы поиска сервера, форматирования данных, взаимодействия с сервером, получения ответа.

Форматирование данных на клиентском stab-е заключается в выполнении двух действий, которые называются ***сериализация*** и ***маршалинг***. Маршалинг заключается в перекодировке и упаковке данных, принятой в конкретной системе, в которой выполняется клиент, форматирование сообщения. Сериализация заключается в преобразовании сообщения в последовательность байтов. После того, как клиентский stab выполнил эти действия, начинает работать транспортный уровень и сообщение отсылается серверу.

Серверный stab, или серверный переходник, реализует серверную часть работы и состоит из программ, которые выполняют запрос клиента. При этом зеркально выполняется десериализация и демаршалинг. Затем осуществляется вызов процедуры, которая выполняется на сервере. Аналогично клиентскому stab-у, после трансляции серверный stab присоединяется к программе сервера. При этом клиент может быть написан на одном языке, а сервер на другом, что существенно повышает возможность удаленного взаимодействия.

Связывание

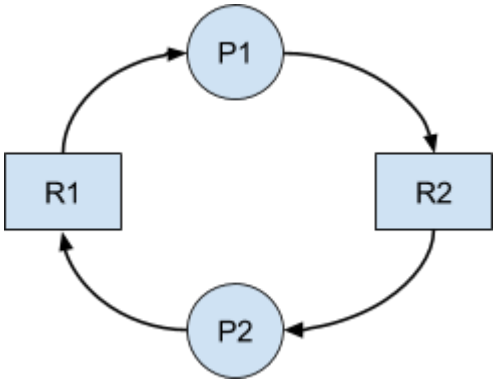
Связывание может быть статическим и динамическим. При статическом связывании информация о сервере, на котором размещена серверная программа, закодирована непосредственно в программе-клиенте. Это может быть например ip-адрес или номер порта. Достоинством статического связывания является его простота. Недостаток заключается в том, что, например, если сервер не работает (завершился), то клиент не сможет получить соответствующий сервис. Если сервер сменил адрес, то клиентская часть должна быть перекомпилирована, так как в ней будет указан новый правильный адрес.

Динамическое связывание, которое и называется binding, повышает гибкость системы, но это требует использование дополнительного слоя, который называется сервером имен и каталогов. Он предназначен для поиска адресов серверов по именам вызываемых процедур.

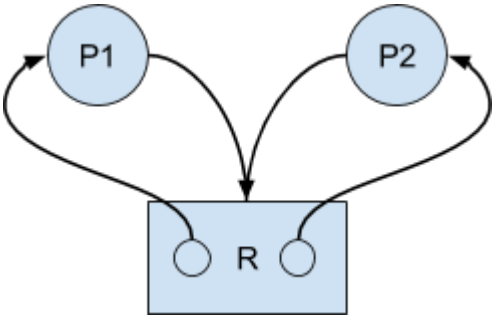
P.S. В общем, это очень общее представление об RPC. Это не параллельное программирование, а основы взаимодействия параллельных процессов. Мы рассмотрели средства взаимоисключения, и в общем познакомились с основными проблемами.

Тупики

Рассм ситуацию: два параллельно выполняющихся процесса (не важно, выполняются квазипараллельно или реально параллельно) выполняются в следующей последовательности. Первый захватил r1, и ему для продолжения выполнения необходим ресурс r2. А второй сначала захватил r2, и ему для продолжения выполнения требуется r1, который занят 1-м процессом. в результате, r1 и r2 занят и ни один из процессов не может их освободить, так как не может продолжить выполнение. В результате, процессы попали в тупик. Тут r1 и r2 - единичный ресурсы (каждый существует в единичном экземпляре). Такой тупик представляется графом.



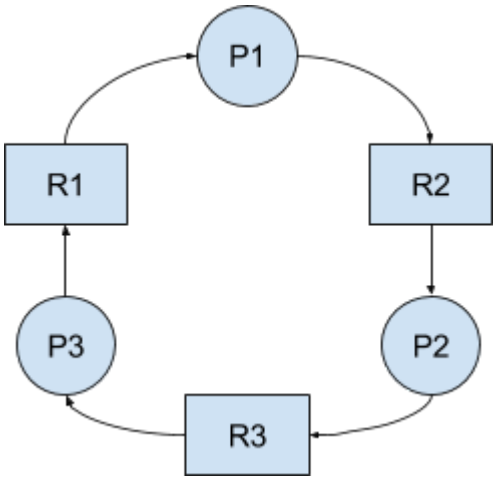
Стрелка от ресурса к процессу называется выделение, а от процесса к ресурсу - запрос. В результате видим замкнутую цепь запросов. Данный граф отображает простейший вариант тупика.



Не обязательно, что тупик возникнет на одиночный ресурсах. Точно такую же ситуацию можно получить, если существует такой же ресурс в количестве 2 шт. Кружочками

обозначены единицы ресурса R. В результате ни один из процессов не может продолжить свое выполнение и освободить единицы ресурсов.

Очевидно, что тупик не обязательно возникает между 2-мя процессами. Их количество зависит от задачи, от системы взаимодействующий процессов. Например,



Тупики представляются с помощью графовой модели Холта. В ее основе лежит двудольный направленный граф.

Тупик, или тупиковая ситуация - это ситуация, которая возникает в результате монопольного использования разделяемых ресурсов, когда процесс, владея ресурсом, запрашивает другой ресурс, или дополнительный ресурс, занятый непосредственно, или через цепочку запросов, другим процессом, ожидающим освобождения ресурса, занятого первым процессом.

Типы ресурсов

Ресурсы с точки зрения особенности их использования делятся на повторно используемые и потребяемые. Повторно используемый ресурс - можно использовать многократно, при этом использование ресурса никоем образом этот ресурс не изменяет. К таким ресурсам относятся:

1. аппаратная часть компьютера - устройства ввода/вывода, память…
2. реентерабельные коды ОС
3. системные таблицы
4. объекты ядра, например, семафоры.

Потребляемые ресурсы - сообщения. Сообщение, которое процесс берет из очереди сообщений перестает существовать. Это могут быть сигналы, сопровождающие некоторые события системы, прерывания, и собственно сообщения, содержащие какую-то информацию, в которой заинтересован некий процесс.

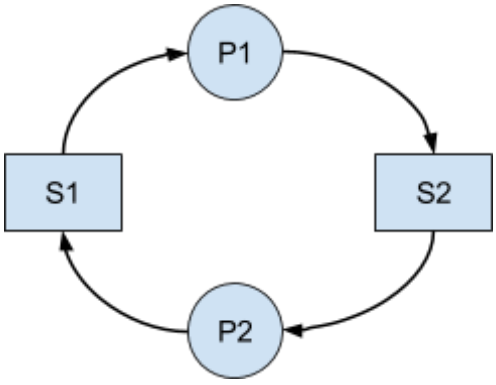
Очевидно, что каждый тип ресурсов обладает дополнительными способностями. В частности, повторно используемые ресурсы имеют следующие свойство: число единиц ресурсов данного типа ограничено и определено. Потребляемые ресурсы в этом смысле отличаются. Число единиц потребляемых ресурсов динамически меняется. Более того, сообщения могут производиться в неограниченном количестве.

Потребление сообщений приводит к уменьшению числа ресурсов. Говоря о сообщениях, можно интерпретировать [ситуацию тупика] следующим образом - процесс p1 ждет сообщения от процесса p3, который в свою очередь ждет сообщения от процесса p2, а процесс p2 ждет сообщения от процесса p1. В результате ни один из процессов не может продолжить свое выполнение, и все эти процессы находятся в тупике.

Тупик на семафорах

Перед тем как перейти к мониторам мы говорили, что появление мониторов было спровоцировано тем, что программы которые оперируют большим количеством семафоров часто попадают в тупиковые ситуации, так как сложно протестировать такие ситуации.

В частности например программа, в которой используется 2 семафора s1 и s2. Запущено два процесса. Первый процесс захватывает и освобождает семафоры в следующем порядке.



P1:...	P2:...
P(S1) - захват семоф.	P(S2)
P(S2)	P(S1)
CR1	CR2
V(S2)	V(S1)
V(S1)	V(S2)

Условия возникновения тупиков

Надо сказать, что у Шоу очень подробно рассмотрена теория тупиков относительно используемых ресурсов.

Рассмотрим повторно используемые ресурсы и определим условия возникновения тупиков. Для возникновения тупика необходимо 4 условия:

1. взаимоисключение (mutual excusion) - когда процессы монопольно используют ресурсы.
2. ожидание (hold and wait) - когда процесс удерживает занятые им ресурсы, ожидая предоставления дополнительного ресурса, для того чтобы иметь возможность продолжить свое выполнение.
3. неперераспредиляемость (no preemption)- когда ресурсы нельзя отобрать у процесса до его завершения или до того момента, когда процесс сам освободит занимаемые ресурсы.
4. круговое ожидание (circular wait) - когда возникает замкнутая цепь запросов,в которые каждый процесс занимает ресурс, необходимый другому процессу для продолжения выполнения.

Методы борьбы с тупиками

//тупик становится проблемой когда он возникает.

Существует три основный метода борьбы с тупиками:

1. недопущение тупиков - в системе создается такая ситуация, при которой возникновение тупиков невозможно.
2. обход тупиков, или предотвращение - связан с предварительным анализом ситуации, который позволяет обойти тупиковую ситуацию.
3. обнаружение тупиков, т.е. тупики возможны, они возникают, а когда они возникли, то их необходимо обнаружить. Это значит, что необходимо найти процессы которые попали в тупик.

Обнаружение непосредственно связано с восстановлением работоспособности системы (имеется в виду не вычислительная система, а система процессов). Это особенно актуально в распределенных системах. Понятно, что необходимо восстановить работоспособность такой системы, поэтому необходимо обнаружить процессы попавшие тупик и принять соответствующие меры.

Этот метод получил название стратегии Хавендера, который показал, что возникновение тупика невозможно, если нарушено хотя бы одно из условий возникновения тупика.

3 основных способа исключения самой возможности возникновения тупика:

1. Опережающее требование

Способ заключается в том, что процесс до своего выполнения запрашивает все потенциально необходимые ему ресурсы. т.о. устраняется условие ожидания. Это крайне неэффективный процесс - задерживает ресурсы задолго до их его реального использование, а некоторые вообще не используются в некоторой траектории выполнения процесса. Также такой подход требует знаний процессами их потребности в ресурсах. Системы слишком разнообразны, но такая система должна быть хорошо детерминирована, в которой о процессах известно все - это какие-то специальные системы.

Данный подход не исключает бесконечного откладывания.

Для более эффективного использования ресурсов можно задания разделить на более мелкие задачи, и выделять ресурсы каждой отдельной задаче.

Вывод: все это требует знания какой задаче какие ресурсы потенциально необходимы, и еще это связано с бесконечным откладыванием. Его можно устранить, если в системе есть достаточное количество необходимых ресурсов, что возможно в очень специальных системах.

2. Иерархическое распределение.

Этот подход предполагает упорядочивания ресурсов. Ресурсы делятся на классы, которым присваивается номер. При этом процессы могут запрашивать ресурсы, принадлежащие классам с большими номерами, чем номера классов ресурсов, которыми они уже владеют. Этот подход устраняет круговое ожидание.

Если процессу при выполнении потребуется ресурс с номером класса меньшим, чем номер класса ресурса, который он удерживает, то процесс должен освободить занимаемые ресурсы и запросить ресурсы в правильном порядке. При этом если процессу требуются ресурсы с номерами классов в порядке, обратном их нумерации в системе, то это в итоге приведет к первому случаю, т.е. в итоге своих многократных действий процесс запросит все

необходимые ему ресурсы сразу. Назначение номера классу ресурсов должно отражать наиболее вероятный порядок использования процессами ресурсов. Обычно самые дефицитные ресурсы имеют наибольшие номера.

Не смотря на то, что данных подход решает проблему тупиков, часто бывает невозможно определить порядок нумерации ресурсов, который удовлетворял бы все выполняемые процессы, поскольку ресурсы в системе слишком разнообразны. Это и системные таблицы и область paging swaping, это объекты ядра (средства, предоставляемые процессу ядром - пока рассмотрели только семафоры, но еще есть очередь сообщений, разделяемая память и т.д.). Т.е. число потенциальных ресурсов и вариантов их использования слишком большое и их систематизация затруднительна. В каких-то системах данный подход может использоваться...

Некоторые системы в тупики попадать не могут. Например. системы реального времени - управляют внешними по отношению к вычислительной системе объектами и процессами. Система, управляющая ядерным реактором не попадет в тупик.

3.

Способ связан с тем, что если процесс не может получить нужный ему ресурс, он освобождает занятые им ресурсы. Это весьма проблематичный подход, потому что освобождение ресурса - непростая задача - откат - процесс должен вернуться к состоянию до запроса ресурса, которые он удерживает. Для этого нужно сделать точки отката. Эти точки фиксируются в памяти машины и запоминается информация для того, чтобы вернуть процесс в предыдущее состояние. Или прописываются обратные действия, что тоже не тривиально. Способ требует дополнительный накладных расходов. Он неприемлем в системах реального времени, потому что неизвестно, сколько процесс потратит времени на такие действия, но более того, в результате процесс

Тем не менее, такой подход допускается.

Обход тупиков.

Если в первом подходе тупики невозможны, то здесь тупик возможен, но предпринимаются дополнительные действия для того, чтобы тупик не возник. Самым известным, самым обсуждаемым алгоритмом обхода тупиков является алгоритм банкира. Предложен Дейкстрой. Он выполняет действия, похожие на действия банкира.

У банкира есть деньги. Он хочет больше. При этом банкир должен знать, кому выдать заём, а кому отказать. Выдает тому, кто может вернуть, да еще и с процентами. Возникает ситуация, когда заемщик может вернуть только в том случае, если получит дополнительную ссуду?

У нас в качестве банкира выступает менеджер ресурсов. Заемщики - процессы, которые заявляют свои потребности в ресурсах. Заявки процессов отражают максимальную потребность каждого из процессов в ресурсах каждого класса.

Алгоритм банкира имеет несколько существенных ограничений.

1. Процесс должен подать заявку. В ней он должен указать максимальное количество потенциально необходимых ему ресурсов. При этом требуется выполнение следующих условий:

- процесс не может запросить при выполнени больше ресурсов, чем указано в его заявке.
- в заявке процесс не может указать большее количество ресурсов данного класса, чем имеется в системе.
- количество всех распределенных ресурсов каждого класса не может превышать общее количество единиц ресурсов каждого класса.
- число процессов ограничено, т.е. задача решается для опр числа процессов.

Менеджер ресурсов гарантирует, что тупиковая ситуация не возникнет, и проверяет каждый запрос по отношению к количеству свободных единиц запрошенного ресурса в системе. Этот анализ делается на основе заявки процесса. В заявке процесс указывает свою максимальную потребность в каждом виде ресурса. Менеджер процессов анализирует количество свободных единиц запрошенного ресурса, количество уже полученных процессом единиц данного ресурса и максимального количества единиц данного ресурса, которые он указал в заявке. В результате этого анализа он удовлетворяет только запросы тех процессов, которые могут гарантированно завершиться, получив нужное количество единиц ресурса и это количество свободных единиц ресурса у системы имеется.

Пример на следующей лекции.

В процессе работы процесс не может потребовать единиц ресурсов каждого типа больше, чем указано в его заявке. Очевидно, что если просуммировать заявленные потребности в единицах ресурса каждого типа, то в сумме они не

должны превышать количества ресурса данного типа в системе. Имея такую информацию менеджер ресурсов может анализировать текущую ситуацию. Изменение ситуации происходит в результате запроса процессов. Когда процесс делает запрос, менеджер ресурсов анализирует, приведет данный запрос к тупиковой ситуации, или нет. Ситуация в системе называется безопасной относительно тупика, если можно выстроить такую последовательность, что первый процесс в этой последовательности сможет завершиться, даже если он полностью выберет ресурсы по своей заявке. Завершившись, процесс освободит занимаемые единицы ресурса, и в результате второй процесс последовательности сможет завершиться, даже если он выберет полностью все ресурсы по своей заявке. И т.д. В результате такой цепочки действий i-й процесс сможет завершиться, если успешно завершились i-1 процессы и вернули системе занимаемые ими ресурсы, что позволит i-му процессу завершиться нормально, даже если он затребует то число ресурсов, которое он указал в своей заявке. Если такую последовательность выстроить нельзя, то ситуация называется небезопасной относительно тупика. Рассмотрим иллюстрацию

<table>
<tr><th>Процессы</th><th>Текущее распределение</th><th>Свободные единицы ресурсов</th><th>Заявка</th><td rowspan="10">Показанная ситуация является безопасной относительно тупика. Так как имеется последовательность процессов, которые могут гарантированно завершиться, даже если они затребуют полностью все единицы ресурса, которые они указали в своей заявке. В самом деле, процесс P2 сможет гарантированно завершиться, так как ему максимально может потребоваться 2 единицы ресурса, 3 он удерживает, 5 указано в заявке, значит для его гарантированного завершения ему может потребоваться еще 2 единицы ресурсов. Эти единицы у системы есть.</td></tr>
<tr><td>P1</td><td>1</td><td rowspan="3">2</td><td>4</td></tr>
<tr><td>P2</td><td>3</td><td>5</td></tr>
<tr><td>P3</td><td>5</td><td>9</td></tr>
<tr><td colspan="4">следующее состояние:</td></tr>
<tr><td>P1</td><td>2</td><td rowspan="3">1</td><td>4</td></tr>
<tr><td>P2</td><td>3</td><td>5</td></tr>
<tr><td>P3</td><td>5</td><td>9</td></tr>
</table>

Значит, 2-й процесс сможет гарантированно завершиться. Завершившись, он освободит занимаемые им ресурсы, и освобожденные единицы ресурсов в сумме с имеющимися у системы свободными единицами ресурса смогут удовлетворить максимальную потребность первого процесса. 3-й процесс также сможет гарантированно завершиться.

Т.е. можно построить такую последовательность процессов, каждый из которых сможет гарантированно завершиться, т.е. данная ситуация безопасна относительно тупика. Однако, то что текущее состояние надежно относительно тупика, абсолютно не означает, что все последующие состояния также будут надежны. Например, если система из данного состояния перейдет в следующее (2), то из надежного состояния система переходит в ненадежное.

Система удовлетворяет запрос процесса на дополнительный ресурс, только тогда, когда ее состояние после выделения этого ресурса остается надежным. Хотя, ненадежное состояние не обязательно может привести к тупику в силу того, что процессы могут не затребовать максимально потребного числа ресурсов, т.к. при данной реализации указанное число в заявке ресурсов может не потребоваться.

В соответствии с данным алгоритмом, всякий раз, когда процесс делает запрос на необходимый ему ресурс, менеджер ресурсов должен проанализировать ситуацию и найти такую последовательность процессов, в которой каждый процесс может гарантированно завершиться. Только в этом случае запрос процесса может быть удовлетворен. В результате необходимо исследовать n! последовательностей прежде чем можно будет признать состояние системы безопасным или небезопасным. Из всех ограничений и из анализа затрат, которые можно произвести для реализации данного алгоритма следует, что данный алгоритм имеет скорее теоретическое значение. т.е. Дейкстрой было показано, что при определенных условиях можно обойти возникновение тупиков. (ограничения слишком серьезные, хотя вполне возможно, что может и существовать такая система с ограниченным числом процессов, если они хорошо детерминированы. Тогда в системе будет информация о том, сколько ресурсов может потребовать каждый процесс. Но даже если такая система существует, то затраты на выявление безопасной или небезопасной ситуации огромные - n!).

Мы говорили, что существуют системы, которые попадать в тупик не могут. Очевидно, что не всегда можно применить рассмотренные выше методы. В современных системах процессы создаются по мере необходимости,

ресурсы выделяются по мере надобности в результате запросов. Все выполняется динамически, при этом процессы не знают своей максимальной потребности ресурсов.

Тупики в системах реального времени. Алгоритмы.

Системы реального времени - системы, которые управляют внешними по отношению к вычислительной системе системами или процессами. Очевидно, что такие системы не должны попадать в тупики. В каждом конкретном случае это решается для каждой системы реального времени индивидуально. Поэтому алгоритм Дейкстры имеет теоретическое значение - он показал, что действительно можно обойти тупик. Для того, чтобы сократить временные затраты на анализ ситуации в системе предлагаются еще ряд алгоритмов. Алгоритм Хабермана очень часто упоминается в учебниках.

Рассмотрим алгоритм, который называется **O(n^2)** //OrderOf??. Отсюда следует, что затраты в системе сводятся к O(n^2)

S - количество процессов

цикл пока S<>[]

начало

найти процесс A в посл-ти S, который может завершиться;

если нет, то состояние небезопасное - вывести процесс A из S: отобрать у A ресурсы и добавить их в пул свободных ресурсов;

конец;

состояние безопасное;

Коммент непонятно к чему: для систем общего назначения не очень важно, попадут они в тупик, или нет. Тупиковая ситуация важна для каких-либо обслуживающих систем, которые не обязательно являются системами реального времени. Они должны обслуживать за гарантированный интервал времени. В вычислительных системах есть какие-то общие подходы, но для конкретной системы строится конкретное решение.

Алгоритм Габермана(Хабермана)

Менеджер ресурсов поддерживает массив S[0, r-1], где r - число ресурсов одного типа.

S[i] = z - i для всех i: 0 <= i < r

если процесс, заявивший Claim единиц ресурса и удерживающий hold единиц ресурса, запрашивает единицу ресурса, то S[i] декрементируется для всех i: 0 <= i <= Claim_hold.

В результате, если какой-то из S[i] < 0, то состояние опасное.

Как видим, в этом подходе, который называется обход тупиков, выполняются все 4 условия возникновения тупика, т.е. тупики в принципе возможны. Процессы монопольно используют разделяемые ресурсы, процессы удерживают ресурсы и запрашивают дополнительные ресурсы, но при этом проводятся соответствующие действия, которые анализируют состояние системы. У процессов ресурсы могут не отбираться, но процесс, запрос которого признан небезопасным, может быть переведен в состояние ожидания до более благоприятного момента в системе. Система удовлетворяет только те запросы, при которых ее состояние остается безопасным.

3-й подход - обнаружение тупиков.

Тупики возможны - все 4 условия возникновения тупика существуют, процессы создаются по мере необходимости, ресурсы выделяются по мере надобности, т.е. все динамически, никаких ограничений. И в этом случае возникает задача обнаружения тупика.

Соответственно, для обнаружения тупика используются специальные методы. Это графы. Для описания состояний системы используются двудольный направленный граф. Графовая модель, которая описывает состояние системы называется моделью Холдта. Это двудольный, или бихроматический, направленный граф. В этом графе имеются 2 непересекающихся множества вершин: вершины соответствующие процессам, выполняемые в системе, и вершины, соответствующие ресурсам, имеющимися в системе. Вершины из двух непересекающихся множеств соединяются дугами. При этом никакая дуга не соединяет вершины одного множества. процессы обозначим буквой P, ресурсы буквой R.

$R = \{r_1, ..., r_m\}$ |

| X = R U P

$P = \{p_1, ..., p_n\}$ |

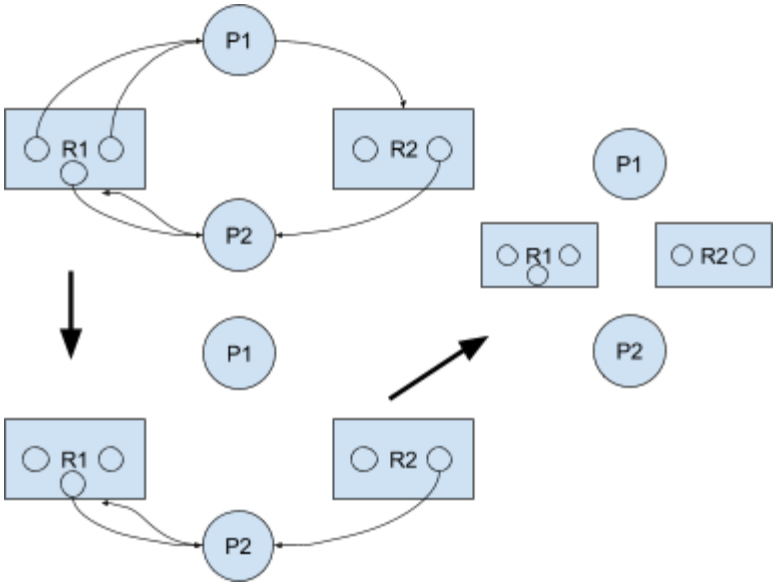
Дуга (r, p) из вершины ri к вершине pj - приобретение или получение.

Дуга (p, r) из вершины pj к вершине ri - запрос

Речь идет не о вычислительной системе в целом, а о системе взаимодействующих параллельных процессов.

Такая модель позволяет обнаружить тупик. Обнаружение тупиков выполняется методом редукции графа.

Система находится в тупике, если она не может поменять своего состояния, или, другими словами, процесс находится в тупике в некотором состоянии, если он не может изменить это состояние ни в результате запроса и последующего получения ресурса, ни в результате освобождения занимаемого им ресурса. Или наоборот, процесс не попавший в тупик, не заблокированный в тупике, может приобретать любые ресурсы, нужные ему для продолжения выполнения и освобождать ресурсы, которые стали ненужными. Освободившиеся ресурсы могут быть распределены другим процессом в системе. Редукция графа выполняется путем удаления ребер, при этом граф сокращается процессом pi, если тот не является ни заблокированной, ни изолированной вершиной путем удаления всех ребер, как входящих в pi, так и выходящих и pi. Рассмотрим пример.

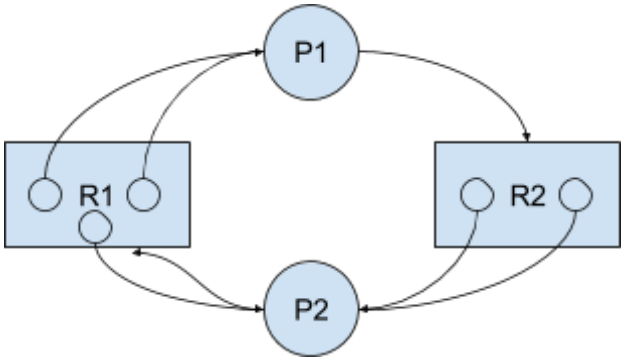


Данный граф иллюстрирует полностью сокращаемый граф. Граф может быть сокращен по вершине p1, т.к. p1 запрашивает ресурс r2, и эта единица у системы есть, значит p1 может получить эту единицу ресурса. В результате своего выполнения он сможет освободить занимаемые им ресурсы 1.

В результате освобождения процессом p1 занимаемых им единиц ресурсов, процесс p2 может продолжить свое выполнение, т.е. данный граф сокращается и по вершине p2.

В итоге, все вершины графа стали изолированными. Фундаментальной работой по теории тупиков является работа Шоу. ей посвящена целая графа в книге "логическое проектирование ОС" . Там приводится ряд теорем и лемм с доказательствами.

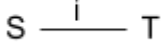
Теорема 1 -Граф является полностью сокращаемым, если существует такая последовательность сокращений, которая устраняет все дуги. Если граф нельзя полностью сократить, то анализируемое состояние является тупиковым.



Рассмотрим пример тупиковой ситуации (4), т.е. ситуации, при которой процесс не может получить запрашиваемый им ресурс. Каждый из процессов занимает единицы ресурсов, которые необходимы другому процессу для продолжения выполнения. Имеется замкнутая цепь запросов, которые не могут быть удовлетворены. т.е. данная иллюстрация показывает ситуацию, в которой граф не может быть редуцирован не по вершине p1 ни по вершине p2, т.е. запросы этих процессов не могут быть удовлетворены.

Теорема 2 ????Цикл в графе повторно используемых ресурсов является необходимым условием тупика. Под циклом понимается замкнутая цепь запросов.

Теорема 3 Замкнутая цепь запросов может возникнуть только в результате запроса.



Если s не является состоянием тупика, и переход от s к t, где t является состояние тупика, то операция, которая переводит систему в состояние t есть запрос. p_i - запрос

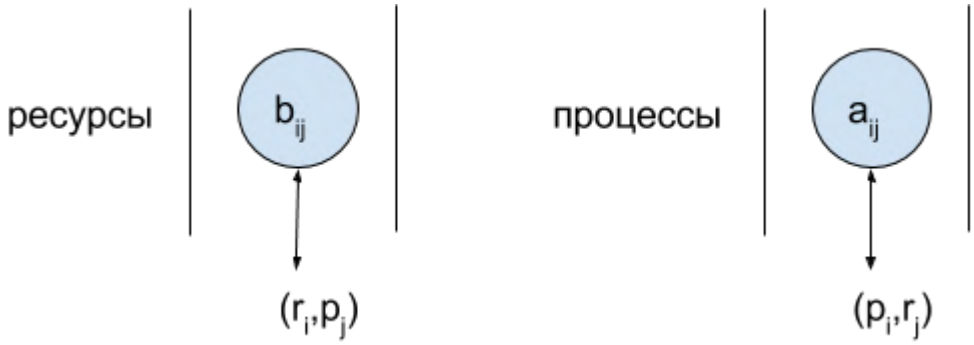
Отсюда следует: т.к. тупик может наступить только в результате запроса, то редукцию следует выполнять только если выполнен запроса некоторого ??

Графы позволяют решать поставленную задачу. (исследуем теорию тупиков для повторно используемых ресурсов.) Встает вопрос, как представить граф для того, чтобы можно было написать ПО, обнаруживающие тупик.

Представление графа и алгоритмы обнаружения.

Граф двудольный, направленный, может быть представлен в виде двух матриц - матрицы текущего распределения, и матрицы запросов. Или с помощью соответствующих связных списков. Матрицы более наглядны, а связные списки позволяют легко добавлять новые процессы. Матрица текущего распределения.

B = {r, p} A = {r, r}



Матрица текущего распределения(B) отражает количество единиц каждого ресурса выделенного определенному процессу. Матрица запросов(C) отражает количество запрошенных единиц ресурсов rj, процесса pi.

\\новая лекция

Необходимо представить систему в виде, доступному для проведения детального анализа. Таким видом являются матрицы или связные списки. На матрицах это более просто показать, поэтому мы будем рассматривать их.

Двудольный граф показывать будем на 2-х матрицах - матрице текущего распределения - отражает количество единиц ресурса, выделенного процессу и матрицей запросов - отражает количество единиц ресурса, которые запрашивает процесс.

Существует несколько общих подходов для реализации обнаружения тупиков. Смысл действия - выполнить редукцию графа - можно сократить дуги запросов, если эти запросы могут быть удовлетворены. В результате сокращения графа мы получим либо полностью сокращаемый граф, в котором все вершины являются изолированными (вершинами являются процессы и ресурсы). Это является признаком того, что система в тупике не находится. Если же в результате редукции остались несокращенные вершины, то значит есть процессы, которые попали в тупиковую ситуацию, по матрице ресурсов можно посмотреть, какие ресурсы запросили процессы.

Методы обнаружения тупиков

Метод прямого обнаружения. В этом методе по порядку просматривают матрицу запросов. Там, где это возможно, производятся сокращения дуг графа. До тех пор, пока нельзя будет сделать ни одного сокращения, процессы, оставшиеся после всех сокращений, находятся в тупике.

В самом худшем случае, когда сокращения выполняются в обратном порядке следования процессов, число проверок будет n*(n+1)/2, при этом каждая проверка требует испытания m ресурсов. Таким образов затраты m*(n^2).

Второй подход (без названия) - более эффективный алгоритм. Эффективность достигается за счет хранения дополнительной информации о запросах, а именно, для каждого ресурса хранятся запросы, упорядоченные по размеру. Для каждого процесса заводится счетчик ожидания, который содержит число типов ресурсов, которые вызвали блокировку процесса в ожидании освобождения запрошенного ресурса. Затем среди заблокированных процессов ищется цикл запросов, т.е. замкнутая цепь запросов.

Еще одним алгоритмом можно считать следующий - когда к матрице запросов и матрице распределений добавляется еще вектор свободных ресурсов(F). F=[...fj...], fj - количество свободных единиц j-го ресурса. При этом если просуммировать все выделенные единицы j-го ресурса: суммируем по процессам j-ый ресурс + свободные единицы - в итоге получаем количество единиц ресурса в системе.

$\sum_{i=1}^{n} b_{ij} + f_j = r_j$ – количество всех единиц ресурса в системе

Алгоритм основан на сравнении векторов. Пусть имеется два вектора C и D. Отношение C <= D означает, что каждый элемент вектора C меньше либо равен соответствующему элементу вектора D: $C \le D : c_i \le d_i$ для $1 \le i \le n$

Тогда если i-ый процесс запрашивает j-ый ресурс, и при этом строка запросов i-го процесса меньше либо равна вектору F, то такой запрос может быть удовлетворен. Процесс, запрос которого может быть удовлетворен, может завершиться и освободить занимаемые им ресурсы. Т.е. граф может быть сокращен по этому процессу. Рассмотрим пример.

Распределение ресурсов. Всего ресурсов имеется R = {6, 9, 3}. Из матрицы получаем количество свободных F = {4, 0, 2}

Матрица распределения ресурсов				Матрица запросов			
p/r	1	2	3	p/r	1	2	3
1	0	1	1	1	1	1	0
2	1	3	0	2	0	0	1
3	1	5	0	3	1	1	2
всего	2	9	1				

Второй процесс имеет строку запросов, меньшую или равную вектору F, следовательно, его запрос может быть удовлетворен. Можно обнулить эту строку. F = {5, 3, 2}

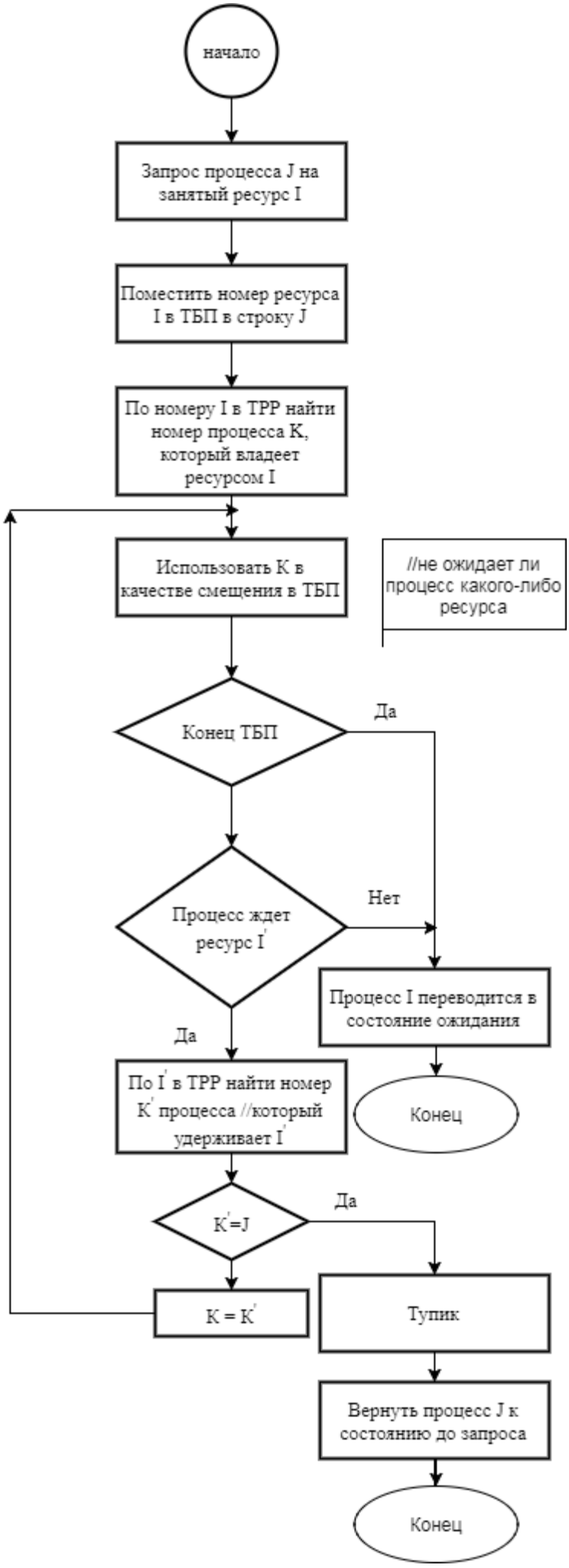
p/r	1	2	3		p/r	1	2	3
1	1	1	0	=>	1	0	0	0
2	0	0	0		2	0	0	0
3	1	1	2		3	1	1	2

Далее тоже самое с первым процессом F = {5, 4, 3}. А потом с третьим. F = {6, 9, 3} . (матрицы полностью обнулится)

Есть еще пример обнаружения тупиков в книге Медника-Донована. Алгоритм предложен Бенсусан и Мерфи. Используется матрица распределения и матрица процессов, ожидающих ресурсы, т.е. матрица блокированных процессов (ТБП). Эти матрицы определяются таблицами. Таблица распределенных ресурсов(ТРР)

ТРР		ТБП	
Ресурс	Процесс	Процесс	Ресурс
1	1	1	3
2	3	2	2
3	2	3	
4	2		
5	1		

Работа алгоритма начинается в тот момент, когда процесс запрашивает занятый ресурс. Предыдущий пример оперировал несколькими типами ресурсов в разном количестве, а этот реализован только на единичных ресурсах - т.е. каждого ресурса одна единица. Поэтому если ресурс занят, то запрос j-го процесса не может быть сразу удовлетворен.


начало
Запрос процесса J на занятый ресурс I
Поместить номер ресурса I в ТБП в строку J
По номеру I в ТРР найти номер процесса K, который владеет ресурсом I
Использовать K в качестве смещения в ТБП
//не ожидает ли процесс какого-либо ресурса
Конец ТБП
Да
Процесс ждет ресурс I'
Нет
Процесс I переводится в состояние ожидания
Да
По I' в ТРР найти номер K' процесса //который удерживает I'
Конец
K'=J
Да
K = K'
Тупик
Вернуть процесс J к состоянию до запроса
Конец

Блок -схема

Пример. Пусть выполняется следующая последовательность действий -

1. процесс P1 занимает ресурс R1. Этот факт должен быть отражен в соответствующей таблице
2. процесс P2 занимает ресурс R3
3. процесс P3 занимает ресурс R2
4. процесс P2 занимает ресурс R4
5. процесс P1 занимает ресурс R5 // в результате все ресурсы заняты
6. процесс P1 запрашивает R3 //обозн. p1→r3

   3-й ресурс занят, начинает работать алгоритм. 1-й процесс запрашивает, следовательно, мы должны поместить номер ресурса 3 в строку с номером 1. Какой процесс занимает 3-й ресурс - процесс с номером 2. K = 2.
7. процесс P2 запрашивает R2. //обозн. p2→r2

   R2 занят, мы должны поместить в строку 2 2-й ресурс. по TPP можем посмотреть, какой процесс занимает 2-й ресурс. K = 3. Процесс P3 не блокирован, соответственно можно спокойно заблокировать 2-й процесс на 2-м ресурсе.
8. процесс P3 запрашивает ресурс R5. //обозн. p3→r5

   Его удерживает P1. 1-й процесс блокирован на 3-м ресурсе. i' = 3. R3 удерживает 2-й процесс. K’ = 2. проверка условия j == K’. выполняем присваивание K присвоить 2. Далее определяем, что 2-й процесс ждет 2-й ресурс. по TPP находим, что он занят 3-м процессом, следовательно K’ равно 3. условие j = K’ выполняется. Следовательно у нас тупик.

Это ограниченное решение на единичных ресурсах - просто симпатичный пример.

Посколько тупик может возникнуть только в результате запроса. Мы записали теорему, что если в в системе требуется именно защита от таких ситуаций, то необходимо выполнять непрерывное обнаружение - анализировать каждый запрос. Соответственно, если запрос приводит к тупику, то такой запрос выполнен быть не может и необходимо выполнить откат. Обычно процесс блокируется на некоторое время. Данный пример демонстрирует интересные возможности обнаружения тупиков и сам этот процесс. Как мы сказали, обнаружение тупиков связано с восстановлением работоспособности системы. Если в системе необходимо производить постоянный анализ - это одна ситуация. Могут быть другие ситуации, когда подобный анализ проводится по таймауту. Есть система взаимодействующих процессов. и в этой системе нет никакого движения какое-то время. Тогда может быть включен алгоритм обнаружения тупиковой ситуации. Если тупик обнаружен, то необходимо восстановить работоспособность системы. Самый простой способ - перезапуск - потеря всей проделанной работы.

Возможно два подхода разрешения тупиков.

Первый - завершение работы процессов, которые попали в тупик. При этом завершение выполняется последовательно. В тупик могут попасть несколько процессов, может возникнуть довольно длительная цепочка запросов, и не обязательно убивать все. Ресурсы, которые занимают данные процессы возвращаются системе. Их может оказаться достаточно для того, чтобы другие процессы, попавшие в тупик, могли продолжить свое выполнение.

Второй - перехватывать ресурсы у тех процессов, которые в тупик не попали. Сложно отобрать у процесса уже выделенный ему ресурс. Это связано с откатом - возвращению процесса в состояние до выделения ему ресурсы. Естественно, процессы, у которых отбирается ресурс могут выбираться по разным критериям, например, выбираться самые низкоприоритетные. Но сначала желательно убедиться, владеет ли он нужными ресурсами.

Определим цену вопроса. Выбор процессов для уничтожения определяется наименьшей ценой уничтожения такого процесса. Ценой прекращения может быть:

1. Приоритет процесса
2. Цена повторного запуска процесса до точки, в которой его работа была прекращена.
3. Внешняя цена - она основывается на простых отношениях?? процесс, запущенный кем-то???? зависит от многих обстоятельств.

Для ресурсов характерно знания количества - их количество в системе постоянно - для повторно используемых ресурсов.

Такая же задача может возникнуть в системе с потребляемые ресурсам. У Шоу такие системы классифицируются:

1. Производители известны, потребители известные. Такая система сводится к системе с повторно используемыми ресурсами.

2. Производители известны, а потребители нет, или наоборот.

3. Не известны ни производители, ни потребители.

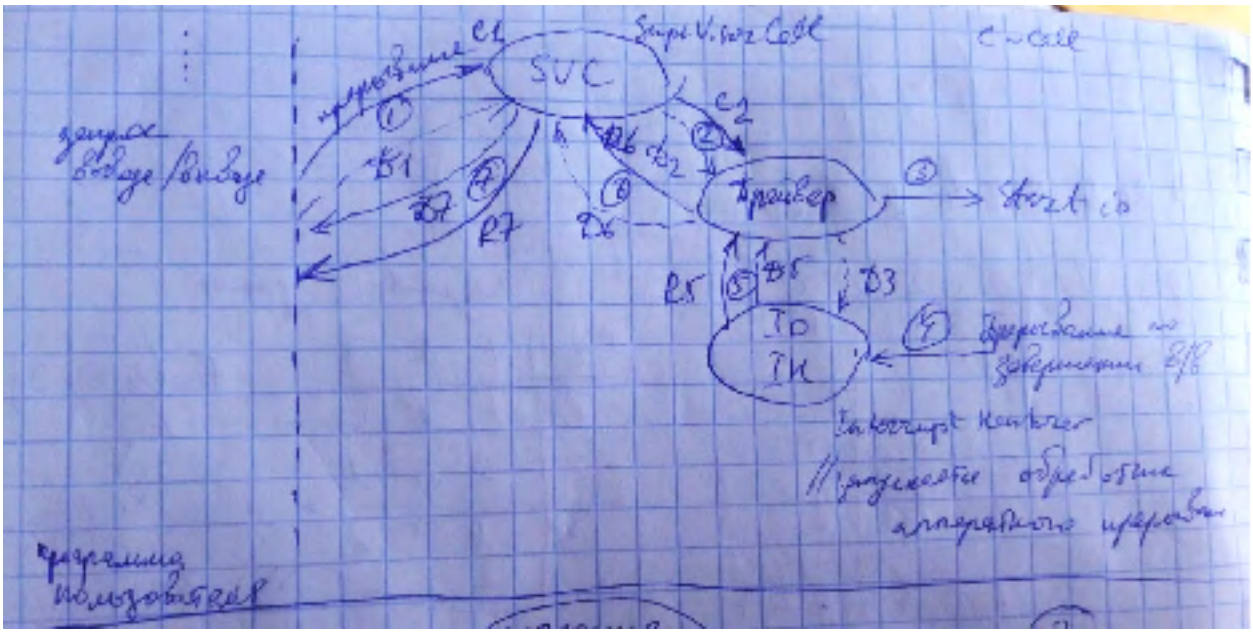
????бред????Шоу пытается все эти ситуации свести к уже доказанным теоремам и рассмотренным случаям с соответствующими добавлениями. Для того, чтобы их рассмотреть, нужно более подробно вникать в теоремы для повторно используемых ресурсов. Анализ может быть выполнен с определенными ограничениями и условиями.

Надо рассмотреть архитектуры ядер ОС. Есть ОС с монолитным, с микро-, с нано-, с экза- ядрами. Возьмем два варианта и рассмотрим их особенности: ОС с монолитным ядром и ОС с микроядром.

ОС с монолитным ядром

ОС с монолитным ядром, как и системы с микроядром, их работа базируется на прерываниях. Одними из основных прерываний в системе являются аппаратные прерывания, поэтому мы рассмотрим аппаратные прерывания. На семинаре рисовали 3-х-шинную концептуальную архитектуру, построенную по принципам фон-Неймана. В современных ОС есть система прерываний, которая объединяет 3 вида прерываний - системные вызовы, исключения, аппаратные прерывания.

Рассмотрим диаграмму выполнения запроса ввода-вывода. Эта диаграмма из книги Шоу. Это самое общее представление о тех действиях, которые выполняются в системе в результате запроса процесса на ввод/вывод. Процесс - программа в стадии выполнения.



Если посмотреть код библиотечных функций, мы увидим, что в итоге присутствует системный вызов - read write. Для работы с файлами есть и другие (открыть файл и т.д.) //системные вызовы входят в систему прерываний. В результате системного вызова мы обращаемся к ядру ОС, который принято называть супервизором. ???. Супервизор - ОС в стадии выполнения. ???. Еще передаются данные. Это 1-е действие. Обработав системный вызов, ОС определяет, какому устройству предназначен этот запрос и вызывает соответствующий драйвер устройства. Драйвер формирует соответствующую команду устройству ввода-вывода и посылает ее этому устройству. Все команды посылаются по шине данных. Все устройства, за исключением видеокарточек, адресуется с помощью портов ввода-вывода. На шину адреса выставляется адрес, на шину управления выставляется соответствующий управляющий сигнал read/write. Это является началом ввод-вывода устройства. Получив запрос ввода-вывода, супервизор блокирует процесс. Послав команду, драйвер также блокируется, т.е. процессор не управляет внешним

устройством, им управляет контроллер. Процессор может переключиться на другую работу. по завершению операции ввода-вывода устройство посылает соответствующий сигнал контроллеру прерываний. Контроллер прерываний в наших системах формирует вектор прерывания, а получив этот вектор процессор находит точку входа в обработчик прерывания. Обработчик прерывания формирует данные для ответа. В любом случае формируются данные, даже если это запись, но процесс все равно получает инфу о том, что запись произведена успешно. Обработчики прерываний входят в состав драйверов устройств, т.е. обработчики прерывания являются одной из точек входа драйвера. У драйвера могут быть дополнительные функции, которые дополнительно обрабатывают данные. Далее через интерфейс системы???. Для этого, чтобы вернуть процессу данные, он должен быть разблокирован. Все это время, пока система обслуживает запрос на ввод-вывод, процесс находится в состоянии блокировки. По завершению обслуживания, когда происходит передача данных процессу, он должен быть разблокирован. такой сон нельзя прерывать. Это непрерываемый сон, поэтому там буква D - девайс. У линуксоидов все девайс.

Прерывания делятся на 2 типа - **быстрые и медленные**. Быстрым прерыванием является только прерывание от системного таймера.

Медленные делятся на две части. Они называются tophalf и buttonhalf. В windows такое деление реализовано через DPC (Deferred procedure call). Быстрая часть выполняется от начала до конца, прерывать ее выполнения нельзя ...??? при запрещенных прерываниях, но есть системы, в которых реализуются вложенные прерывания.

Продолжаем обсуждать аппаратные прерывания. "Прерывани - это зло, но от них нельзя никуда деться". Прерывания связаны с дополнительными переключениями контекста. Сама обработка прерываний также является нетривиальной, но в системах с распараллеливанием функций - начиная с 3-го поколения ЭВМ появилась полноценная архитектура, в основу которой положен принцип распараллеливания функций. Функцию работы с внешними устройствами в шинной архитектуре взяли на себя контроллеры устройств, в канальной - каналы. Процессор перестал управлять работой медленных внешних устройств.

Собственно interrupts называются аппаратными прерываниями. Они приходят от аппаратуры - системный таймер, внешние устройства, и от сетевых устройств - внешние прерывания.

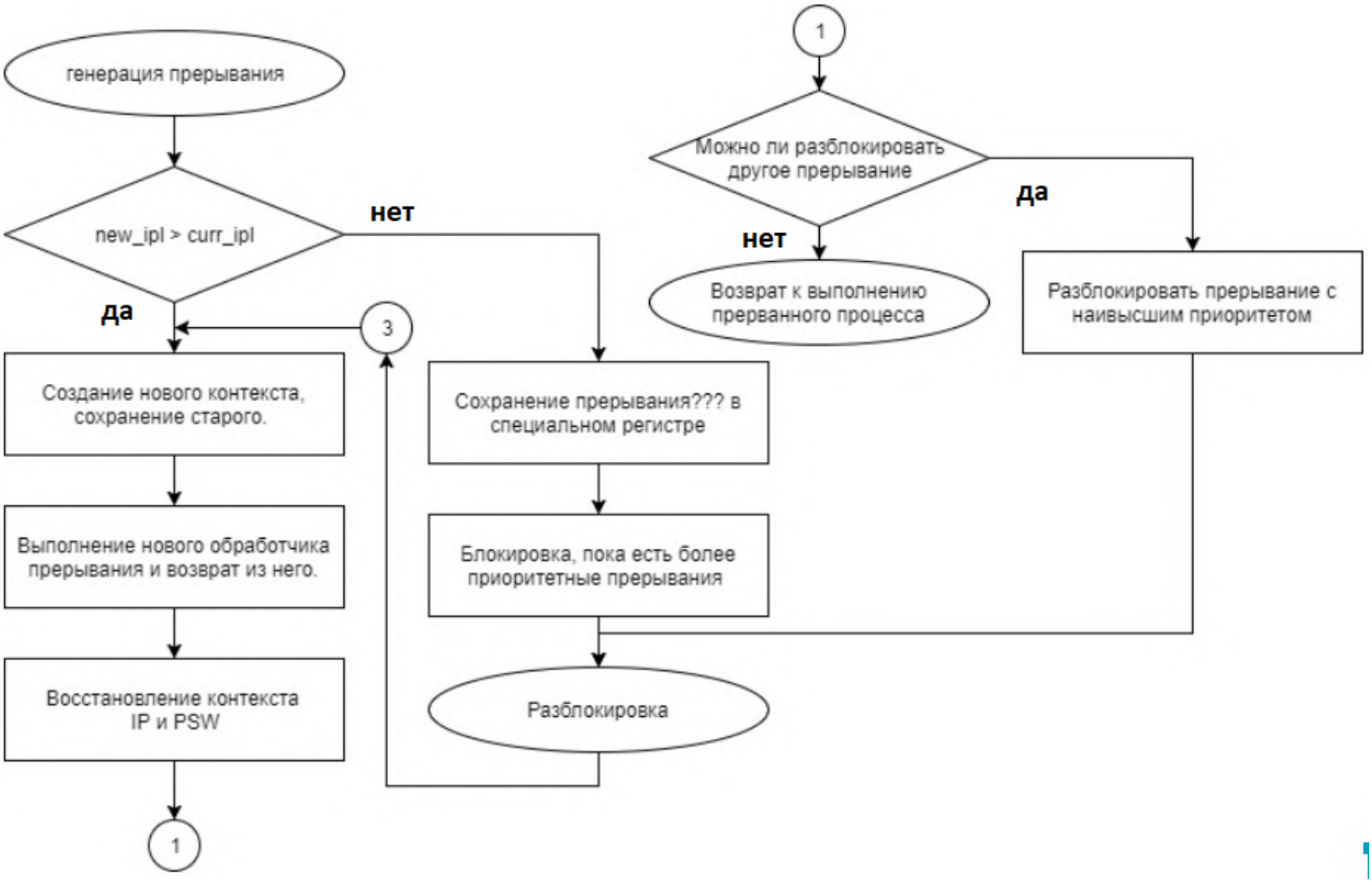
Мы нарисовали диаграмму, которая показывает этапы обработки запроса на ввод-вывод прикладной программы. В этой диаграмме показано, что драйвер инициирует операцию ввода-вывода после чего процессор отключается от управления внешним устройством, управление берет на себя контроллер. По завершению операции ввода-вывода ??? формирует прерывания ??? программируемый контроллер прерывания. Этот контроллер формирует вектор прерывания, который используется как смещение к дескриптору прерывания в таблице дескрипторов прерываний. Но обработка прерываний имеет также и другие аспекты.

Основная функция прерываний заключается в том, чтобы позволить периферийным устройствам информировать систему о завершении ввода-вывода или о каких-то ошибочных состояниях, или о завершении работы какой-то задачи, причем все эти события требуют внимания ОС. Аппаратные прерывания происходят независимо от текущих действий, выполняемых в системе. Это абсолютно асинхронные события. В результате возникновения аппаратного прерывания система вызывает на выполнение, или на обслуживания прерывания ISR. Обработчик прерывания выполняется в режиме ядра в системном контексте. Так как прерванный процесс обычно не имеет никакого отношения к возникшему в системе аппаратному прерыванию, его обработчик не должен обращаться к контексту прерванного процесса. По этой причине он не обладает правом блокировки (вспомнить диаграмму выполнения команды.) В конце выполнения каждой команды процессор проверяет наличие сигнала прерывания. Если сигнал прерывания пришел, то процессор переключается на обработку этого прерывания - на выполнения соотв. ISR, т.е. выполняемый процесс прерывается. Соответственно, сохраняется его аппаратный контекст. Аппаратное прерывание не может иметь доступ к контексту прерванного процесса, однако прерывание оказывает некоторое влияние на выполнение текущего процесса. Время, потраченное на обработку прерывания является частью выделенного процессу кванта.

Например, обработчик прерывания системного таймера использует тики текущего процесса, и поэтому нуждается в доступе к его структуре proc. Важно отметить, что контекст процесса не полностью защищен от доступа обработчиков прерываний. Неверно написанный обработчик прерывания может нанести вред любой части адресного пространства процесса.

Программные прерывания и исключения

Ядро, или система, поддерживают программные прерывания или исключения, которые генерируются при выполнении специальных инструкций. Не смотря на то, что такие прерывания происходят синхронно с работой системы, они обрабатываются также, как и аппаратные прерывания. Прерывания могут возникнуть в результате самых разных независимых событий в системе, поэтому реальна ситуация, когда во время выполнения одного прерывания, возникает выполнение второго. При этом прерывания от аппаратного или системного таймера должны обслуживаться сразу. Например, раньше прерывания пришедшего от сети, тем более, что такое прерывание может потребовать больших объемов вычислений в течении нескольких тиков таймера. В связи с этим вводится поддержка различных уровней приоритетов прерываний. В UNIX они называются ipl - interrupt priority level. Рассмотрим укрупненный алгоритм обработки прерываний.



Это алгоритм обработки вложенных прерываний. На некоторых аппаратных платформах поддерживается глобальный стек прерываний, который используется всеми обработчиками. На платформах, не имеющими глобального стека обработки прерываний, задействуется стек ядра текущего процесса. В системе должен быть обеспечен механизм изоляции остальных частей стека ядра от обработчика. Для этого ядро помещает в этот стек так называемый уровень контекста перед вызовом обработчика. Этот уровень контекста содержит в себе информации, необходимую для восстановления контекста предшествующего вызову обработчика прерывания. В качестве примера рассмотрим 2-ю системы - IBM 360, в которой были реализованы вложенные прерывания.

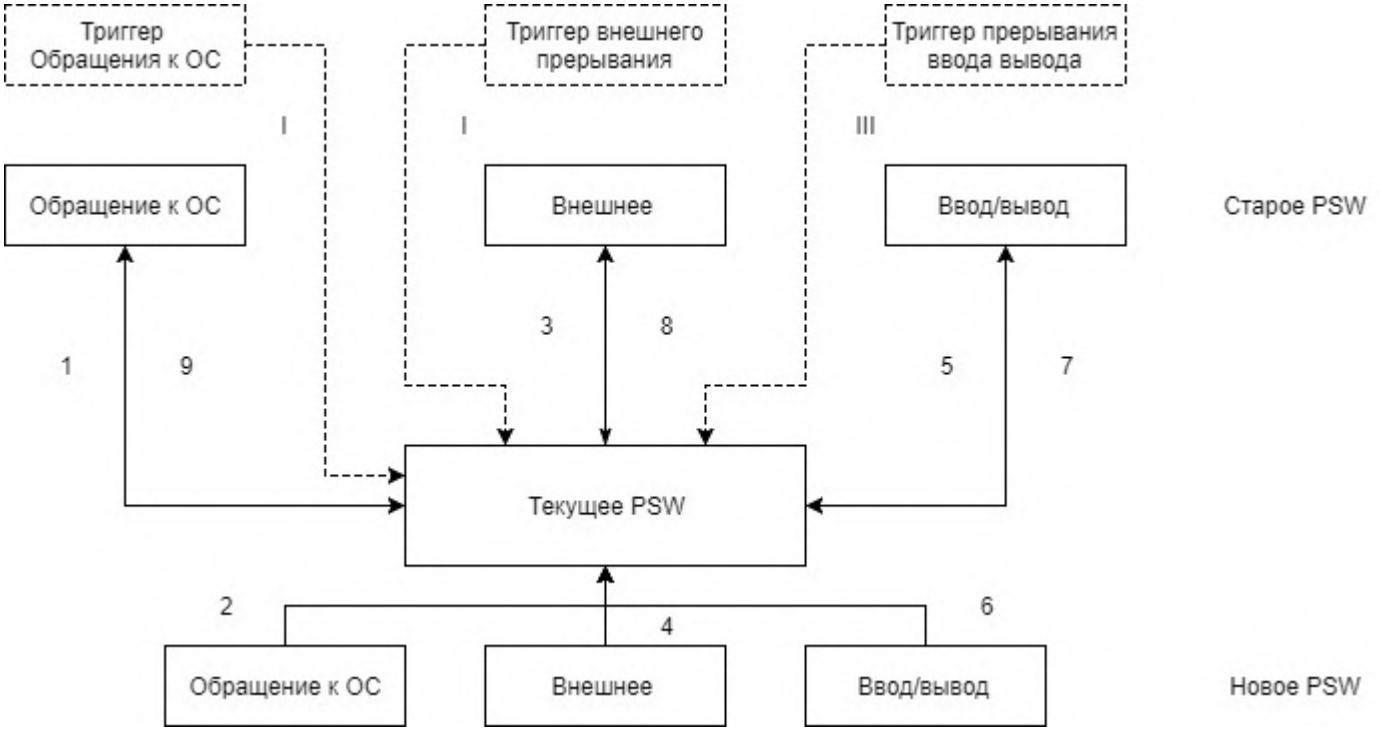
В пример демонстрируется следующая ситуация. Одновременно пришли 3 сигнала прерывания - системный вызов (обращение к ОС), внешнее прерывание, прерывание ввод-вывода. Системное имеет приоритет 5, внешнее 3, ввода-вывод 2. Рассмотрим действия на простой схеме:

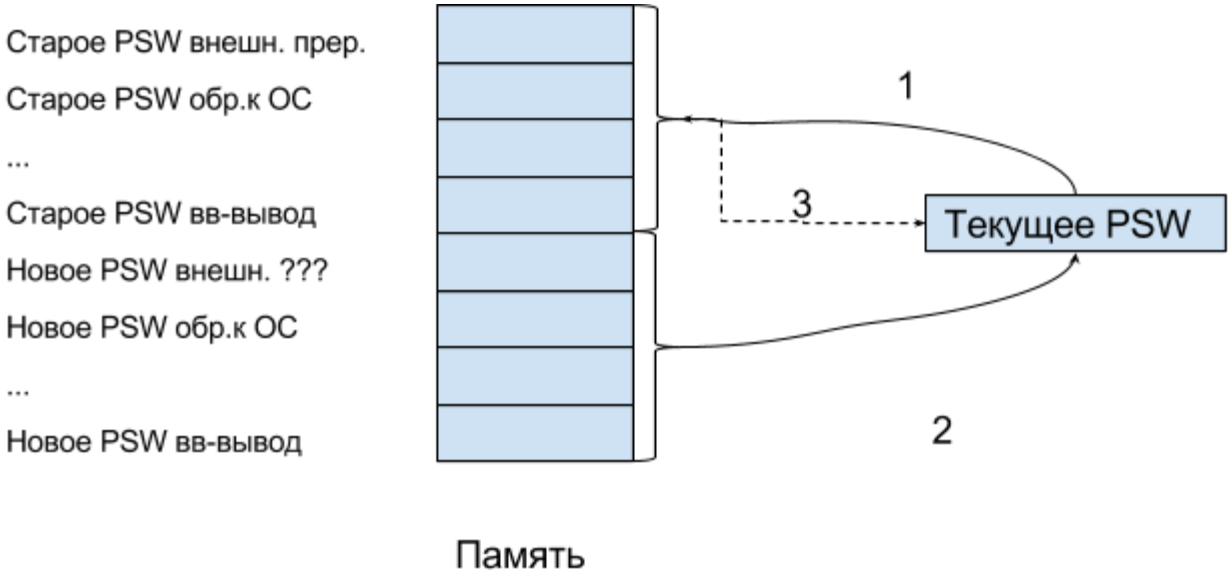
В системе 360 для поддержки нужных прерываний в адресном пространстве ядра системы были выделены области с фиксированными адресами - это были области памяти для сохранения старого и нового PSW. При этом сначала сохраняется текущее PSW в области старых PSW, из области новых PSW загружается соответствующий новый контекст, когда завершается обработка прерывания, из области старых PSW возвращается старый контекст. Из

данной иллюстрации мы видим, что прерывания начинают обрабатывать в порядке, обратному их приоритету, а заканчивается их обработка в порядке приоритетов.

??? что-то говорит. но я забила

Если расписать это чуть по-другому, то получится следующая ситуация - (2) т.е. реализован стековый алгоритм обработки прерываний - т.е. прерывания обрабатываются в соответствии с их приоритетами.





Архитектура ядер ОС (или структура ядер ОС)

Рассмотрим 2 типа ядер - монолитное ядро и микроядро.

Монолитное ядро является единой программой, которая имеет модульную структуру, т.е. состоит и подпрограмм. Windows и Unix Linux имеет монолитное ядро. Не смотря на то, что Windows была продекларирована как ОС с микроядром, но это не состоялось. Unix имеет минимизированное ядро, но тем не менее оно монолитное. Современные системы с монолитным ядром являются многопоточными. Windows является объектно-ориентированной системой. Unix Linux таковой не является. Поскольку монолитное ядро является единой программой, то единственным способом изменить его конфигурацию, например, добавить новые компоненты или удалить, является перекомпиляция ядра. Но современные версии ядер позволяют вносить какие-то изменения в ядро без его перекомпиляции. В Windows реализован так называемый стек драйверов - т.е. Windows предоставляет

пользователю возможность написания соответствующих драйверов, например, драйверов-фильтров верхнего или нижнего уровня, они могут изменять функциональность внешнего устройства, или выполнять анализ частей системы или процессов.

В Unix Linux это загружаемые модули ядра - в распоряжение пользователя предоставляется механизм, называемый загружаемый модуль ядра, это может быть драйвер. Включение в ядро драйверов в Windows или загружаемых модулей в Unix выполняется с помощью специальных действий без перекомпиляции. Но не все действия можно выполнить с помощью загружаемого модуля ядра. Чтобы получить интересующую информацию приходится подменять в структурах ядра какие-то функции, или дописывать что-либо. Это называется патчи. В этом случае ядро перекомпилируется. Патч - заплатка. В монолитном ядре взаимодействие приложений с ядром осуществляется с помощью системных вызовов. Взаимодействие с внешними устройствами начинает осуществляться с [помощью] системных вызовов и заканчивается аппаратным прерыванием. В наших системах нет глобального стека ядра, глобального стека прерываний, поэтому каждый процесс должен иметь два стека - стек режима ядра и стек режима пользователя. Kernel stack и user stack. При этом несколько процессов не могут использовать один стек пользователя. Это связано с тем, что состояние стека является частью контекста процесса и должно сохраняться при переключении на выполнение других процессов. Нельзя один и тот же стек использовать и как стек ядра, и как стек пользователя, так как программа может делать все что угодно со стеком уровня пользователя. Она может организовать несколько стеков уровня пользователя, может изменять размер этих стеков. Кроме того, в режиме ядра могут находится сразу несколько процессов, и между ними могут происходить переключения, поэтому у каждого процесса должен быть отдельный стек режима ядра. В монолитном ядре одной из основный функций ядра является обработка аппаратных и программных прерываний, и исключительных ситуаций.

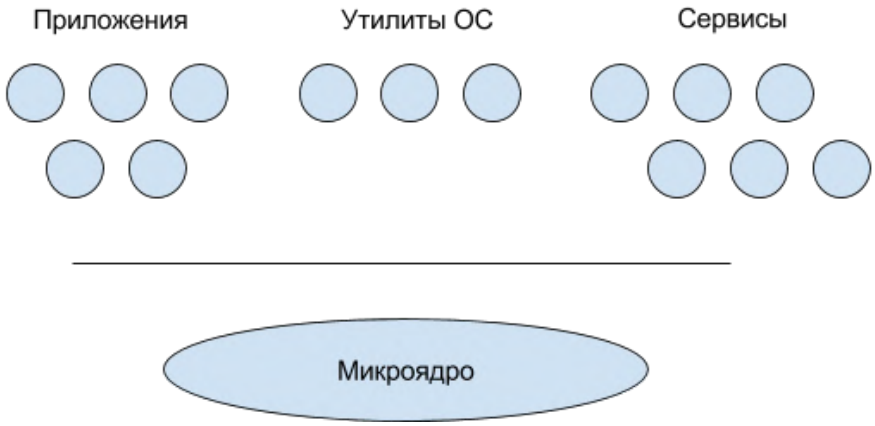
Микроядро

Микроядро, или микроядерная архитектура, в этой архитектуре все компоненты являются самостоятельными программами, которые возможно выполняются в разных адресных пространствах. Взаимодействие между такими программами осуществляется с помощью сообщений, и это взаимодействие обеспечивает микроядро. Микроядро - это модуль ядра ОС, который выполняет самые низкоуровневые действия - действия, непосредственно связанные с управлением аппаратной частью системы. Микроядро обеспечивает возможность взаимодействия процессов системы с помощью сообщений. Микроядро должно выполнять действия:???

Мы рассматривали иерархическую машину. С точки зрения процессов, самой низкоуровневой операцией является непосредственное выделения кванта процессорного времени процессу - т.е. диспетчеризация.

Планирование - более высокороуровое действие. По отношению к физической памяти - самым низкоуровневым является выделение физической страницы процессу. Решение о том, какую страницу вытеснить, какую заместить, какую загрузить - эти решения могут приниматься на более высоком уровне.

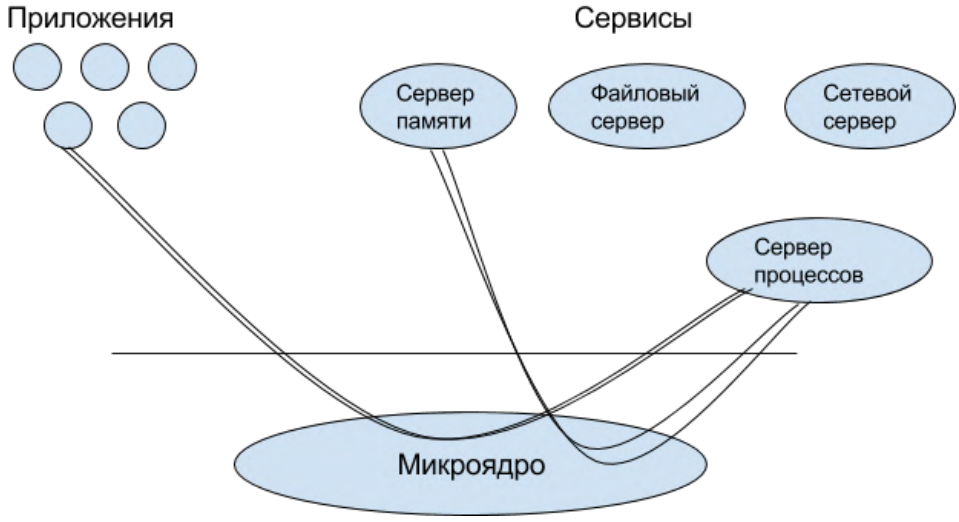
Управление внешними устройствами. Разработчики внешних устройства предоставляют в распоряжение разработчиков ОС так называемые функциональные драйвера. Он учитывает всю специфику внешнего устройства. Это пишут разработчики драйвера, так как только они знают формат данных, необходимых для работы устройства???? ???? Написать функциональный драйвер может только человек, который знает форматы всех передаваемых данных. Это нижний уровень управления внешними устройствами. Файловую подсистема имеет свою иерархию. Самым высоким уровнем иерархии является именование файлов - или символьный уровень файловой подсистемы. Этот уровень очень удобен пользователю, т.к. ему привычно обращаться к файлам по их имени. Есть логический уровень файловой подсистемы. И естественно, файловая подсистема должна обеспечивать хранение файлов на физическом носителе. (Например, на жестком диске, который является устройством.) В конечном итоге файловая подсистема обращается к системе ввода-вывода. Учитывая все это была поставлена задача минимизировать ядро ОС. Это декларируется с помощью следующих схем. (3)



????

Основная задача ОС - выделять процессам ресурсы. Но выполняется это самостоятельными процессами, которые могут выполнятся в соответствующих адресных пространствах.

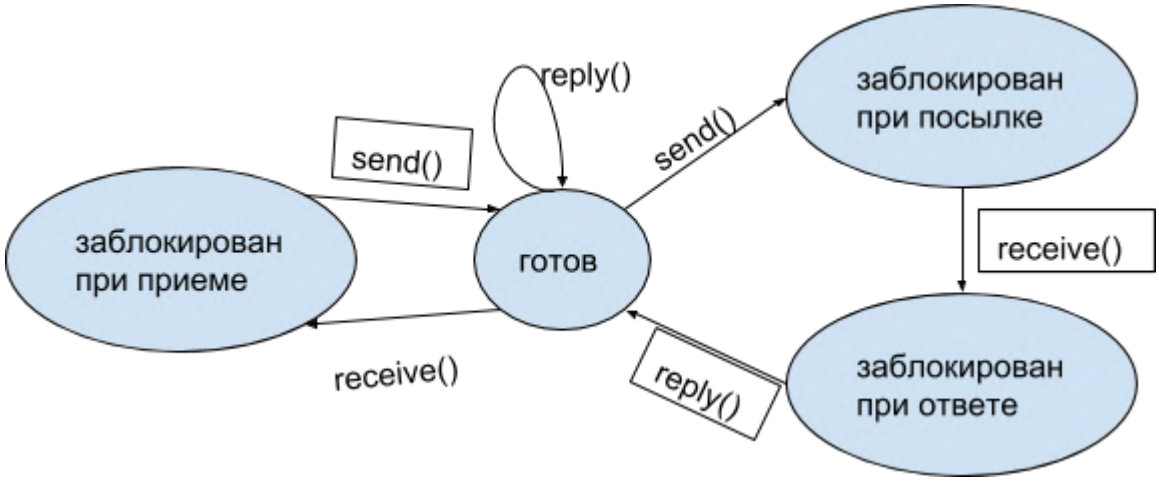
Микроядро выполняется по схеме(модели) клиент-сервер. Это отражено на картинке. Взаимодействие приложений с программами ОС и программ ОС друг с другом выполняется по модели клиент-сервер с помощью сообщений.



Например, программа решила создать новый процесс с помощью системного вызова fork, она обратилась к серверу процессов, он отвечает этой программе. Сервер процессов может обратиться к серверу память, который должен прислать ответ в результате проделанной работы.

Все взаимодействие в системе должно быть надежным. ???. Это приводит к необходимости синхронизации работы соответствующих процессов. И в связи с этим приводит к тем состояниям блокировки, которые мы с вами рассмотрели.

3 состояния блокировки процесса при передачи сообщений. В прямоугольниках функции, которые выполняются от имени другого пользователя. reply используется чтобы подчеркнуть, что это ответ (он посылается с помощью send)



//именно в микроядерной архитектуре неизбежно проявляются все эти три состояния блокировки.

Процесс заблокирован при посылке, если он вызвал системный вызов send, а процесс, которому он выслал сообщение, не готов его принять. Как только адресат вызовет receive, процесс, пославший сообщений будет заблокирован при ответе, и будет находится в этом состоянии до тех пор, пока запрос не будет обработан, и не будет вызван системный вызов reply. Если процесс вызвал системный вызов receive, а сообщение ему еще не отправлено, то процесс будет заблокирован при приеме.

Цитата из Соломона-Русиновича, который озаглавлен - “Основана ли Windows 2000 на микроядре". Windows 2000 не является ОС на основе микроядра в классическом понимании этого термина. В ОС с микроядерной архитектуре основные компоненты ОС, такие как диспетчеры памяти, процессы ввод-вывода выполняются как отдельные процессы в собственных адресных пространствах и представляют собой надстройки над примитивными сервисами микроядра (имеется в виду примитивы - функции ядра. исторически это функции нижнего уровня.)"

21.12.17.

Мы классифицируем современные ядра на монолитные и микроядро. Если ядро очень сильно минимизировано, т.е. оно реализовано как совсем небольшое, то можно встретить название наноядро, но смысл один и тот же. Монолитное ядро - единая программа, имеющая модульную структур, в которой реализованы все функции ОС, современные монолитные ядра многопоточные. Винда имеет объектно-ориентированное ядро, а линукс нет, в связи с тем что обработка объектов требует доп затрат. Ядро юникс линукс структурировано, в отличии от виндового.

Микроядро - ядро в котором оставлены только основные функции, их набор определяется разработчиками ОС, но основной задачей, которая характерна для всех микроядерных архитектур является обеспечение взаимодействия процессов. Они построены по модели клиент-сервер. Пилотным проектом ОС с микроядром была ОС Mach (керниган). Микро ???Когда пользовательской программе-клиенту требуется вызвать какую-либо функцию ОС, например, прочитать данные и файла, она посылает сообщение серверу, который реализует данную функцию - файловый сервер. Один и тот же процесс (микроядерная архитектура имеет небольшое ядро за счет того, что высокоуровневые функции вынесены из ядра и реализованы в виде высокоуровневых программ). Не только приложение может быть клиентом, а и серверы могут выступать в качестве клиентов, если им необходимо использовать функции других серверов. Например, файловая система является сервером для процесса, открывшего файл и клиентом для драйвера диска, на котором этот файл располагается. т.е. в этих системах отсутствует четкая граница между функциями ОС и функциями прикладных программ. Например, в ОС с микроядром Hurd пользователь может создать собственную файловую систему (в след. семе увидим, что пользователь линукс также легко может создать собственную файловую систему).

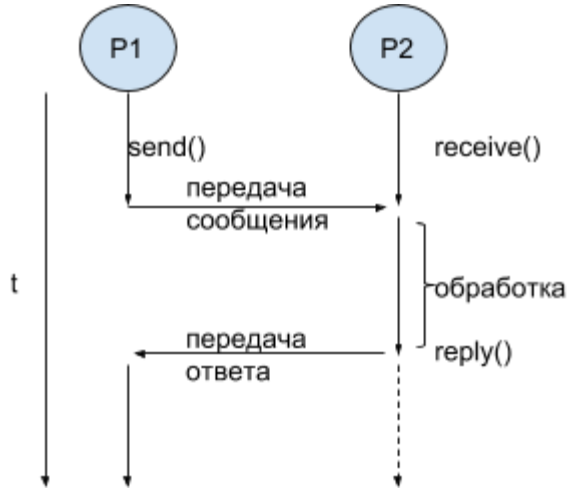
В монолитном ядре, когда программе нужен какой-либо сервис, она вызывает библиотечную функцию и передает ей необходимые параметры. Эта функция просто загружает переданные параметры в регистры, после чего

осуществляет системный вызов. В случае микроядерной ОС параметры должны быть собраны в сообщения, причем необходимо указать не только адрес, но и размеры всех областей памяти, которые используются как параметры и указать буфера для получения результата. Так как процесс, получающий сообщение не имеет доступа к адресному пространству процесса, пославшего сообщение, все параметры сообщения должны быть скопированы в адресное пространство процесса, принимающего сообщения. в чем разница? Отображены - значит, что процесс получает соответствующий указатель на область памяти, в которой находятся соответствующие данные. Мы рассматривали разделяемые сегменты, так вот разделяемый сегмент отображается на адресное пространство процесса, и рассматривали очереди сообщений. А сообщения копируются в адресные пространства процессов.

Результат обработки сообщения может быть либо скопирован в адресное пространство процесса, запросившего обслуживания, либо выделенная для него область памяти должна быть заранее отображена в адресное пространство процесса, который обрабатывал это сообщение.

Мы рассмотрели состояния блокировки процесса при передачи сообщений. Это связано с тем, что передача сообщений в системе с микроядром должна производится надежно. В системе QNX используется системный вызов reply - они разделили прием и посылку ответа.

Для того, чтобы отправить сообщение, запросить обслуживание, процесс вызывает системный вызов send, где указывает что и от кого ему нужно. Чтобы принять сообщение, процесс, обслуживающий запросы выполняет системный вызов receive. Тут большое значение имеет синхронизация процессов.



Нельзя предугадать где и когда процесс придет в определенную точку, все процессы асинхронны, выполняются с собственной скоростью.

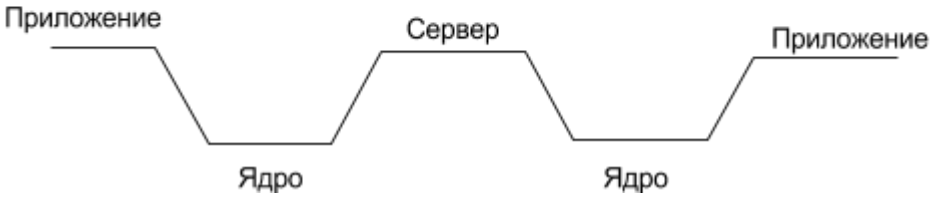
При блокирующем синхронном соответствующий процесс, который послал сообщение-запрос, блокируется, пока его сообщение-запрос не будет принят. Когда его сообщение принято, он блокируется в ожидании ответа.

Еще раз: когда процессу требуется обслуживание другого процесса, он формирует сообщение-запрос. Вызывает системный вызов send, в результате которого это сообщение передается по соответствующим средствам связи и процесс, который сформировал сообщение-запрос будет блокирован до того момента, пока процесс, которому направлен этот запрос не выполнит системный вызов receive. Когда он выполнит receive, исходный процесс переходит в состояние ожидания ответа. Понятно, что это единая блокировка, но здесь это соображение очень важно.

Например, P2 готов принять сообщение-запрос. Он блокирован на receive до тех пор, пока другой не вызовет системный вызов send. т.е. такое взаимодействие связано с блокировками, которые мы нарисовали (3 состояния). Этот момент позволяет проанализировать производительно микроядерных архитектур.

1) в системе с монолитным ядром обработка системного вызова связана с 2-мя переключениями - из режима пользователя в режим ядра и возвратом в режим пользователя.

В микроядерной архитектуре это как минимум 4 переключения.



Эти задержки можно оценить. Но задержки, связанные с передачей сообщений, с блокировкой, являются вероятностными, их оценить невозможно, но они существуют.

Цитата Соломона-Русиновича стр. 25 книги винда 2000 "основана ли винда 2000 на микроядре. винда 2000 не является ОС основанной на микроядре в классическом понимании этого термина" (декларировалась как система с микроядром). "в ос с микроядерной архитектурой основные компоненты ос, а именно испеттчеры файлов, проуессов ввод-вывода, выполняются как отдельные процесса в собственных адресных пространствах и являются ???? надстройками над функциями микроядра.?????"

Пример современной системы с архитектурой на основе микроядра "mach". Она реализует крошечное микроядро, которое включает сервисы планирование потоков, передачи сообщений, виртуальной памяти, драйвера устройств. Поддержка сетей, файловая система, различные API, работают в пользовательском режиме. однако в коммерческих реализациях на основе этой ОС, код поддержки сетей и т.д. выполняются в режиме ядра. С коммерческой точки зрения системы на основе микроядра не практичны из-за слишком низкой эффективности. Почему не угасает интерес к микроядерной архитектуре? Потому что ОС с микроядром значительно проще модифицировать. Модифицируются процессы уровня задачи, не затрагивая микроядро.

Mach является системой общего назначения

Рассм основные концепции ОС Mach. те базовые понятия, которые обсуждались в течении сема, характерны для всех ОС.

Разработчики ОС всегда выбирают базовые абстракции. (У современных разработчиков нет выбора, т.к. базовые абстракции одни и те же, но есть особенности их реализации). Ядро ОС Mach построено на 5 абстракциях.

1. процессы

2. нити (иногда обозначается как поток в русс лит), при этом поток является независимо планируемым контекстом. Единицей диспетчеризации, единицей планирования становится поток. Но поток не имеет собственного адресного пространства, владельцем ресурсов является процесс.

3. объекты памяти

4. порты

5. сообщения

Объект памяти (memory object) - представляет структуру данных, которая может быть отображена в адресное пространство процесса. Объекты памяти занимают одну или несколько страниц и образуют основы для системы управления виртуальной памятью. Когда процесс ссылается на объект памяти, который отсутствует в физ памяти, точно также как и в любых ОС с виртуальной памятью, возникает страничное прерывание. Однако, в отличии от других систем, при обработке этого страничного прерывания, этого исключения, ядро Mach для загрузки отсутствующей страницы посылает сообщение серверу пользовательского режима. Соответственно этот сервер должен решить, в какой страничный кадр нужно загрузить нужную страницу. Любая система заинтересована в том, чтобы заранее освободить кадры памяти, к которым дольше всего не было обращения. Система заинтересована, чтобы на момент страничного прерывания у нее были свободные кадры памяти, т.к. это гарантирует более быстрое продолжение работы процесса. Для организации взаимодействия процессов вводится понятие порт. При этом порт - это почтовый ящик, фактически буфер. Порт создается в адресном пространстве ядра и способен поддерживать

очередь, т.е. упорядоченный список сообщений. В системе Mach имеется несколько типов портов. Порт процесса используется для взаимодействия с ядром. Многие функции ядра процесс вызывает путем отправки сообщений на порт процесса, а не с помощью системного вызова. Зарегистрированные порты используются для обеспечения взаимодействия процессов со стандартными системными серверами. Порт особых ситуаций используется системой для передачи сообщений об ошибках процессу (например деление на ноль). Системы с микроядром не эффективны. Это связано с тем что тратится время на передачу сообщений, возникают блокировки в различных ситуациях. Речь идет о системах общего назначения.

Системы реального времени - гуглим - обнаружим, что выводится довольно обширный список коммерчески значимых систем реального времени, и эти системы являются ОС с микроядром. Это связано с тем, что системы с микроядром леко модифицировать, легко дополнять новыми функциональностями. при этом ядро должно быть достаточно гибким, чтобы удовлетворить заказчиков.

Рассмотрим ОС реального времени QNX NEUTRINO RTOS v62.

Дадим определение ОС реального времени по стандарту POSIZ 1003.1. Он определяет ОС реального времени следующим образом. Реальное время в ОС - это способность ОС обеспечить требуемый уровень сервиса в определенный промежуток времени. т.е. задачи реального времени составляют одно из сложнейших и важнейших областей применения в вычислительной технике. Многие ОС общего назначения также поддерживают сервисы реального времени, однако, ключевым отличием сервисов ядра ОСРВ является детерминированый, основаный на строгом контроле времени, характер их работы. В данном случае под детерминированностью понимается, что для выполнения одного сервиса или одной услуги ОС требуется временной интервал заданной продолжительности. Разделяют ОСРВ с жестким и гибким реальным времени. В ОС с **жестким** РВ все интервалы обслуживания внешних запросов строго определены и не могут быть превышены, так как это может привести к трагическим последствиям. В ОС с **мягким** реальным временем возможны небольшие отклонения от заданного интервала, в течении которого система должна обработать внешний запрос. К жестким системам относятся системы управления атомными реакторами, а к гибким системам относятся, например, система продажи авиабилетов. В этих системах небольшие задержки в ответе системы не несут катастрофических последствий.

Внешние события, на которые система должна реагировать, можно разделить на периодические и непериодические. Периодические события возникают через определенные интервалы времени, а непериодические возникают случайным образом. Для системы, на которую поступают периодические события, можно определить, является ли такая система планируемой или нет ,т.е. способна ли система обработать данное событие за требуемые интервалы времени. Если в систему поступает M периодических событий, причем i-ое событие поступает с периодом pi, и на его обработку отводится ci миллисекунд.

$$\sum_{i=1}^{m} \frac{c_i}{p_i} \leq 1$$

И если выполняется это условие, то такая система называется планируемой, т.е. она способна выполнять управление такими процессами.

Рассмотрим (поверхностно) ОСРВ QNX NEUTRINO. Точно так же как и mach, она построена по модели клиент-сервер - это общий подход к построению ОС с микроядром. При этом, понятие клиента и сервера достаточно условны, т.к. одна и та же программа может быть и клиентом и сервером. В ОС Юникс реализуется многозадачность, диспетчеризация потоков на основе приоритетов и структура которая приводится (6).

менеджер процессов, устройств, файловой системы, сети.

Ядро имеет уровень привилегий 0, менеджеры и драйверы - 1 и 2, и прикладные процессы - 3. В отличии от линукса, тут задействованы все 4 кольца защиты. Как видно из этой схемы, драйверы устройств выполняются в системе как отдельные процессы. Следует обратить внимания, что управление процессами выполняется не микроядром, а менеджером процессов, но этот менеджер процессов скомпонован с микроядром в один модуль. Это хитрость с точки зрения разработчиков системы. Микроядро обеспечивает обмен сообщениями, управление потоками, планирование потоков, синхронизации потоков, управление сигналами и управление таймерами. Все процесс QNX имеют защищенные адресные пространства - защищенные виртуальные адресные пространства. За реализацию такой защиты отвечает MMU. Важнейшим моментом при разработки любой ОСРВ с микроядром является обеспечение быстрого переключения контекста. Мы видели, что в микроядерной архитектуре переключение контекста выполняется как минимум в 2 раза чаще, а переключение контекста - затратная операция. Поэтому разработчики ОС включили в состав основных абстракций потоки, которые разделяют адресные пространства процессов, и это занимает меньше времени. Но от переключение процессов никуда не дется. Но в QNX реализовано быстрое переключение контекста процессов. 2-м важным моментом для ОС жесткого РВ является вопрос управления виртуальной память. Необходимо понимать, что paging является крайне затратным действием в системы, поэтому процессы жесткого РВ должны полностью находится в ОП, т.е. для них paging недопустим.

Драйверы устройств в QNX - это процессы, которые являются посредниками между ОС и устройствами. Так как драйверы запускаются как процессы уровня пользователя (говорили, что кольца защиты 1 и 2), то добавление нового драйвера в QNX не влияет на микроядро. В QNX реализованы все средства межпроцессного взаимодействия которые мы изучали - и разделяемая память и очередь сообщений и программные каналы, но основным способом межпроцессного взаимодействия является передача сообщений. Указывается, что QNX была первой коммерческой ОСРВ, которая использовала передачу сообщений в качестве основного способа межпроцессного взаимодействия. В ней реализованы 3 системных вызова - send, reply, receive. Причем, передача сообщений реализуется как синхронное блокирующее.

Т.о. QNX ядро выполняет 2 важнейшие функции - передачу сообщений, т.е. ядро обеспечивает маршрутизацию сообщений передаваемых процессов. В системе, и диспетчеризацию потоков. Само ядро никогда не получает управление в результате диспетчеризации. Код ядра выполняется только в результате системных вызовов, исключений, или аппаратных прерываний. Как мы говорили, в QNX процесс выполняется в собственном защищенном виртуальном адресном пространстве и простейший процесс содержит один поток. Отсюда можно сделать вывод, что диспетчеризация потоков происходит по одному из 3 событий:

1. вытеснение - в результате поступления в очередь потока с более высоким приоритетов, при этом вытесненный поток ставится в очередь первым.

2. блокирование - поток во время своего выполнения может вызвать функцию, которая приведет к его блокировке.

3. уступка - (yield) поток может добровольно передать управление если вызовет функцию sched_yield(). В этом случае уступивший поток ставится в конец очереди.

Интересной особенностью QNX является дисциплина планирования. QNX поддерживает следующие дисциплины планирования - fifo, rr, спарадическое планирование (sparadic scheduling). спарадическое - значит, случайное. Для ОСРВ это странное название, но вот так вот. Спарадическое планирование предполагает, что приоритет процесса меняется в зависимости от следующих параметров:

1. начальный бюджет - это интервал времени, за которое поток может выполниться с нормальным приоритетом, прежде чем он получит пониженный приоритет.

2. пониженный приоритет - уровень приоритета, до которого может быть понижен приоритет потока. Если приоритет потока понижен, то он выполняется в **фоновом** режиме. Если поток выполняется с нормальным приоритетом, то говорят, что поток выполняется с приоритетом **переднего** плана.

3. период выполнения - это период времени, в течении которого поток может расходовать свой бюджет выполнения.

4. максимальное число текущих пополнений - поток не может пополнять свой бюджет больше чем установленное количество раз.

Все эти 4 параметра позволяют разработчикам создать нужный им алгоритм планирования. Процессы РВ могут иметь статические и динамические приоритеты. Статические назначаются в начале выполнения и в процессе выполнения не меняются. В современных системах процесс РВ могут иметь динамический приоритет. Спарадическое планирование является примером динамического изменения приоритета.

Когда возникают состояния блокировки при передачи сообщений:

Механизм передачи сообщений в QNX называется SRR по первым буквам функций, которые используют для передачи. send служит для передачи сообщений от клиента серверу и получения ответа. При вызове функции send клиент блокируется в одном из двух состояний SEND и REPLY. Когда сервер вернет сообщение-ответ клиенту, клиент будет разблокирован. msg_receive() служит для приема сооб от клиентов. Сервер вызывает receive и блокируется в состоянии RECEIVE, если ни один из клиентов еще не послал ему сообщение, т.е. не вызвал функцию msg_send(). reply используется для передачи ответа клиенту. При вызове функции reply блокировка сервера не происходит, он продолжает работать. Это сделано потому, что клиент уже находится в заблокированном состоянии и доп синхронизация не требуется.

Заключение

Количество систем РВ огромное. Потому что это одна из основных задач, которая возлагается на вычислительные системы - управление внешними объектами или процессами. Это могут быть старые системы, потому что в промышленности, военных, системы долго не меняют. Там имеется большая инерционность. Поэтому то, что мы рассматривали очень важно. но фактически ничего нового мы не услышали. все строится на процессах и потоках,, но ОСРВ имеют свои особенности.

не советую при ответе на экзамене путать системы реального времени и систему разделения времени!!!!

Процессы в Unix (из семинаров)

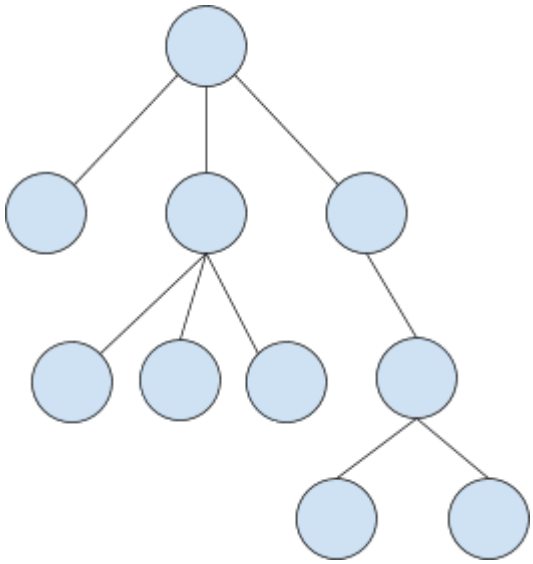
Процесс - программа в стадии выполнения, единица декомпозиции системы. Ему выделяются ресурсы, основной ресурс – процессорное время. В многопроцессорной ОС одновременно существует большое количество процессов. За время жизни процесс переходит из одного состояния в другое. Каждому запущенному процессу выделяется структура (дескриптор) процесса (строка в таблице процессов).

Говорят, что процесс часть времени находится в режиме задачи(выполняет собственный код), а часть времени в режиме ядра (выполняет реентерабельный код).

Системный вызов fork

Все процессы в Unix создаются единообразно - при помощи системного вызова fork() Любой процесс может создать любое кол-во процессов(ну на самом деле есть максимум). При этом процессы в Unix находятся в отношении предок – потомок. В результате создается иерархия процессов.

Иерархия поддерживается при помощи указателей в дескрипторе процесса. В системе одна главная таблица процессов (в BSD - двусвязные списки).



В результате системного вызова fork() создается копия процесса предка, в том смысле, что процесс-потомок наследует код предка, дескрипторы открытых файлов, сигнальную маску. Процесс-потомок начинает параллельно с вызвавшим fork() выполняет ту-же программу.

fork() возвращает потомку 0, а предку pid потомка. Причина создания иерархии - по ней возвращаются статусы.

“Умный” fork()

В старых системах Unix (для pdp 11) код предка копировался в адресное пространство потомка. В результате в системе могло существовать одновременно большое количество копий одной и той же программы. Память используется неэффективно. В современных системах используется "умный fork", который реализуется следующим образом: для процесса – потомка создаются собственные карты трансляции адресов (таблица страниц), которые ссылаются на страницы адресного пространства предка, т.е. в таблице страниц потомка находятся дескрипторы страниц адресного пространства предка (адреса страниц адресного пространства предка).

Для страниц сегментов Данных и Стека права доступа меняется на ReadOnly (OnlyRead) и устанавливается флаг CopyOnWrite. Если предок или потомок пытаются изменить какую-либо страницу, то возникает исключение по правам доступа. Обрабатывая исключение супервизор (ОС в стадии выполнения) обнаруживает установленный флаг CopyOnWrite. Вызывается менеджер памяти (MMU) и создаст копию страницы в адресном пространстве того процесса, который пытался её изменить. В таблице страниц (для процесса, который пытался изменить) создается дескриптор для этой страницы, а старый аннулируется.

Такая ситуация в системе действует до тех пор, пока потомок не выполнит системный вызов exit() или exec(). Если потомок выполнил эти системные вызовы, то страницы предка получают обычные права (read / write) и флаг CopyOnWrite сбрасывается.

Системный вызов exec

Для перехода на выполнение другой программы используется системный вызов exec().
exec() переводит процесс на новое адресное пространство - пространство программы, которая передана в качестве параметра. exec() создает низкоуровневый процесс (не имеющий pid), для этого процесса создается адресное пространство (таблица страниц). После создания таблицы страниц exec() заменяет указатель в дескрипторе процесса на указатель на новую таблицу страниц. В счетчик команд записывается адрес точки входа. После этого программа начинает выполняться. В exec передаются исполняемые файлы.
Для запуска новой программы:
fork()
из тела функции exec()

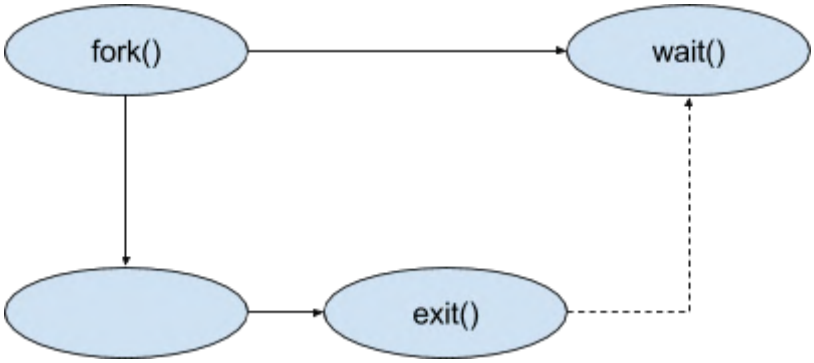
Процесс-сирота

Процесс-сирота - процесс, у которого завершился процесс предок. При завершении любого процесса система проверяет, не остались ли у него незавершенные потомки, если такие процессы – потомки у него остались, то выполняются действия по усыновлению осиротевших процессов **терминальным процессом**. Для этого модифицируются соответствующие дескрипторы. Процесс потомок в parentid получает id терминального процесса, а терминальный процесс получает указатель на осиротевший процесс. При завершении потомков процессы-предки получают статусы завершения своих потомков. Если произошло усыновление, то статус завершения получит терминальный процесс. Хороший предок должен дождаться завершения своих потомков.

Системный вызов wait

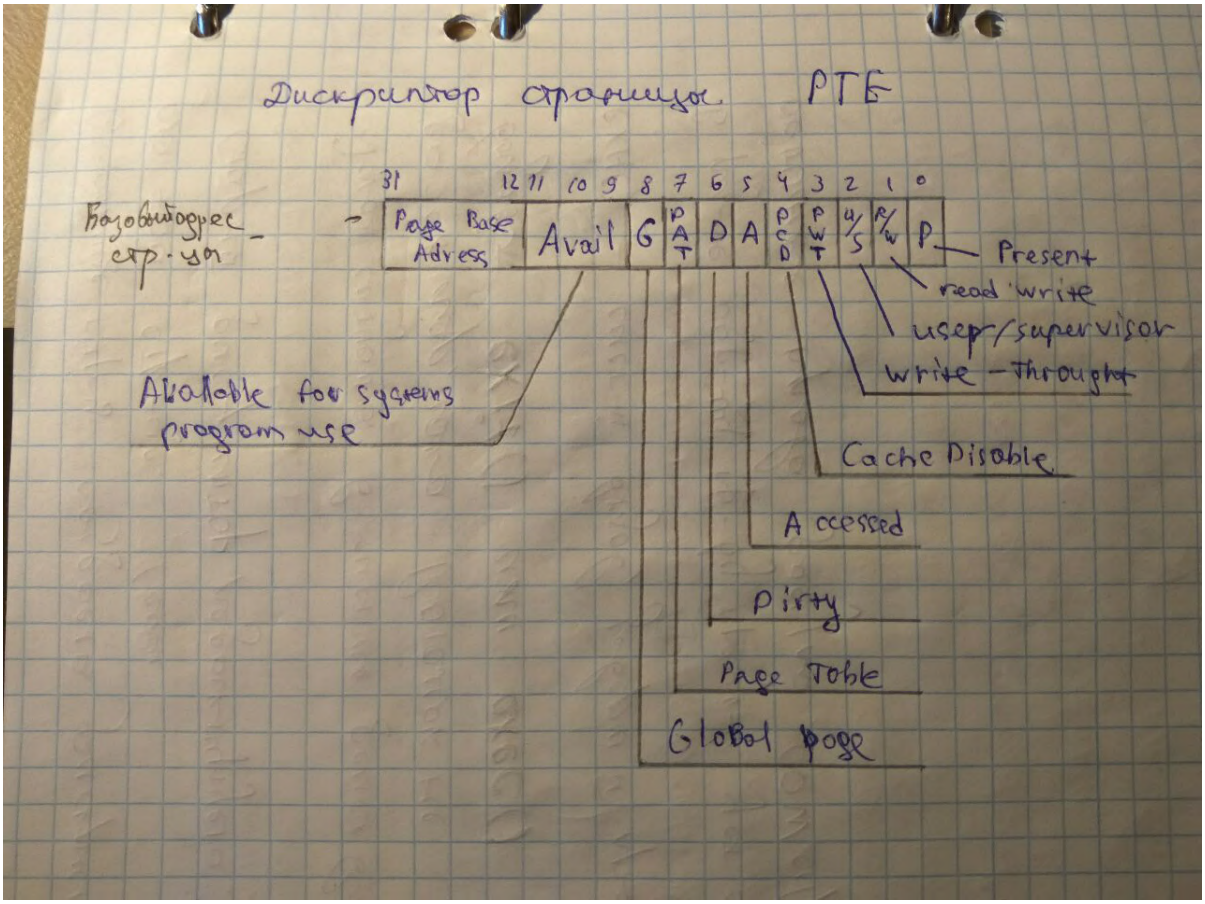
wait - переводит процесс в состояние ожидания. Предназначен для перевода процесса-предка в состояние ожидания завершения своих потомков (блокирует процесс до завершения его потомков). При этом предок получает статус завершения потомков.

Процесс-зомби - процесс, у которого отобраны все ресурсы, кроме последнего - строки таблицы процессов. Это необходимо для предотвращения вечного блокирования предка на wait(). Все процессы проходят это состояние.



Демоны - процессы, “не имеющие” предка. Реально предок существует.

тут пока просто так дескриптор страницы приведу ХЗ куда его конкретно пихнуть.



Взаимодействие процессов (IPC System 5)

Unix BSD и System 5 - коммерческие системы. В каждой из систем свои системные вызовы. Постепенно в ходе развития была утрачена совместимость.

Средства многопроцессорного взаимодействия System 5:

1) Сигналы
2) Программные каналы (именованные и неименованные)
3) Семафоры

4) Разделяемая память
5) Очереди сообщений (в BSD это сокеты)

Сигналы.

В Unix - сигналы, в Windows – события.

Механизм сигналов в Unix позволяет процессам реагировать на события, которые могут произойти или внутри самого процесса или вне его. Сигнал – базовое средство информирования процессов о событиях в системе. Самым важным событием в системе является завершение процесса. Получение процессом сигнала указывает ему на необходимость завершиться. Вместе с тем, реакция процесса на принимаемый сигнал зависит от того, как сам процесс определил свою реакцию на данный сигнал. Для этого процесс должен содержать свой обработчик сигнала, если процесс желает изменить реакцию на получаемый сигнал, он должен определить эту реакцию с помощью своего обработчика сигнала.

Процесс может: принимать сигналы, порождать сигналы и обрабатывать сигналы.

Сигналы бывают:

1) Синхронные - инициирует сам процесс.
2) Асинхронные - инициированы пользователем или другим процессом.

В классическом Unix 20 сигналов, располагаются в <signal.h>

\\ Рязанова скалаза все помнить не надо

SIGINT - завершение процесса по нажатию ctrl+c

SIGGEGV - нарушение границы сегмента, выход за пределы сегмента

SIGUSR1, SIGUSR2 - на них можно повеситься =)

SIGALARM - прерывание от таймера.

Группы процессов

В Unix процессы объединяются в группы. Процессы одной группы могут получать одни и те-же сигналы. Одна из важнейших групп - терминальная - все процессы, запущенные на данном терминале. Любой процесс вызывающий fork становится лидером группы.

Средством приема и посылки сигналов является системные вызовы kill() и signal().

signal не входит в POSIX, но есть в ANSI\ISO => есть в любой поставке Unix.

Системный вызов Kill

kill(pid, sig) - отправка сигнала sig всем родственным процессам процесса с идентификатором pid.

Если pid <= 1 Сигнал посылается группе процессов.

Если pid == 0, то sig будет послан всем процессам с идентификатором группы, совпадающим с id группы процесса, осуществившего системный вызов, кроме процессов с id 0 и 1.

Системный вызов Signal

void (*signal(int sig, void(*handler)(int))(int))

Процесс может определить свою реакцию на сигнал. В результате вызова signal будет вызвана функция handler(int).\\Возвращается указатель на старый обработчик.

Функции системного таймера

Windows

Немного инфы о таймере (в защищенном режиме, но в целом тоже).

В системе разделения времени время квантуется. Система разделяет время между процессорами (процессами). Главных тиков может быть несколько. В системе по главному тику выполняются действия, которые должны быть выполнены заранее. Декремент (см ниже) может делать только таймер.

Когда истекает квант, пересчитывать приоритет уже поздно, поэтому пересчет осуществляется по главному тику.

Планирование - заранее.

Прерывание от системного таймера - единственное быстрое прерывание (это очень высоко приоритетное действие).

обработчик прерывания от таймера не может долго работать, поэтому он не вызывает функции напрямую, с помощью команды call, а он просто код функции переводит из состояния “sleep” в состояние “running”, то есть просто ставит их в очередь на исполнение.
Планировщик - высоко приоритетная программа, но таймер не будет ждать его исполнение. Такой вызов называется Differate Procedure Call (DPC).

Функции системного таймера в Windows
1. По тику:
   - Инкремент значения счетчика системного времени.
   - Декремент счетчика времени, оставшегося до выполнения отложенных задач
   - Декремент кванта текущего потока на постоянную величину (3)
2. По главному тику:
   - Добавление в DPC-очередь диспетчера настройки баланса для динамического пересчета приоритетов.
   - Активизация обработчика ловушки профилирования ядра путем добавление в DPC-очередь.
3. По кванту:
   - Инициирует диспетчеризацию потоков (добавлением в DPC-очередь объекта диспетчера-планировщика потоков для перераспределения процессорного времени.)

Unix
1. По тику:
   - Инкремент значения счетчика тиков системного таймера.
   - Обновление статистики использования процессора текущим процессом (инкремент значения p_cpu).
   - Декремент значения счетчика времени, оставшегося до выполнения отложенных вызовов; установка флага, который сообщает о необходимости запустить обработчик отложенных вызовов, если счетчик становится равным нулю.
   - Декремент кванта времени.
2. По главному тику (время и список операций может меняться в зависимости от системы):
   - Декремент значения счетчика времени, оставшегося до отправки процессу сигналов SIGALRM, SIGVTALRM или SIGPROF.
   - Сообщение о необходимости вызова процедуры shedcpu(), которая пересчитывает приоритет текущего процесса.
   - Сообщение о необходимости вызова PageDaemon и Swapper (если требуется).
3. По кванту:
   - Посылка текущему процессу сигнала SIGXCPU, если он превысил выделенный ему квант процессорного времени.

Защищенный режим (IDT, GDT, теневые регистры)
Регистры - 32 разряда, шина адреса - 32 разряда. Режим называется защищенным потому, что адресные пространства процессов должны быть защищены.
(Реальный режим - 16-ый, шина адреса - 20-разрядная, Спец режим - V86 защищенного режима ???)

Регистры процессора принято делить на 7 групп:

1. Регистры общего назначения (регистры данных) (EAX, EBX, ECX, EDX, 32 разрядные)
2. Регистры-указатели (EDI, ESI, EBP, ESP, 32 разрядные)
3. Сегментные регистры (CS, DS, ES, SS, FS, GS, 16-ти разрядные)
4. Управляющие регистры (CR0, CR1, CR2, CR3, 32 разрядные); CR - Control Register
5. Регистры системных адресов (GDTR, IDTR, LDTR, TR - Task Register, 16-ти разрядный)
6. Отладочные регистры
7. Тестовые регистры
8. Счётчик команд (EIP)

9. Регистр флагов (EFLAGS)

CR0 - регистр слова состояния.

31	...	4	3	2	1	0
pg	...	et	ts	em	mp	pe

pg - paging enable 0 - сегментами по запросу, 1 - страничное преобразование.
pe - protection enable 0 - в реальном режиме, 1 - в защищенном.
CR1 - зарезервирован.
CR2 - регистр линейного адреса ошибки обращения к странице, т.е. страничной неудачи.
CR3 - регистр базового адреса каталога таблиц страниц.
GDTR - регистр базового адреса таблицы глобальных дескрипторов.
IDTR - Регистр базового адреса таблицы дескрипторов прерываний.
LDTR - регистр таблицы локальных дескрипторов.
GDTR и IDTR содержат линейный адрес начала таблицы, в LDTR не хватает места для линейного адреса - это селектор. **Линейный адрес** - адрес, адресующий байт памяти.
EFLAGS - флаги в младших битах соответствуют реальному режиму. Например, 9-й бит - IF (Interrupt flag); 17 бит - VM - virtual mode - спец режим (V86). В режиме VM работает виртуальная память. Когда PE = 1, то VM должна быть в 1.
Если в PE = 1 в CR0 и VM = 1 в EFLAGS, то процессор работает в V86
В современных системах используют страничное управление памятью. Если PE и PG - выполняется пейджинг.

Таблица глобальных дескрипторов (GDT)

GDT описывает сегменты физической памяти (доступные всем). CNP архитектура - равноправные процессоры с общей памятью ⇒ GDT одна на компьютер.
Доступ к дескриптору осуществляется при помощи селектора. ВАЖНО: сегментные регистры в защищенном режиме называются **селекторами**.

15-3: index	2	1	0

0, 1 - RPL requested privilege level (уровни привелегий 0 - система, 3 - приложения. 1, 2 - хз)
2 - TI table indicator
index - смещение в ТГД.
GDT начинается с нулевого дескриптора (он пустой).
Формат дескриптора сегмента (8 байт):
0-1 - limit - размер сегмента;
2-3 - base low
4 - base middle
7 - base high
5 - attribute 1

(8 бит) 0 - Attach, устанавливается автоматически при доступе к сегменту, если равен 0, то обращения к сегменту не было;
1 - 3 - тип сегмента (code segment, data segment + права доступа, STACK, причем:
1 бит: **(сегмент кода)** 0 - чтение из сегмента запрещено, но не для выборки команд, 1 - чтение разрешено; **(сегмент данных)** 0 - модификация сегмента данных запрещена, 1 - разрешена.
2 бит: CODE: бит подчинения (0 - подчиненный, 1 - обычный), DATA & STACK: 0 - DATA, 1 - STACK.
3 бит: бит предназначения, 0 - DATA or STACK, 1 - CODE.
4 (S) - 1 - обычный сегмент, 0 - системный объект.
5-6 - DPL - уровень привилегий дескриптора.
7 - Present, бит присутствия. 0 - сегмент в памяти, 1 - не в памяти.

6 - attribute 2

(тоже 8 бит) 0-3 - лимит.
4 - USER BIT (не используется);

5 - равен 0 (не используется);

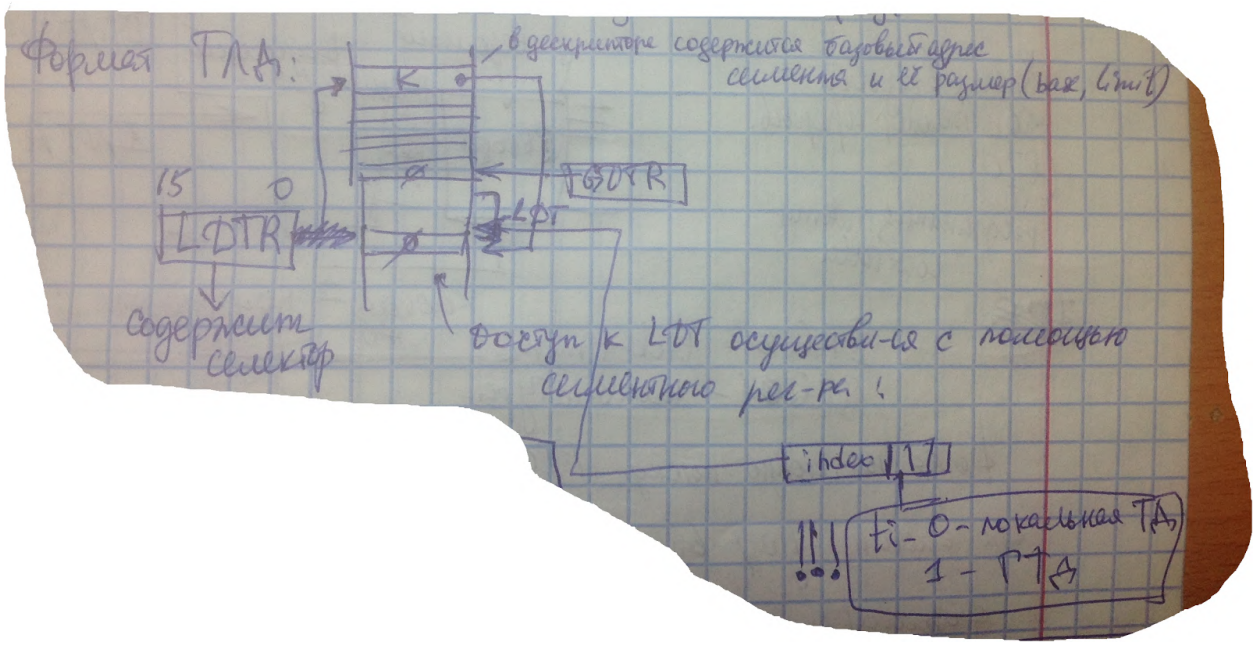
6 - Digit, 1 - операнды 32 разрядные, 0 - 16 разрядные

7 - **бит гранулярности**, если он установлен в 1, то память считается в 4 Кб.

Таблица локальных дескрипторов (сегментное виртуальное АП) (LDT)

LDT описывает АП виртуальных процессов (таких таблиц столько, сколько виртуальных процессов). LDT- системная таблица и занимает физическую память.

Формат LDT:



В дескрипторе (локальном) содержится базовый адрес сегмента и его размер.

Доступ к LDT осуществляется при помощи сегментного регистра (там второй бит указывает 0 - локальная ТД, 1 - глобальная, также там содержится смещение в LDT).

Уровни привилегий

0-й - уровень привилегий ядра системы.

1-й и 2-й не используются.

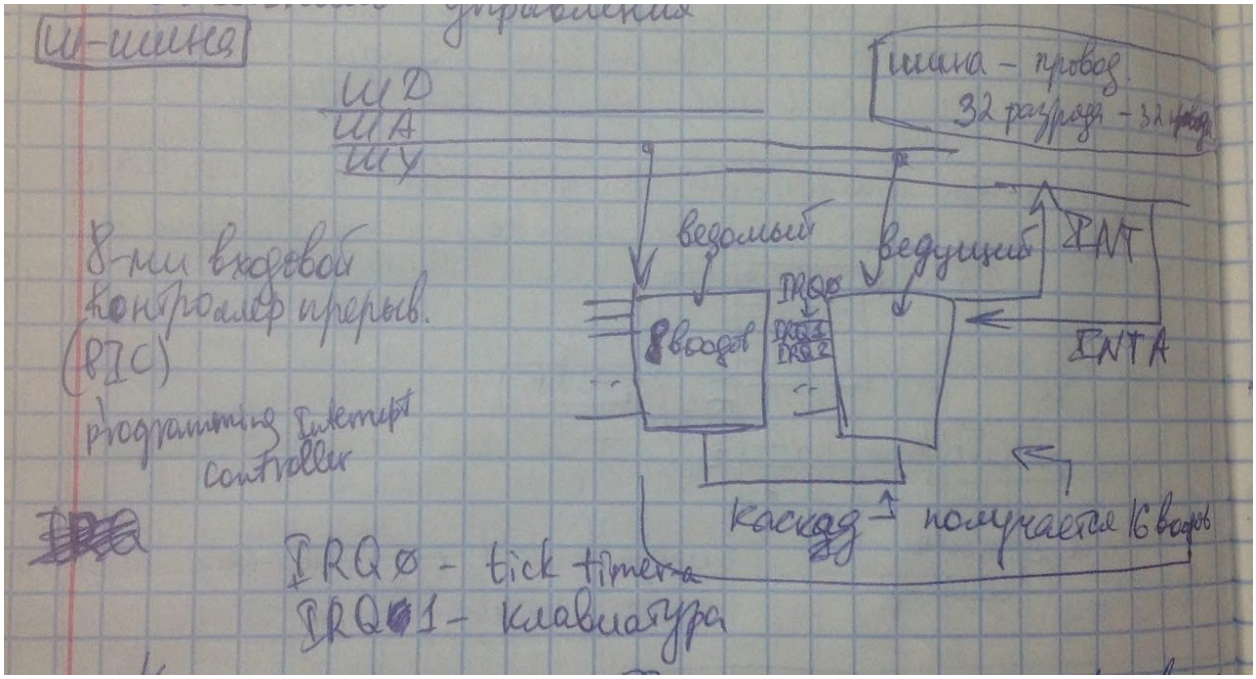
3-й - уровень привилегий пользователя (приложения).

Прерывания

Сигналы могут быть 3-х типов:

1. Данные: данные и команды
2. Адреса
3. Сигналы управления

8-входовой контроллер прерываний (PIC - programing interrupt controller)



Когда на любую ножку PIC приходит сигнал прерывания от внешнего устройства, то если он не замаскирован, то формируется сигнал INT, который по шине управления поступает на выделенную ножку процессора.

В конце цикла выполнения каждой команды, процессор проверяет наличие сигнала прерывания на этой ножке. Если прерывание пришло, то процессор посылает ответный сигнал INTA (INT Answer). Получив INTA, PIC формирует **вектор прерывания**: к базовому вектору прибавляется номер линии IRQ. В реальном режиме базовый адрес вектора ведущего контроллера = 8.

Вектора прерывания располагаются с нулевого адреса. (FAR адрес на обработчик прерывания)

В защищенном режиме: для адресации прерываний используется таблица дескрипторов прерываний (IDT) - их столько, сколько процессоров в компьютере. Только один обрабатывает Int8h, так как иначе будут траблы со временем и не только.

IDT: 0-й дескриптор значимый, первые 32 дескриптора для исключений, размер дескриптора 8 байт. PIC адресуется, по ША ему посылается маска. Можно перепрограммировать его на другой базовый вектор. При переходе в защищенный режим надо сделать базовым вектор 32, а не 0 (default).

Формат дескриптора прерывания

смещение (6-7 байты)	аттрибут*	резерв	селектор	смещение

*- содержит 0-3 - тип ловушки, 4 = 0, 5 - DPL(2 бита) см. GDT, 7 - бит Present, если он равен 0, то вектор неиспользуемый.

В дескрипторе прерывания содержится селектор и смещение. С их помощью получаем точку входа в обработчик прерывания. Классическое название обработчика прерывания - ISR - Interrupt Service Routine.

Порт - адрес. Существует 2 АП - АП ФП (команда mov, Memory Mapping, отображение на память) и АП портов ввода-вывода (команды in/out, IO Mapping, отображение на устройства ввода-вывода)

Дальше код перехода в защищенный режим…

Сохранение масок прерываний для ведущего и ведомого контроллеров

Запрет маскируемых и немаскируемых прерываний.

Разрешение …

Линия A20

В реальном режиме в intel 20 адресных линий. Адресная 20 линия существует т.к. разрядность 32 или 64 бита….

ПК стартует в реальном режиме, при старте 20 линия обнулена, при переходе в защищенный??? режим обнуление снимается.

\\Открыть линию A20 IN AL, 92h	\\ Закрыть линию A20 IN AL, 92h

OR AL, 2 OUT 92h, AL	AND AL, 0fdh OUT 92h, AL

В спецификации EMS - extended memory specification (растянутая память). Увеличивает объем доступной в реальном режиме памяти.

Существует еще более упоротая схема управления памятью - PAE там не просто таблица гиперстраниц, а еще есть каталоги таблиц страниц, долго, но зато дает доступ к 64 гигам оперативы.

Про **TLB кэш** без сильных подробностей.

TLB, кэш команд и кэш данных находятся в том же кристалле, что и процессор, но только 1 кэш (**теневые регистры,** сопоставленные сегментным регистрам, в них загружается информация из дескрипторов сегментов) принадлежит процессору. ???

Во всех этих кэшах реализована схема: 4-направленный, ассоциативный по множеству буфер.

Все кэши частично являются ассоциативными.

В TLB находятся адреса физических страниц, к которым были последние обращения. К кэшу TLB обращаемся по виртуальному адресу. Сначала адрес физ. страницы ищется в кэше. Если он не найден, то происходит обращение к таблице страниц.

Если все элементы кэша заняты, то происходит замещение по схеме достоверности LRU. Для этого в кэше есть блок достоверности:

0 бит - бит достоверности (бит обращения)

1-3 биты - реализация алгоритма псевдо LRU.

При сбросе процессора/очистке кэша все биты достоверности сбрасываются в 0. Когда ищется замена, ищется первый недостоверный бит.

Если все биты достоверности установлены в 1, то для замещения выбирают строку по алгоритму ниже (а может и нет. Я не уверен, что псевдо ЛРУ помогает выбрать наименее часто используемую страницу, как это должно быть в ЛРУ).

Псевдо LRU

Биты 1-3 (b0, b1, b2, см. картинку) блока достоверности модифицируются при каждом “попадании” (страница находится в кэше, похоже на страничную удачу) или заполнении страницы следующим образом:

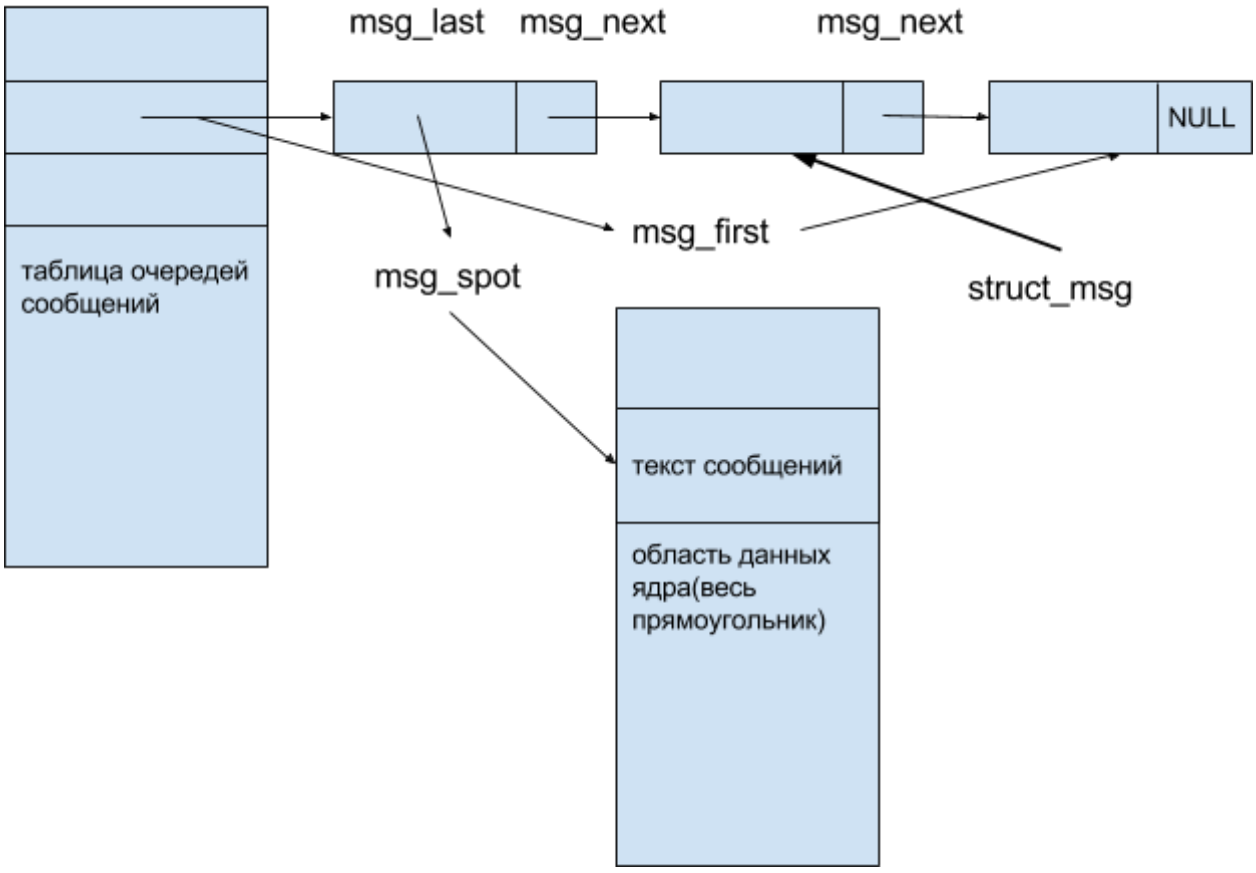
- если последнее обращение было в L0/L1 (см. картинку), то b0 устанавливается в 1, при L2/L3 - в 0.
- если последнее обращение в паре L0/L1 было к L0, то b1 в 1, иначе в 0.
- если последнее обращение в паре L2/L3 было к L2, то b2 в 1, иначе в 0.

Значение TLB велико (до 99% (?) удовлетворенных запросов на доступ страниц) именно потому, что реализован максимально близкий в LRU алгоритм.

При переключении процессов теряется актуальность содержимого TLB, и это проблема.

Очереди сообщений

Очереди сообщений(ОСб) - средство обмена сообщениями между параллельными процессами. Оно отражает особенность такого ресурса как сообщение. Само название предполагает, что в ядре создается очередь, в которую процесс может поместить или взять сообщение. ОСб поддерживаются в ядре соответствующей таблицей. В ней отслеживаются все созданные ОСб. <sys/msg.h>



//каждая строка описывает 1 созданную очередь сообщений - очередь типа FIFO. Очередь должна иметь указатель на голову и хвост

Когда процесс передает сообщение в конкретную очередь, ядро создает для этого сообщения новую запись и помещает ее в конец связного списка записей указанной очереди. В каждой такой записи указывается тип сообщения, размер сообщений(число байт) и указатель на область данных ядра системы, в которой фактически находится сообщение.

Ядро копирует сообщение из адресного пространства процесса-отправителя в адресное пространство ядра системы. Причем, процесс-отправитель может завершиться, при этом сообщение останется доступным для чтения другими процессами.

Выборка сообщений

Когда какой-то процесс выбирает ??? из очереди, ядро копирует сообщений в адресное пространство процесса-получателя, после чего запись удаляется. Процесс может выбрать сообщение из очереди следующими способами:

Процесс может взять

1) самое старое сообщение не зависимо от его типа
2) сообщение, идентификатор которого совпадает с идентификатором, который указал процесс. Если существует несколько сообщений с указанным идентификатором, то выбирается самое старое
3) сообщение, числовое значение типа которого есть наименьшее из меньших???, или равное значению типа указанного процессом. Если удовлетворяет несколько сообщений, то берется самое старое.

Если вспомнить диаграмму состояний блокировки процесса при отправке сообщений. Все эти состояния возможны, если есть строгий протокол взаимодействия - каждое сообщение требует подтверждения.

Механизм очередей позволяет избежать блокировок - процесс не обязан блокироваться в ожидании получения или при отправке сообщений.

Блокировки замедляют процесс выполнения программы. Часто они неизбежны. Если нужна обязательная синхронизация работы процессов, то от блокировок никуда не деться.

struct msgbuf { long mytype; char mytext[MSGMAX]; };

Системные вызовы для работы с сообщениями: msgget(), msgctl(), msgsnd(), msgrcv();

Пример. Программа создает новую очередь сообщений с ключевым идентификатором 100 и устанавливается права чтения/записи - владельцу, только чтение для группы, запись - остальным. Если msgget выполнен успешно, процесс передаст сообщение “hello” тип 15 и указывает, что данный вызов не блокирующий.

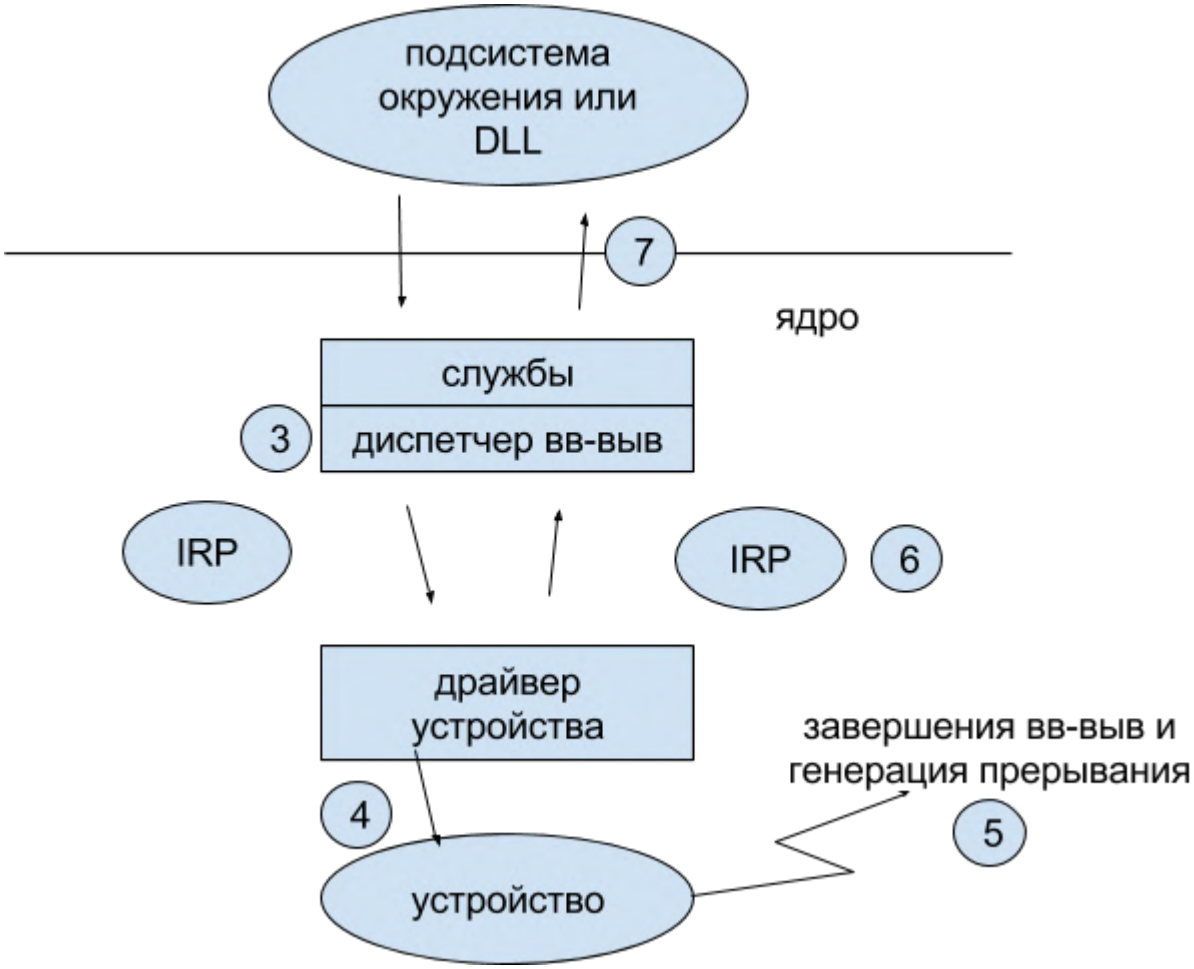
```
#include <string.h>
#include <sys/ipc.h>
#include <sys/msg.h>
#ifndef MSGMAX
#define MSGMAX 1024
#endif
struct msgbuf {
    long mtype;
    char mtext[MSGMAX];
} mobj = {15, "Hello"};

int main()
{
    int fd = msgget(100, IPC_CREATE|IPC_EXC|0642);
    if(fd==(-1)||msgsnd(fd,&mobj,strlen(mobj.mtext)+1,IPC_NOWAIT)
        perror("msg");
    return 0;
}
```

Обработка прерываний в windows

Диаграмма Шоу показывает, какие действия происходят в системе, когда происходит ввод-вывод.

//Быстрое прерывание обрабатывается от начала до конца. Его выполнение прервать нельзя. Остальные делятся на две части - быструю и отложенную. Когда возникает аппаратное прерывания, сигнал прерывания поступает на контроллер прерываний, который формирует вектор прерывания, который используется в качестве смещения в таблице дескрипторов прерываний, в которой находится дескриптор обработчика прерывания - содержит селектор и смещение. По селектору получаем базовый адрес сегмента кода, в котором находится обработчик прерывания., который мы используем, чтобы получить точку входа в нужный обработчик.

Обращение к одноуровневому драйверу:



(1) Передача запроса ввода/вывода черед DLL подсистемы
(2) DLL-подсистема вызывает службу NtWriteFile
(3) диспетчер ввода-вывода создает описывающий запрос вв/выв - IRP-пакет (input/output request packet), который состоит из 2-х частей: фиксированный заголовок(тело пакета) и один или более блоков стека. Заголовок содержит данные о типе и размере запроса (синхронный или асинхронный и т.д.). Блок стека IRP-пакета содержит номер функции, связанные с функцией параметры и указатель на файловый объект вызывающей программы. Причем содержание пакета IRP меняется в процессе выполнения запроса). Диспетчер посылает IRP-пакет драйверу устройства и вызывается функция IOCallDriver.
(4) Драйвер передает на устройство данные из IRP-пакета и инициализирует операцию вв-вывода, т.е. драйвер по шине данных посылает контроллеру устройства соответствующие команды. Управление устройством берет на себя контроллер.
(5) По завершении вв-вывода генерируется прерывание от устройства вв-вывода - аппаратное прерывание
(6) Обработка прерывания и возвращение состояния операции, т.е. сообщение об ошибке или успехе. Драйвер вызывает функцию IOCompleteRequest диспетчера вв-вывода, чтобы уведомить его о завершении вв-вывода.
(7) Драйвер вызывает IOCompliteRequest диспетчера ввода вывода, чтобы уведомить его о завершении в/в. Запрос, описанный соответствующим IRP пакетом завершается. // дальше непроверенная инфа: Диспетчер ввода вывода ????? или Драйвер обслуживает устройство - вызывается обработчик прерывания, который является одной из точек входа драйвера, чтобы проинформировать ??????? ЧТО ТУТ ВООБЩЕ ПРОИСХОДИТ!!!! у меня тоже все грустно(

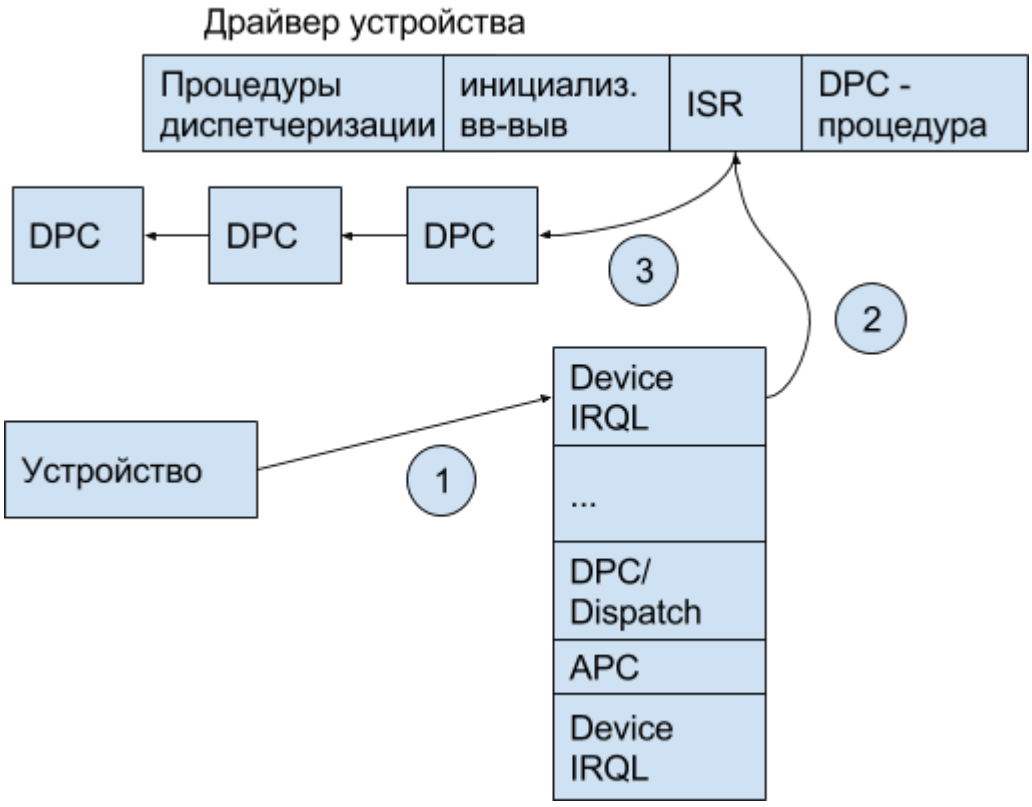
Обслуживание прерывания

1-я ФАЗА

Завершая передачу данных, устройство генерирует прерывание. После этого действовать начинает ядро Windows, а именно диспетчер вв-выв и драйвер устройства. Когда устройство генерирует прерывание, процессор передает

управление обработчику ловушки (trap), который называется ISR(interrupt service routing). Для этого устройство по таблице IDT. ???В Windows ISR обычно обрабатывается в 2 этапа.

1 этап. Вызов ISR выполняется на уровне IRQL device (IRQL соответствующего устройства). В течении времени, необходимого для сохранения состояния устройства и запрета дальнейших прерываний от этого устройства. Потом ISR помещает DPC(Deferred Procedure Call) в очередь и завершается.



//выполняем сохранение в буфере ядра, чтобы впоследствии передать их в адресное пространство процесса.

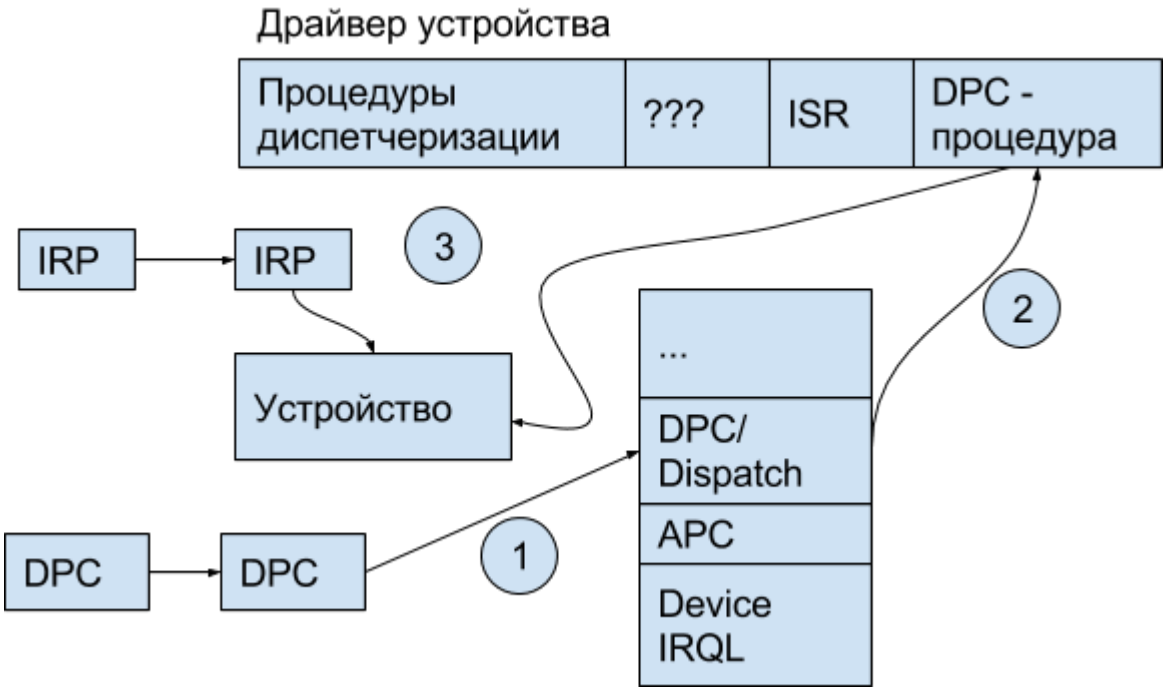
DPC завершает обработку прерываний - выполняет действие ???

1) Устройство генерирует прерывание
2) Диспетчер прерываний передает управление ISR
3) ISR перехватывает прерывание от устройства и помещает DPC в очередь.

DPC выполняется уже на более низких IRQL при разрешенных прерываниях.

2-я ФАЗА

(1) IRQL понижается и начинается обработка DPC
(2) Диспетчер прерываний передает управление DPC-процедуре. Применение DPC делает возможным прерывание от уровня Device IRQL до DPC|dispatch
(3) DPC-процедура начинает обработку следующего запроса IRP из очереди устройства, после чего заканчивает обработку прерывания.

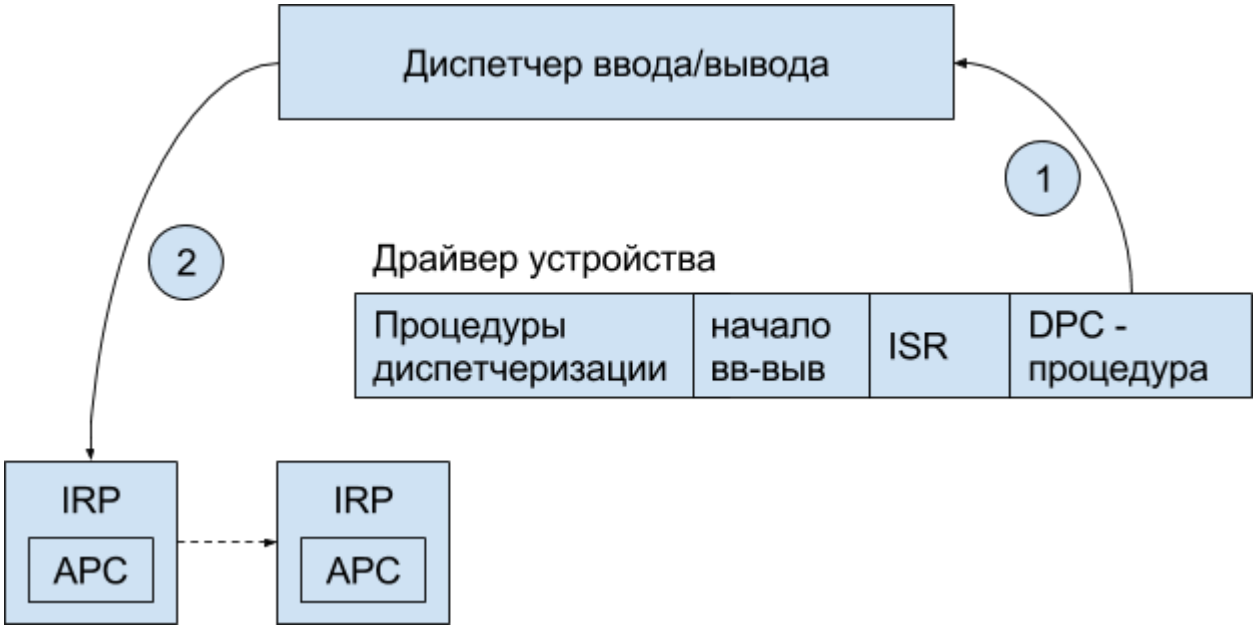


3-я ФАЗА Завершение обработки запросов вв-выв.

После выполнения DPC-процедуры, драйвер устройства еще должен выполнить ряд действий. 3-я фаза начинается с вызова драйверов устройства функции IOComleteRequest, которая уведомляет диспетчер вв-выв о завершении обработки запроса, указанного в IRP-пакете. Выполняемые на этом этапе действия завершают операцию вв-вывода. В любом случае подсистема вв-вывода должна скопировать из системной памяти данные в виртуальное адресное пространство вызывающей программы.

При синхронном завершении - это пространство является текущим и используется напряму.

При асинхронном диспетчер должен отложить до момента получения доступа к адресному пространству вызвавшей программы. Эту задачу диспетчер вв-выв решает путем постановки в очередь вызыв. программы асинхронной процедуры APC.

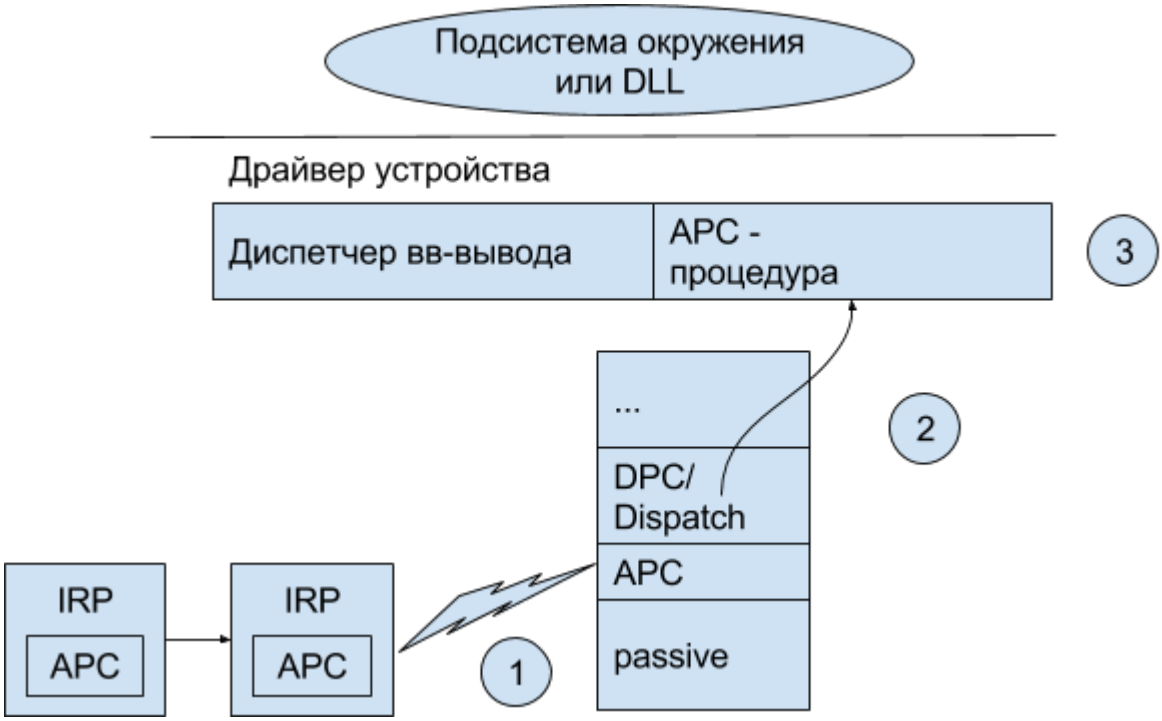


(1) DPC-процедура вызывает диспетчер вв-выв для завершения запроса на вв-выв.

(2) Диспетчер вв-выв ставит APC в очередь для завершения запроса на вв-выв в контексте вызывающей программы, т.е. APC выполняется в контексте программы, которая вызвала запрос на вв-вывод.

DPC выполняется в контексте любого потока.

APC - Async Procedure Call



(1) При следующем выполнении потока вызывающей программы происходит APC прерывание.
(2) Диспетчер прерывания передает управление APC-процедуре
(3) APC-процедура записывает данные в адресное пространство потока вызывающей программы и переводит исходный дескриптор файлов в свободное состояние для синхронного вв-выв

по завершении операции вв-выв IRP-пакет удаляется.

ISR должна выполнять минимум действий по обслуживанию прерывания от своего устройства:

1) сохранить информацию о состоянии прерывания
2) отложить передачу данных и других не критических ситуаций до снижения IRQL до уровня DPC-dispatch.

Программные каналы

pipe-канал - симплексная (односторонняя) связь.

Для дуплексной (двусторонней) нужны два канала.

Именованный программный канал (ИПК) это специальный файл (помечается буквой p в системе), который имеет имя (виден в каталоге файлов).

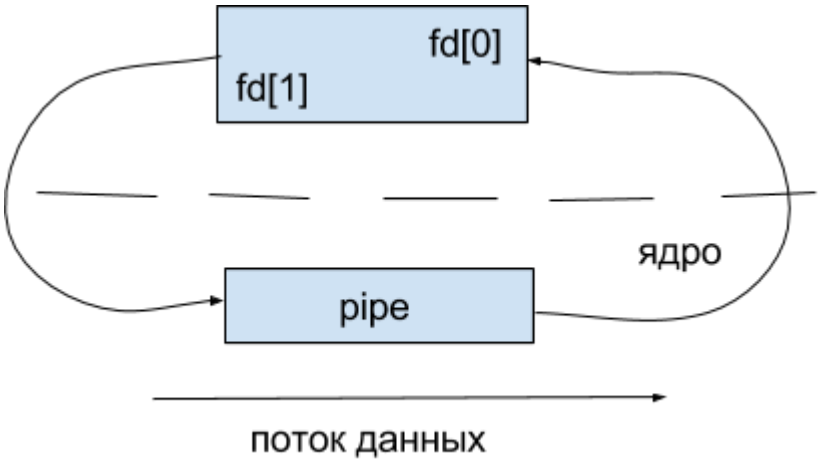
Неименованный программный канал (НПК): он не имеет имени и в файловой системе не виден, но у него есть дескриптор. НПК могут пользоваться только процессы-родственники (т.к. потомок наследует все дескрипторы открытых файлов предка - особенности реализации fork()).

Средство управления программными каналами - массив файловых дескрипторов.

Системный вызов pipe:

```
1.  int pid, fd[2]; int status;
2.  char msg[25] = "";
3.  char str[] = "Example";
4.  if (pipe(fd)  == -1) { perror("pipe"); exit(1);}
5.  if ((pid = fork()) == -1) { perror("fork"); exit(1);}
6.  if (pid == 0) // Child
7.  {
8.      close(fd[1]);
9.      read(fd[0], msg, 25);
10.     printf("%s\n", msg);
11. }
12. else
13. {
14.     close(fd[0]);
15.     write(fd[1], str, strlen(str));
```

```
16.     wait(&status);
17. }
```



Программный канал буферизуется на 3-х уровнях:

- системная память - в области данных ядра системы (1-й уровень);
- при переполнении системной памяти буферы, которые имеют наибольшее время существования, переписываются на диск. При этом используются стандартные функции управления файлами;
- если процесс записывает больше 4096 байт (4 Кб), то pipe буферизуется во времени, блокируя процесс, который пишет в трубу до тех пор, пока данные не будут прочитаны. Такое ограничение повышает эффективность, т.к. каналы небольшого размера целиком помещаются в ОП.

Каналы выделяются в системной области памяти, т.к. АП процессов защищены, и взаимодействовать они могут только через АП “третьей” стороны, которой и является область данных ядра системы.