Содержание

Monday, June 4, 2018 15:20

35. Алгоритмы диспетчеризации процессов с обратной связью.
36. Алгоритмы диспетчеризации процессов, применяемые в ОС реального времени.
37. Взаимодействие и синхронизация процессов. Проблема "критической секции".
38. Взаимодействие и синхронизация процессов. Проблема "поставщик-потребитель".
39. Взаимодействие и синхронизация процессов. Проблема "читатели-писатели".
40. Механизмы синхронизации: активное ожидание, семафоры.
41. Механизмы синхронизации: POST / WAIT.
42. Синхронизация посредством обмена сообщениями.
43. Тупики в ОС. Модель Холта.
44. Методы обнаружения, восстановления и предотвращения тупиков.
45. Прямой и косвенный ввод/вывод.
46. Монопольно используемые, разделяемые и виртуальные устройства.
47. Планирование запросов для последовательно-разделяемых устройств.
48. Физическая система ввода-вывода (косвенный ввод-вывод).
49. Организация ввода-вывода в пакетной мультипрограммной системе (косвенный ввод-вывод).
50. Организация ввода-вывода в диалоговой системе (прямой ввод-вывод).
51. Логическая система ввода-вывода (косвенный ввод-вывод).
52. Буферизация. Загрузка и разгрузка буферов. Режимы пересылок.
53. Простая и обменная буферизация.
54. Функции системы управления данными.
55. Организация доступа к информационному ресурсу.
56. Разграничение полномочий пользователей в ОС.

Часть I. Введение в ОС

13 марта 2018 г. 12:23

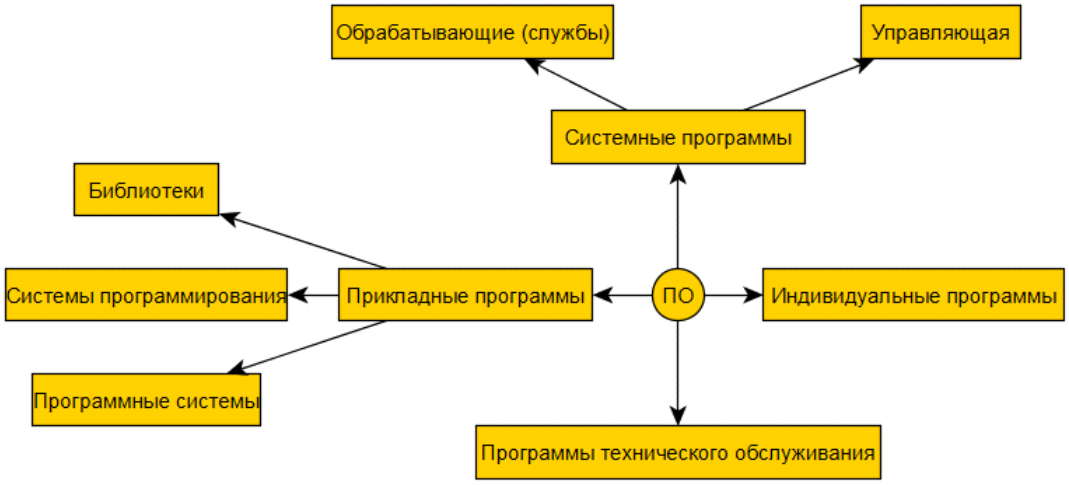
1. Определение понятия "ОС"

6 февраля 2018 г. 21:37

Лектор: Губкин Александр Федорович

Тема 1. Определение понятия "ОС"

1.1. Классификация программ



Индивидуальные программы - программы, которые пишутся разработчиком для личных целей и направлены на быстрое получение результата

Прикладные программы - программы, представляющие собой рыночный продукт и отчуждённые от разработчика

- **Системы программирования** - программы, предоставляющие некий абстрактный интерфейс для программирования в определённой объектной области. Включают в себя библиотеки функций для поддержки этой области, компилятор и т.п.
- **Программные системы** - предоставляют графический интерфейс для работы в объектной области, не требующий программистских навыков

Программы технического обслуживания - тесно связанные с аппаратурой

Обрабатывающие программы - программы, выполняющие вспомогательные функции - **службы** в Windows и **демоны** в UNIX

Функции УП:

1. Управление ОП
2. Управление процессами, задачами
3. Управление уст.
4. Управление вводом/выводом
5. Управление данными, файловой системой

1.2. Определение понятия ОС

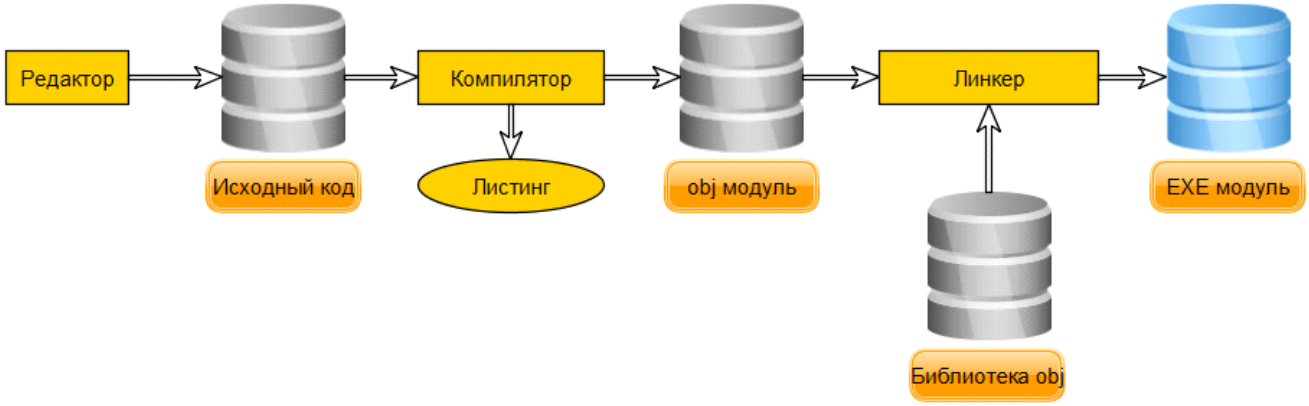
ОС - система программ, которая предназначена для обеспечения определенного уровня эффективности ПК за счет автоматизированного управления его работой и предоставляемого пользователю набора услуг

Пакетная система - система обработки данных.
Критерий эффективности пакетной системы - количество операций в единицу времени
Критерий эффективности диалоговой системы - время отклика

Дистрибутив - загрузочный диск/файл, содержащий все, что содержится в ОС.

1.3. Системы программирования

Классическая система (ex. C++)



Компилятор:

1. Препроцессорная обработка
2. Лексический анализ =>
    - Таблица имен, констант
    - Программа в виде лексем
3. Синтаксический анализ - осуществляет грамматический разбор, строит промежуточную форму
4. Семантический анализ - определяет типы переменных, области видимость и т.п.
5. Генерация кода => объектный модуль

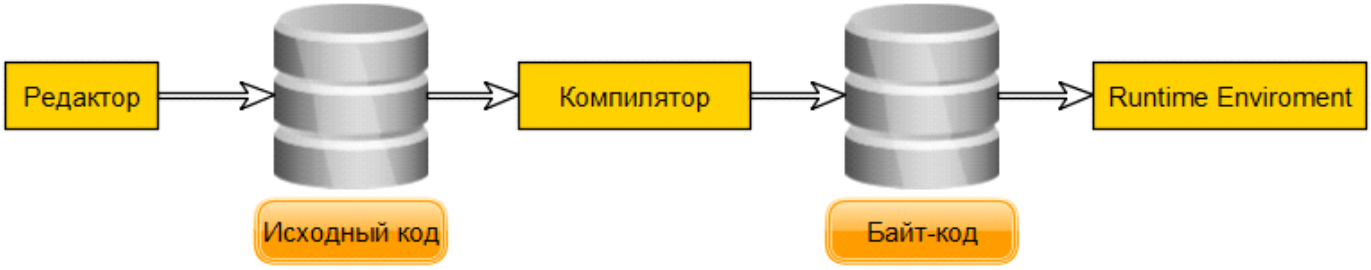
Объектный модуль
Имеет следующую структуру:

1. Таблица внешних символов
    a. Таблица внешних имен (global vars) - имена и адреса всех объектов, которые видно из других модулей.
    b. Таблица внешних ссылок (extern) - имена и адреса всех обращений к внешним объектам.
2. Код

Процесс создания модуля простой структуры из объектных файлов описан в пункте 5.2.1

Определение переменной - выделение под неё памяти.
Если переменная определена в одном модуле, а используются в другом, то пишется extern.
Если не подключается какая-либо библиотека, т.е. внешняя ссылка, на неё нет внешнего имени.

Java, C#



JRE - EXE, интерпретирующий байт-код. У C# аналогичный компонент - RE-Framework.
Управляющая программа может выполнять только загрузочный модуль (exe)

Читать ОС, ч1 - типы ОС

2. Типы ОС

13 февраля 2018 г. 19:30

Тема 2. Типы ОС

2.1. История развития ОС *(подробнее ОС, ч1)*

Доисторический период (1945-1952). Разработка систем программирования
В 1957 создан α-язык для ламповой машины
IBM 7090, 7096 - вычислительные задачи
Минск 2, 22, 32; М20, М200; БЭСМ, -4, -6; Эльбрус

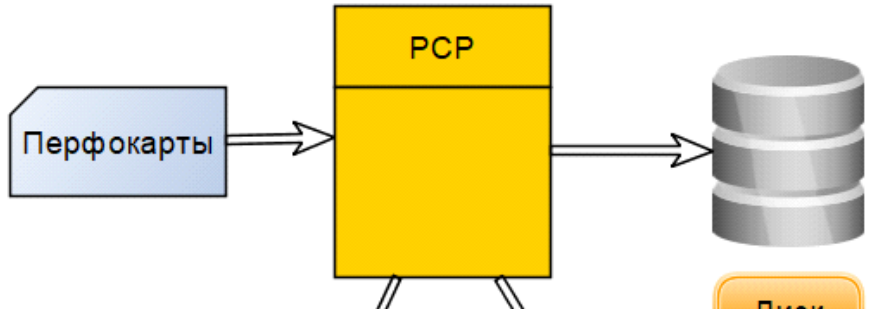
В 1962 IBM создала IBM 360 с байтовой адресацией. ПО к ней появилось в 1964.
OS360 - первая реальная мультипрограммная система
Тогда в ЭВМ не было терминалов, поэтому разрабатывались пакетные системы

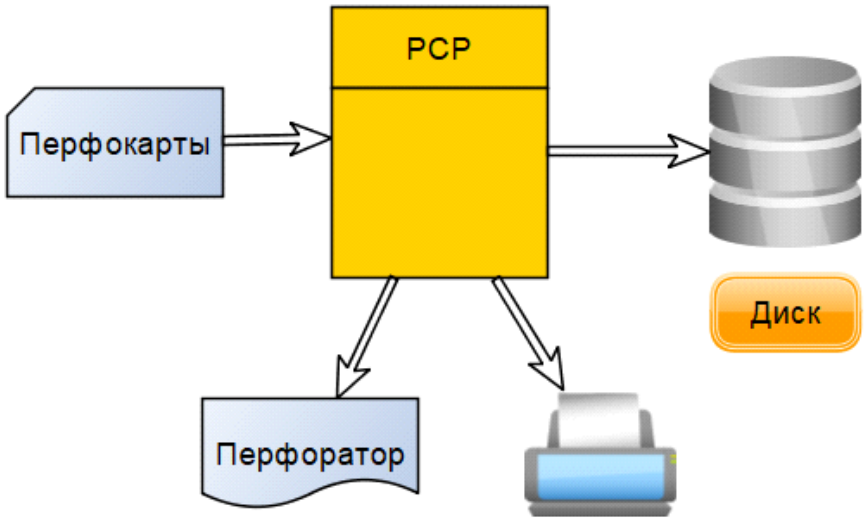
Таблица 1.

Год	Достижения в области ПО	Тип ОС
1945 - 1950	Библиотеки прикладных программ. Макрогенерация. Ассемблеры.	Отсутствие ОС. Доисторический период
1950 - 1955	Методы доступа. Буферизация. Компоновщики. Перемещающие загрузчики.	
1955 - 1960	Система прерываний и обработчики прерываний. Таймеры. Контрольная точка. Управление файлами.	Появление однопрограммных ОС
1960- - 1965	Классическое мультипрограммирование. Создание системы IBM360 и OS360 MFT	Появление мультипрограммных ОС
1965 - 1975	Динамическое распределение памяти ОС360 MVT. Диалоговые системы. Телеобработка. Виртуальная память IBM370 OS370 SVS, VS1, VS2, MVS. Виртуальные машины VM370.	
1975 - 1980	Сети ЭВМ. Сетевые и распределенные ОС.	Сетевые и распределенные ОС. Мультипроцессорные системы
1980 – 1990	Базы данных. Мультимедиа	
1990 – наши дни	Проникновение ВС в повседневную жизнь человека. Создание ОС, имеющих дружественный графический и мультимедийный интерфейс.	Универсальные ОС, поддерживающие мультипроцессорные платформы.

2.2. Однозадачные системы

PCP - Prime Control Program - система для IBM 360



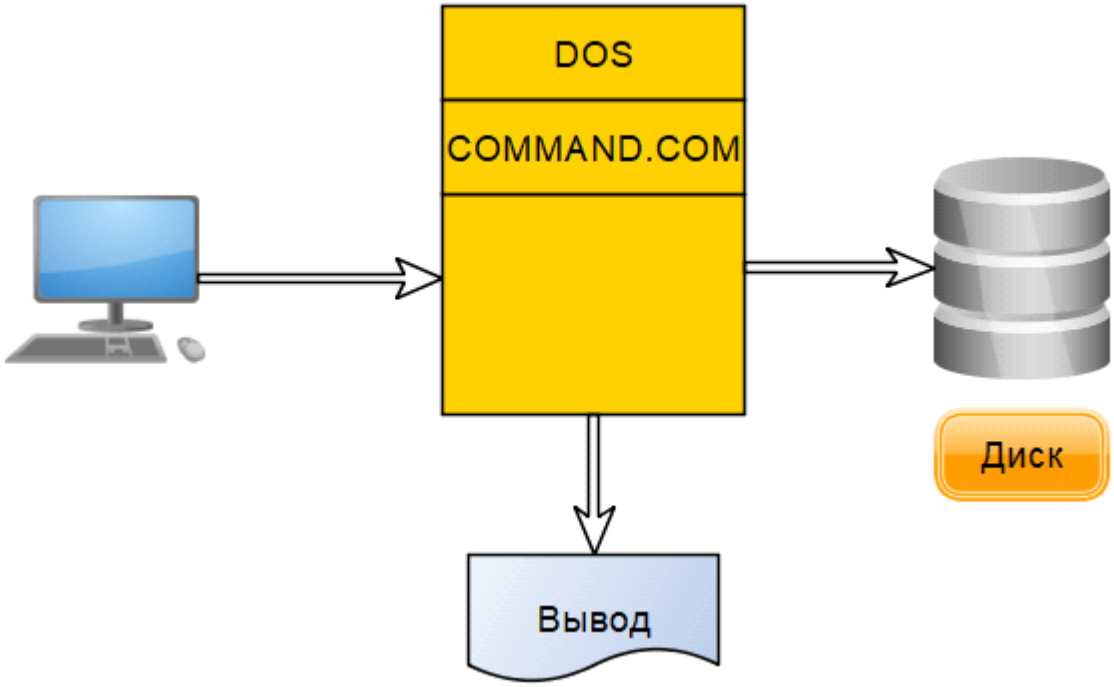


Для работы на этой системе программист должен был составить задание на **Job Control Language**

```
//имя JOB
//step1 EXEC
```

Пользователь динамически вводит команды, командный процессор разбирает её и организует выполнение. Все ресурсы отдаются выполняемой задаче

Другой пример однозадачной ОС- **DOS.**
В DOS есть процессор командного языка **COMMAND.COM**. Он обеспечивает интерфейс пользователя.

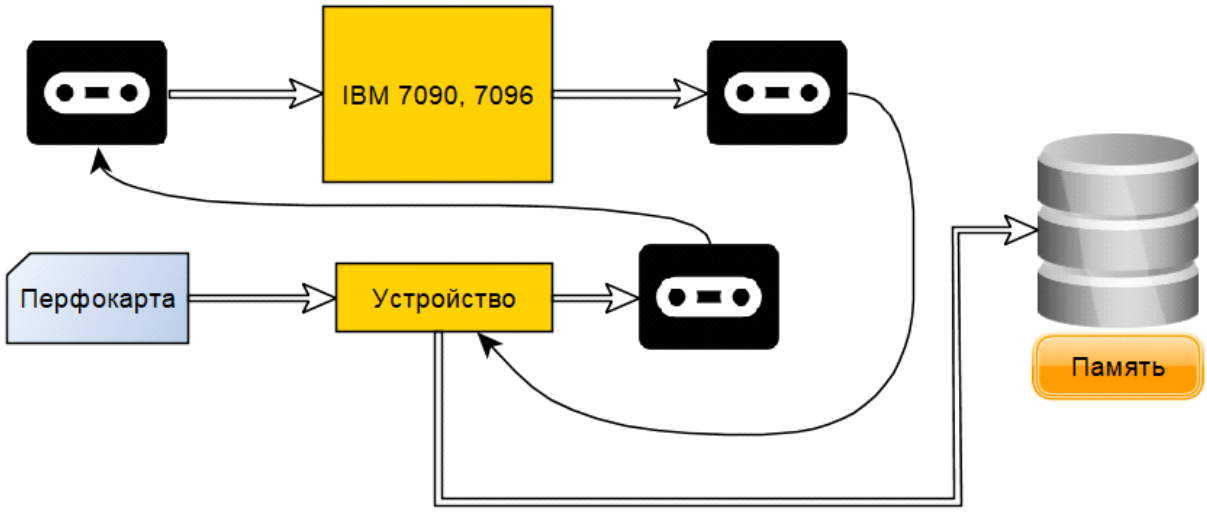


Состав такой системы:

- Процессор командного языка
- Система управления внешними устройствами
- Файловая система

2.3. Пакетный монитор

Производительность пакетных систем измеряется в числе решённых зада за единицу времени. Проблема однозадачной системы в том, что большую часть времени система простаивает, ожидая пользователя. Очевидное решение - исключить пользователя из процесса.

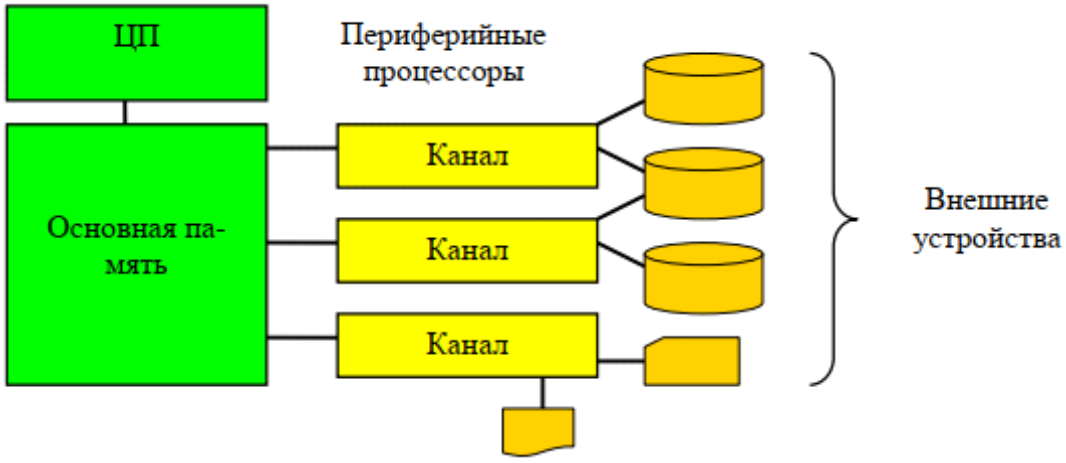


Вводом данных в таком случае будет заниматься более дешевое устройство (например, IBM 1401). Это устройство - **система подготовки данных** - подготавливала данные, вносимые на перфокартах и переносила их на магнитную ленту, которая после этого использовалась в мощной **расчётной системе**. Это обеспечивало почти 100%-ю загрузку системы.

2.4. Мультипрограммные системы

Следующий шаг оптимизации - создание системы, способной обеспечить выполнение нескольких задач, разделяющих устройства системы.
PCP > MFT > MFT+ > MVT > SVS > VS1 > VS2 = MVS

Операции ввода\вывода переносятся на периферийные процессоры, что освобождает ЦП.



В современных ПК используют шины, и рассмотренная технология не используется.

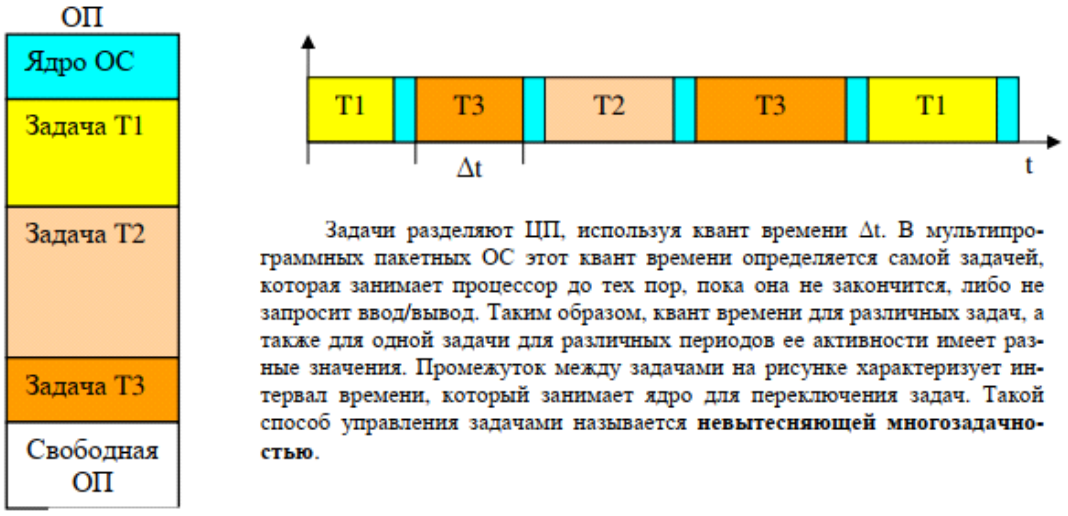
Особенности

1. Чтобы избежать простоя ЦП при переключении задач, ввели

мультипрограммирование. Главная цель - максимальная загрузка устройств системы

2. Для этого потребовалась **защита памяти**, во избежание пересечения задач
   Защита памяти **по ключу:** память выделяется блоками. С каждым блоком связан регистр, в который записывается ключ. У программы тоже есть ключ
3. Использование **прерываний** для переключения задач

Мультипрограммирование - работы с несколькими задачами в ОП

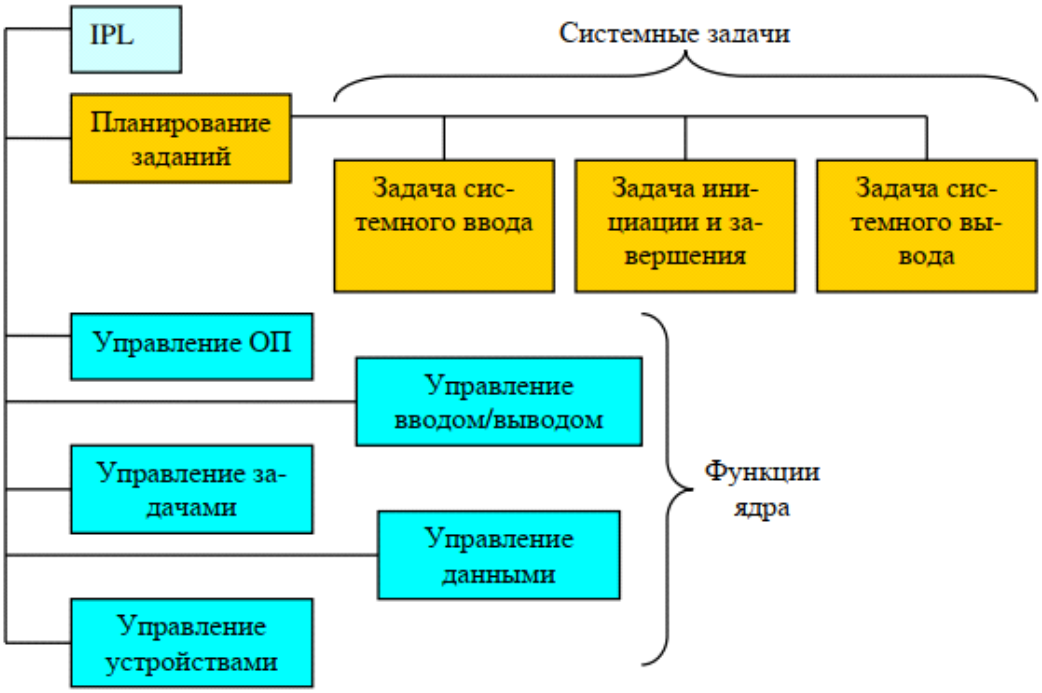


Задачи разделяют ЦП, используя квант времени Δt. В мультипрограммных пакетных ОС этот квант времени определяется самой задачей, которая занимает процессор до тех пор, пока она не закончится, либо не запросит ввод/вывод. Таким образом, квант времени для различных задач, а также для одной задачи для различных периодов ее активности имеет разные значения. Промежуток между задачами на рисунке характеризует интервал времени, который занимает ядро для переключения задач. Такой способ управления задачами называется **невытесняющей многозадачностью**.

В один момент времени одна задача занимает ЦП. Время может различаться, t - время переключения - составляет накладные расходы. Задача покидает процессор по своей инициативе - например, из-за конца задачи или ввода/вывода. Это называется **невытесняющая мультизадачность**.

Смесь задач - качественная характеристика совокупности выполняемых задач. Разные виды задач требуют разные ресурсы - например, **диалоговые (информационные) задачи** активно работают с внешними устройствами, но мало занимают ЦП, а **счётные задачи**, напротив, требуют много времени ЦП. Логично дать счётной задаче возможность занимать ЦП, пока информационная работает с устройствами.

Структура управляющей программы (УП, Control Program)



В мультипрограммной системе существует два уровня управления:

- **Верхний уровень** - системные задачи (z/OS - **JES2, JES3 - Job Entry System**). Они организуют прохождение заданий через систему.
- **Прикладные задачи**

IPL - Initial Program Load - короткая программа, запускающая загрузку системы

Планировщик заданий - обрабатывает входной пакет заданий и управляет прохождение заданий через систему.

Пример текста на JCL

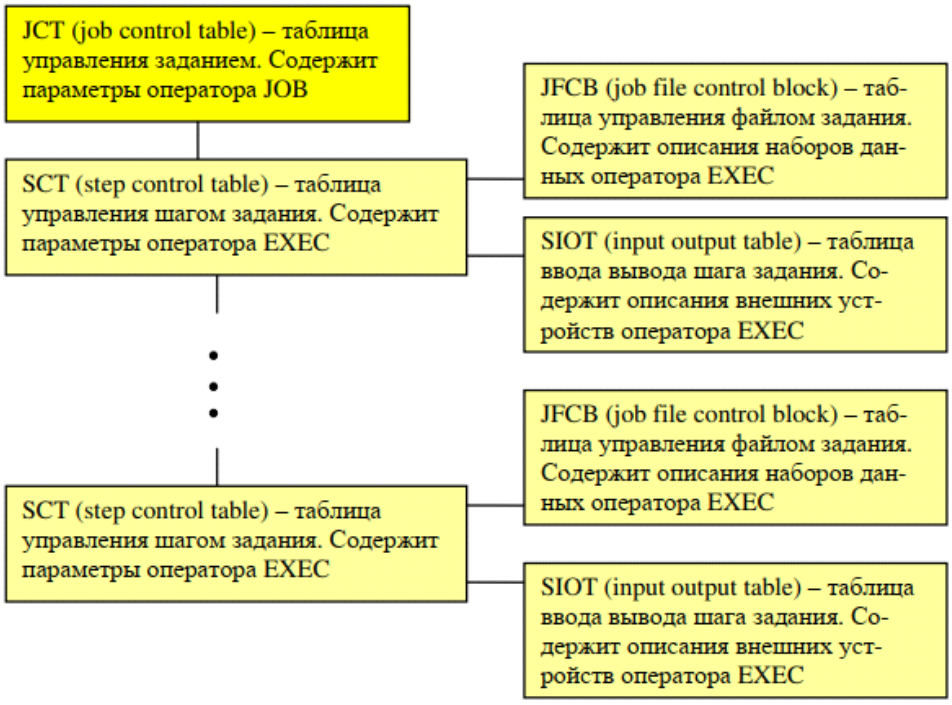
```
//task1                 JOB   <параметры оператора JOB>
//                      EXEC  PGM=FORTRAN <другие параметры оператора STEP>
//<имя DD оператора>    DD    <параметры DD оператора>
. . . . . .
//<имя DD оператора>    DD    <параметры DD оператора>
//                      EXEC  PGM=LINK <другие параметры оператора STEP>
//<имя DD оператора>    DD    <параметры DD оператора>
. . . . . .
//<имя DD оператора>    DD    <параметры DD оператора>
//                      EXEC  GO=*
//SYSIN                 DD    *
            <Входные данные программы>
//task2                 JOB   <параметры оператора JOB>
 . . . . . .<описание  второго задания>. . . . . .
//                                                  . . . . . .
```

- JOB определяет началу задания. Он содержит параметры для всего задания, например, CLASS описывает **класс**, в котором задание будет выполняться.
  Класс характеризует объем ОП и время выполнения.
- EXEC задает программу, которая будет выполняться на этом шаге

Ход обработки задания:

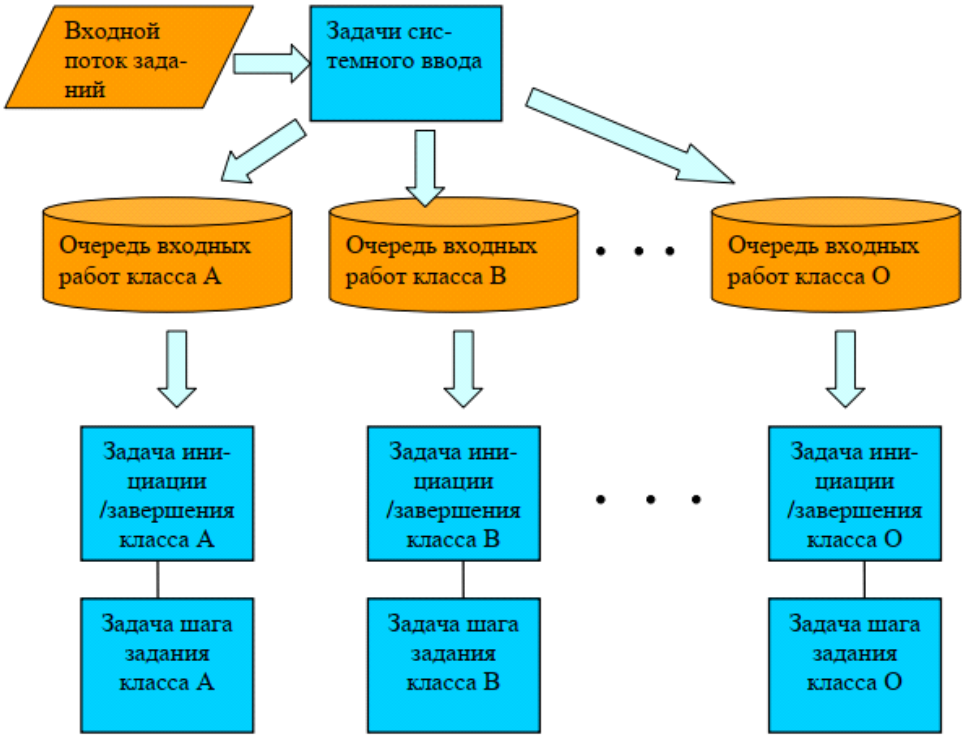
- Задача Системного Ввода вводит задание в систему, производит

разбор текста и создает **Job Control Table**:



JCT составляют **очередь входных работ**, каждое задание в которой принадлежит определённому классу. Таким образом формируется смесь задач - задачи одного задания или одного класса выполняются последовательно

- **Задача инициализации и завершения** обрабатывает очередь входных работ. Эта задача подготавливает шаг задания (выделяет ресурсы и т.п.), передает управление шагу, после чего выполняет операции по завершению шага (освобождает ресурсы и т.п.)



- **Задача системного вывода** управляет выводом данных - задачи выполняются, разделяя время ЦП, что не должно воздействовать на вывод. Для этого создается виртуальное устройство **- очередь**

выходных работ (до 36 различных).

- Очередь выходных работ состоит из **списков вывода**
  - Каждый список состоит из управляющих блоков и относится к одному заданию.

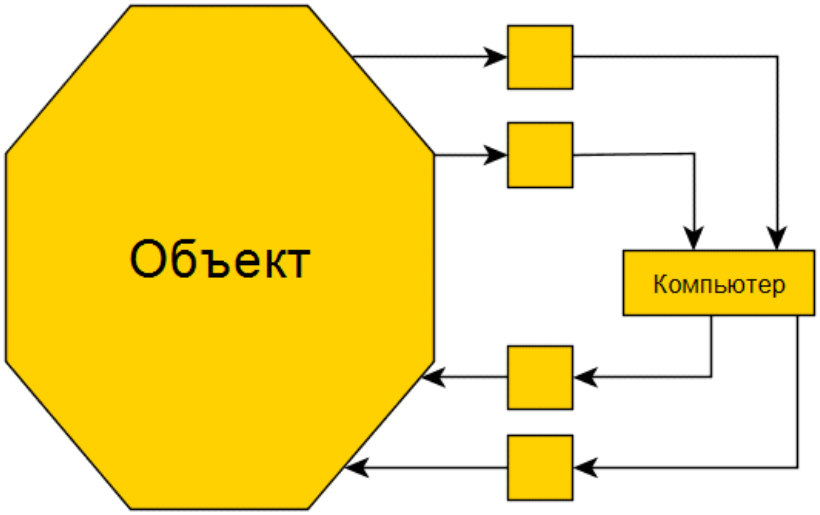
2.5. Диалоговые мультипрограммные ОС

Пакетные системы вышли на рынок раньше, т.к. не требовали устройств для диалога. Диалоговые системы должны обеспечить приемлемое время реакции на запрос пользователя. Пример диалоговой системы - UNIX.
Вытесняющая многозадачность - есть устройство, называемой **таймером**. Через определенные интервалы времени (вычисляется по алгоритму диспетчеризации) таймер запускает прерывание. Прерывание продолжает выполнение задачи и сохраняет состояние предыдущей. При этом Δt стремится не к бесконечности, а к константе - интервалу.
За этот интервал задача может закончится. Если задача не закончилась, она возвращается в очередь готовых задач.
Управляющая программа:

Командный процессор - слушает консоль. Может либо сам выполнить команду, либо запустить какую-либо программу. Это первая задача, которая стартует после загрузки ОС
Обработка данных выглядит так:
Open > read/write > Close.

2.6. ОС реального времени

- тоже мультипрограммная ОС. Главным образом отличается от диалоговой ОС фиксированным набором задач.



Система получает информацию от объекта через преобразователи и возвращает управляющие сигналы через определенный Δt
Пример - РЛС, работающая в двух режимах. Первый режим - поиск цели, второй режим - захват цели и её сопровождение. Реакция системы в данном случае должна быть гарантированно в заданное время.

Особенности системы реального времени:

1. Так как внешние события детерминированы, можно иметь набор задач с изученными заранее характеристиками. Это дает возможность изучить характеристики программ (по быстродействию, занимаемой памяти и т.п.) и получить фиксированное время отклика
2. Имеется частично детерминированный поток событий. Существуют объекты, требующие периодического обслуживания.
3. Простая структура входных и выходных данных

Типы систем реального времени с точки зрения масштаба времени:

1. Системы, в которых нарушение масштаба времени недопустимо - например, системы взлета/посадки.
2. Системы, в которых нарушение допустимо, но влечет потерю экономического эффекта - например, система резервирования билетов.

Типы систем с точки зрения конструкции

1. Встроенные (Embedded) системы - делаются в условиях жесткого масштаба времени для какой-либо сложной задачи.
2. Универсальные системы, не требующие столь жесткого соблюдения. Пользователь задает машине свои функциональные задачи.

3. Принципы построения мультипрограммных ОС

20 февраля 2018 г. 12:46

Тема 3. Принципы построения мультипрограммных ОС

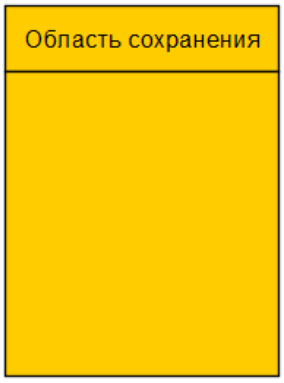
3.1. Модульность. Программный комплекс обладает свойством модульности, если он состоит из программных единиц, каждая из которых является:

- Функционально независимой (самостоятельной)
  Каждая функция должна выполнять определенную задачу. Это облегчает дальнейшее улучшение, сопровождение и поиск ошибок
- Имеет стандартную структуру
- Стандартным образом связывается с другими модулями

Загрузочные модули хранятся на внешних носителях в определенном формате (например, EXE)

Соглашение о связях (Linkage convention)
- устанавливает интерфейсы между модулями в ОС. Например, определяет вызов модуля и возврат управления в вызывающий модуль.

- **Без стека (z/OS MVS)**

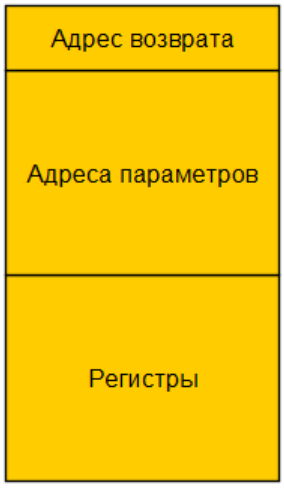


SAVE DS.18F

Система имеет 16 равноправных регистров, которые могут хранить как адреса, так и результаты вычисления. Вот некоторые:
  - R1 - адрес списка параметров
  - R13 - адрес области сохранения, вызывающей программу
  - R14 - адрес возврата
  - R15 - адрес точки входа при входе или Return Code (RC - код)

При входе в программу регистры имеют значения, установленные вызывающей программой. Вызываемая программа сохраняет эти регистры вместе с необходимыми параметрами в **области сохранения** предыдущей программы (по адресу R13).
Это более удобно, чем работа с сохранением данных в вызывающей программе.

- **Со стеком (Open System)**



Введены две команды, изменяющие состояние стека - PUSH и POP.
При вызове команды CALL в стек сохраняется адрес перехода к вызывающей программе, который достается командой RET.
Для передачи параметров вызывающая программа сохраняет данные в стек, а вызываемая - достаёт их оттуда.
Вызываемая программа при этом запоминает место в стеке, где были сохранены параметры и сохраняет дальше на стеке используемые регистры - чтобы избежать ошибок.

В разных языках высокого уровня соглашения о связях разные.

3.2. Принцип параметрической настраиваемости

В настоящее время для компьютерных систем существует широкий спектр устройств от различных производителей. ОС должна обеспечивать их корректную работу.
Когда система стартует, программа в BIOS - POST - проверяет конфигурацию машины. Есть критические вещи - клавиатура, экран, оперативная память и т.п., наличие которых необходимо.
Для каждого устройства есть **UCB (Unit Control Block)**. Список UCB содержит описание всех устройств. При подключении устройства создается новый UCB, где говорится состояние устройств и т.п.
Драйвер - программа, работающая с устройством. Драйвер посылает в машину некоторые команды и ждет ответа. Контроллер устройства инициирует прерывание с нужным вектором и передает управление драйверу.
Система распознает, какое устройство подключено с помощью технологии Plug and Play. Устройства хранят в себе информацию, позволяющие распознать устройство.

3.3. Принцип функциональной избыточности

Заключается во включении в ОС программных средств, выполняющих одну и ту же функцию, но с разной эффективностью в разных ситуациях.

Например, в управляющей программе существует несколько функций, обеспечивающих разные варианты синхронизации процессов.

3.4. Принцип функциональной избирательности

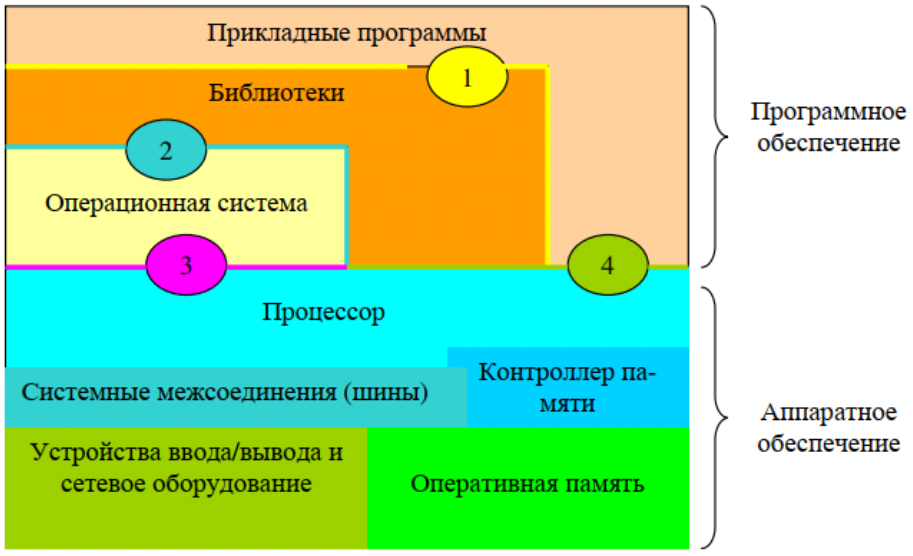
Система предназначена для определенных потребителей. Так, ОС системы Microsoft предназначены для широких масс, даже их серверные версии неудобны для использования на мейнфреймах. Для повышения прибыли производителям ОС нужно расширить сеть потребителей. Т.е. с одной стороны, системы выделена на один класс потребителей, но с другой стороны даже внутри одной системы выделяют версии (Windows Home Edition, Professional Edition), чтобы покрытие рынка было шире

3.5. Абстракция и виртуализация

- HW - Hardware, железо
  - Firmware - программы, обеспечивающие работу с железом (BIOS и т.п.)
    - Управление ОП
      - ...

Пользователю предоставляется некоторый абстрактный интерфейс, и пользователю часто не важна его реализация. **Абстракция** скрывает детали реализации нижних уровней, чем упрощает проектирование. **Виртуализация** - отображение интерфейса и ресурсов объекта на интерфейс и видимые ресурсы системы. В частности, этот принцип включает работу с едиными в физическом смысле объектами как с группой разных с точки зрения какого-то уровня абстракции. Пример - виртуальная машина.

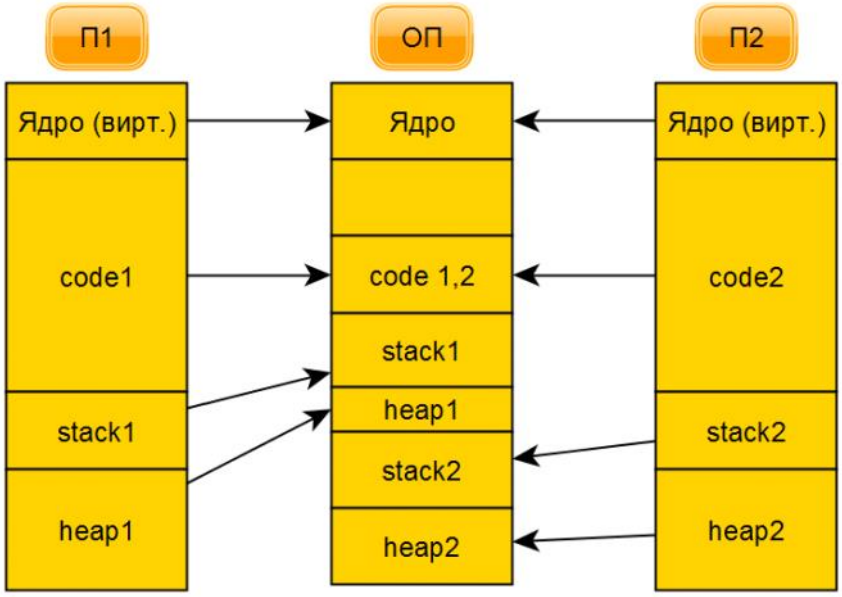
Архитектура компьютерной системы



- Интерфейсы 3,4 - интерфейсы между ПО и АО.
  - Интерфейс 3 - интерфейс процессора в режиме супервизора
  - Интерфейс 4 - множество команд процессора, доступного прикладным программам

- Интерфейс 2 - определяет доступ к функциям управляющей программы
- Интерфейс 1 - высокоуровневые библиотечные процедуры, используемые в прикладных программах.

Процессная виртуализация - UNIX



Каждый процесс работает с виртуальной памятью и считает, что ему выделена память с нулевого адреса. Программное обеспечение, реализующее процессную виртуальную машину, называют **рабочей средой.**

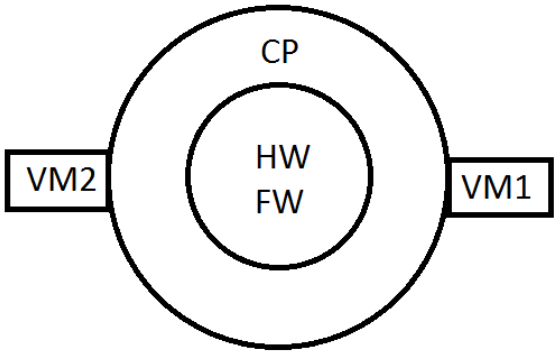
Реентерабельная функция - такая функция, которая может работать в интересах двух процессов. Так, обработку кода возможно сделать реентерабельной, но со стеком и кучей это невозможно.

Системная виртуализация (Z/OS)

Системная виртуальная машина предоставляет полнофункциональную среду для поддержки операционной системы. Эта виртуальная машина обеспечивает гостевую ОС всеми необходимыми интерфейсами и виртуальными устройствами - виртуальным процессором, памятью и т.п.

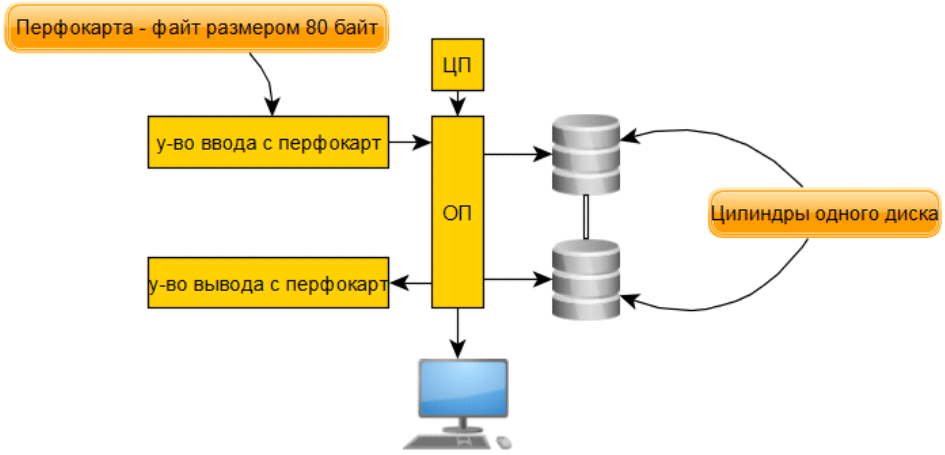
Хост - платформа, поддерживающая ВМ, а **гость** - процесс или систем, выполняемая на виртуальной машине.

Программное обеспечение виртуализации системной ВМ называют **Virtual Machine Monitor**. В таком случае монитор становится управляющей программой, которая управляет виртуальными машинами как ресурсами и задачами



Фирма IBM выпускала пакетные системы - PCP, MFT, MFT2, MVT, SVS, VS1, VS2, MVS. Пользователи не хотели брать новые системы, что не устраивало фирму. Чтобы избежать этого, IBM сделали систему виртуальных машин.

Устройство виртуальной машины:



4. Командный и программный интерфейс

27 февраля 2018 г. 12:28

Тема 4. Командный и программный интерфейс

4.1. Командный интерфейс

Командный интерфейс обеспечивает взаимодействие человека и компьютера. Для осуществления взаимодействия при загрузке системы стартуют задачи, выполняющие посылаемые компьютеру команды.

4.1.1. Пакетные системы

В z/OS MVS такие интерфейсы:

- JES (Job Entry System) - контролирует поток заданий
- TSO (Time Sharing System) - поддерживает диалоговый режим работы
- ISPF/PDF (Interactive System Productivity Facility) - позволяет редактировать тексты программ (в т.ч. на JCL), посылать их в JES и т.п

Пользователь пакетной системы описывает на JCL последовательность задач, которые должны быть выполнены в виде задания. Язык JCL (и пакетный режим в целом) не так удобен для взаимодействия с пользователем, но имеет преимущества, например, в случае необходимости многократной периодической обработки.
Пакетная система отличается использованием **принципа предварительного распределения ресурсов**

```
// имя JOB параметры
// step1 EXEC PGM = … PARM = '…'
// ddname1 DD UNIT=3990,VOL=SER=…,DISP=(OLD,PASS),
```

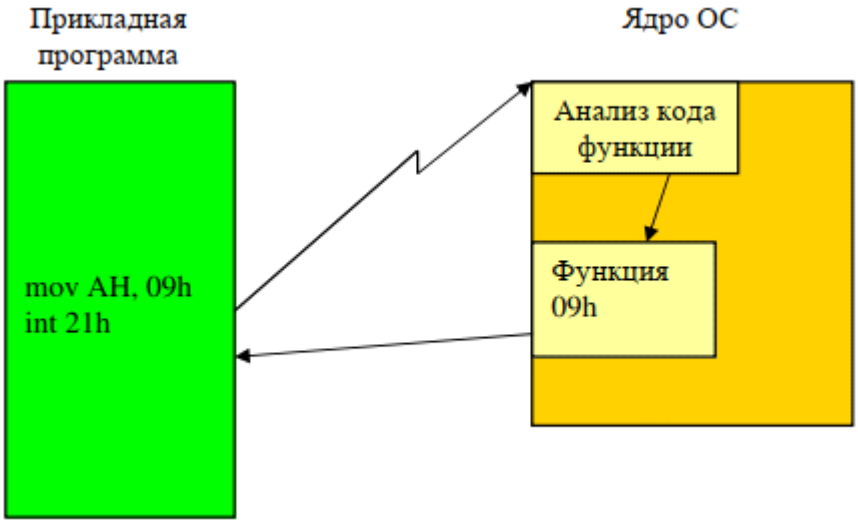
4.1.2. Диалоговые системы

В диалоговых ОС вместо JCL и JES используются **командный язык** и **командный процессор**. В отличие от пакетной системы, в диалоговой системе ресурсы выделяются во время выполнения задачи.
CLI - Command Line Interface.
.BAT-файлы - командные файлы. Состоит из директив и команд.
GUI - Graphic User Interface
Например, при клике мыши происходит прерывание, запускается обработчик, считывает координаты, проверяет, что находится по этим координатам, и либо передается управление командному процессору, либо (если по координатам данные) пытается найти программу, чтобы их открыть.
В Windows графика встроена в ядро, в UNIX же - отдельный процесс.

4.2. Программный интерфейс (API)

Служит для того, чтобы программист в своей программе мог воспользоваться функциями управляющей программы.
Для реализации API используются **программные прерывания** - те,

которые выполняются в темпе работы программы, в Intel-архитектуре - INT, в IBM-машинах - SVC (Supervision Call).
В команде INT 21H указывается номер вектора прерывания. Вектор прерывания устанавливается при загрузке ОС, он содержит адрес программы ядра - обработчик прерывания.
Если после выполнения стоит JC, то произошла ошибка, и в AX - return code.



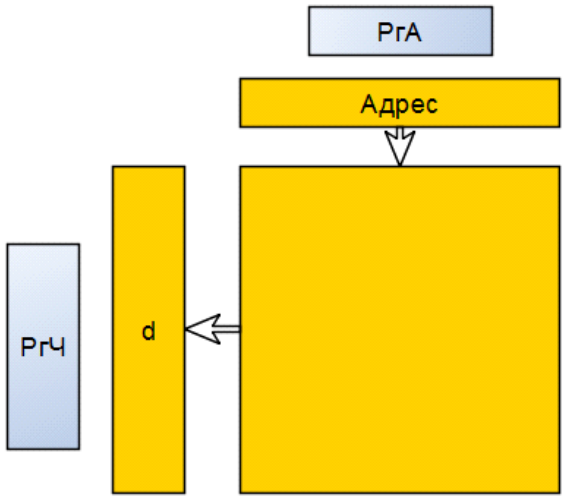
5. Организация и типы программных модулей

6 марта 2018 г. 11:50

Тема 5. Организация и типы программных модулей

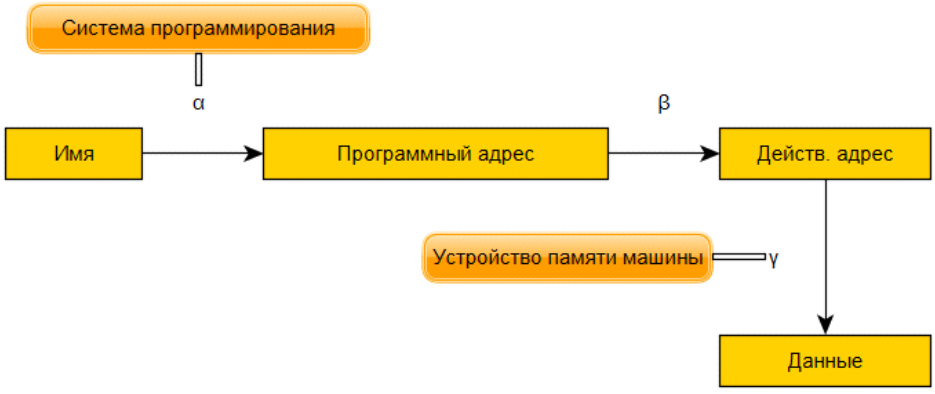
5.1. Преобразование адресов

Работа памяти:



РгА - регистр адреса

РгЧ - регистр числа



- **Имя** используется при написании программы
- Оно преобразуется (α) системой программирования в **программный адрес**
- Программный адрес преобразуется (β) в **действительный** системой адресации компьютера или загрузчиком.
- После этого с помощью операции разыменования (γ) по этому адресу берутся данные

Нас интересует операция β.

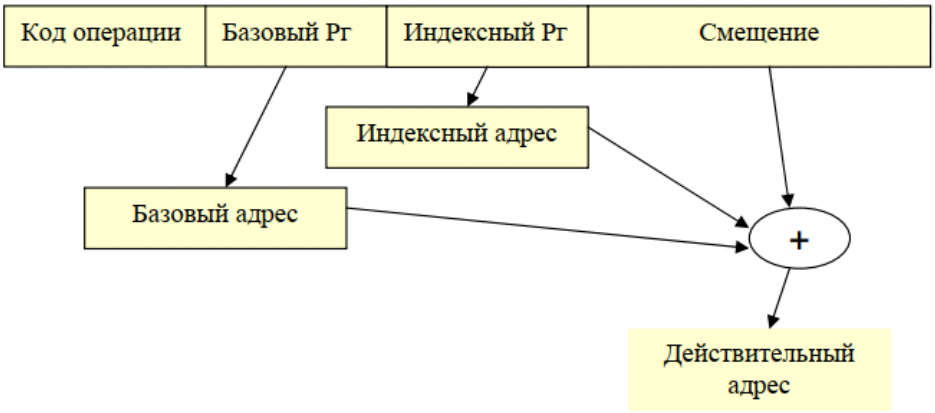
`Mov AX, seg DATA` - пример прямого адреса. Команда mov принимает только прямой адрес. Прямые адреса появляются только при загрузке модуля в память, и знает их загрузчик.

5.1.1. Статическая настройка - настройка программных адресов на то место в памяти, где находится модуль. Настройка происходит до выполнения модуля, но после загрузки в ОП.

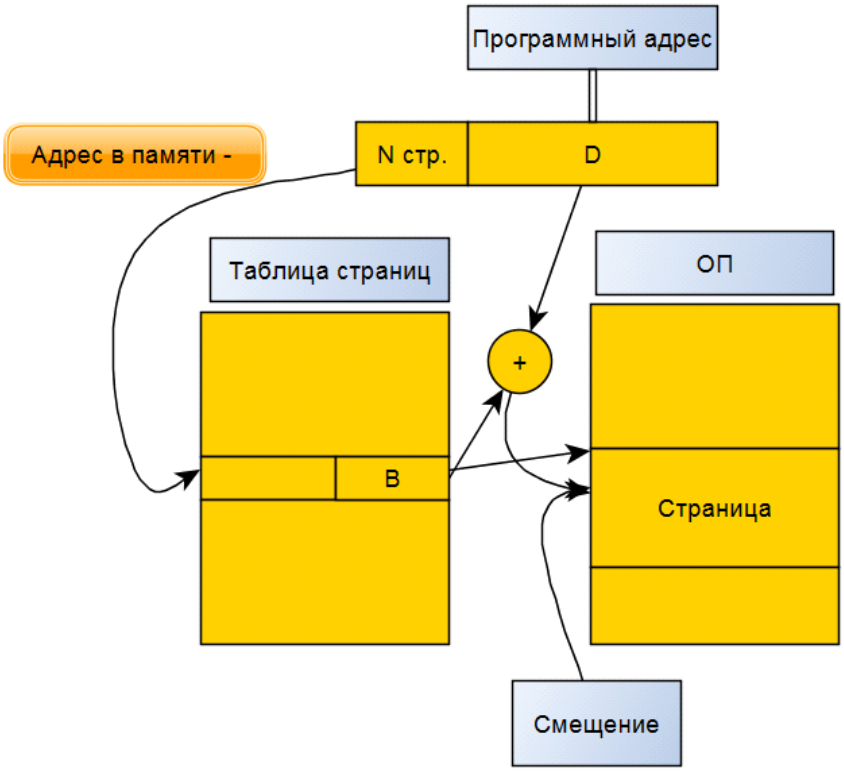
Загрузчик считывает файл загрузочного модуля, Relocation Table и начинает исследовать Relocation Table. По Relocation Table загрузчик определяет, куда подставлять адрес. Загрузчик отыскивает в RT соответствующие команды и подставляет нужные адреса.

5.1.2. Динамическая настройка - происходит в процессе выполнения самого модуля и выполняется системой адресации компьютера.

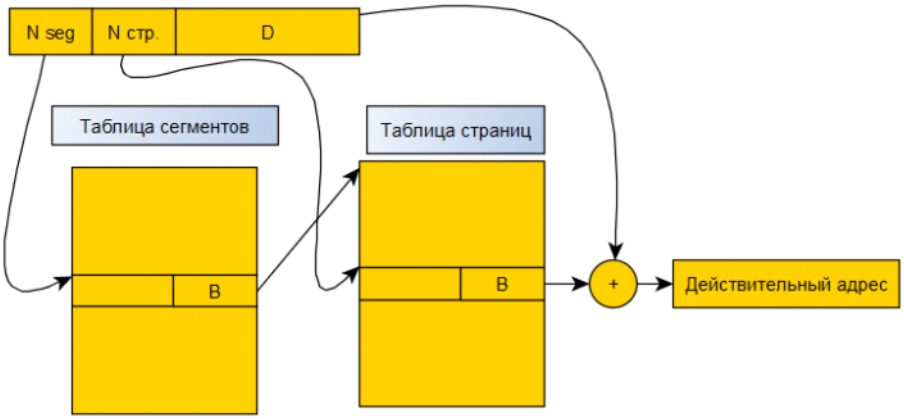
- **Базирование**
  В этом случае под адрес отводится два или более полей. Одно поле занимает **смещение** - адрес относительно начала модуля. Этот адрес определяется системой программирования и не требует корректировки.
  Другие поля занимают **базовый регистр** и, возможно, **индексный регистр**, значения которых определяются при загрузке модуля в ОП. Действительный адрес получается с помощью суммирования



- **Страничная память**
  Программный адрес разделяется на два поля **- номер страницы** и **смещение на странице**.



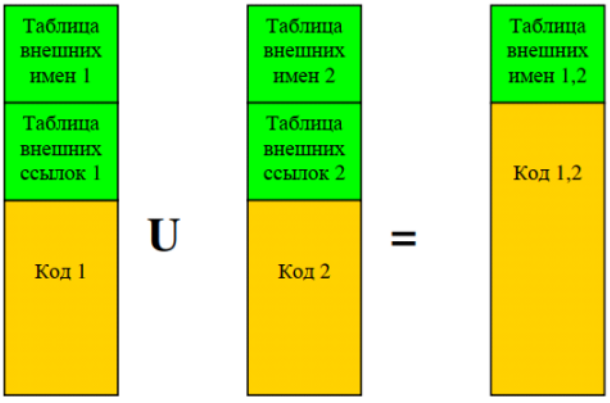
- **Сегментно-страничная память**



5.2. Типы загрузочных модулей

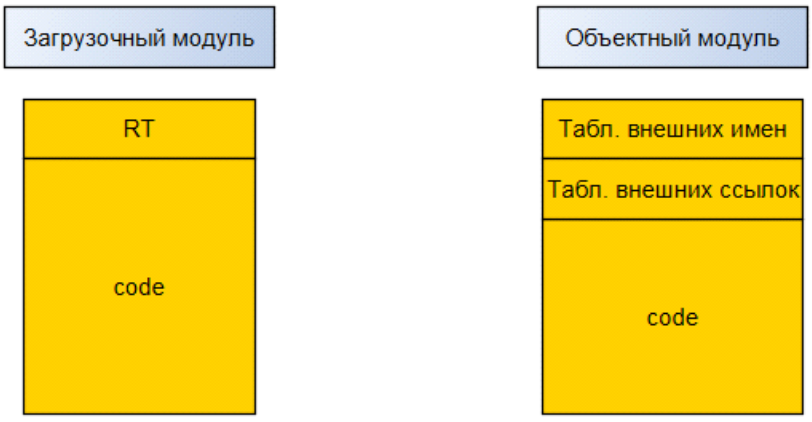
5.2.1. Модуль простой структуры

Модуль простой структуры состоит из совокупности объектных модулей, полученных в результате компиляции.



При создании модуля простой структуры линковщик работает следующим образом:

1. На входе имеется объектный модуль, его таблица внешних имён, таблица внешних ссылок и код.
2. Если есть ещё объектный модуль, то:
   a. Код второго добавляется к коду первого
   b. Таблицы внешних имен и ссылок объединяются
   c. Корректируются относительные адреса
3. Каждое имя (symbol) из таблицы внешних ссылок ищется в таблице внешних имён. Если имя найдено, то адреса обращения к этому имени в коде корректируются на указанные в таблице внешних имён, а из таблицы внешних ссылок имя удаляется.
4. Если есть ещё объектные модули, но назад на шаг 1
5. Если таблица внешних ссылок не пуста, то какая-то зависимость не была разрешена. В этом случае выводится ошибка вроде "unresolved external symbol"

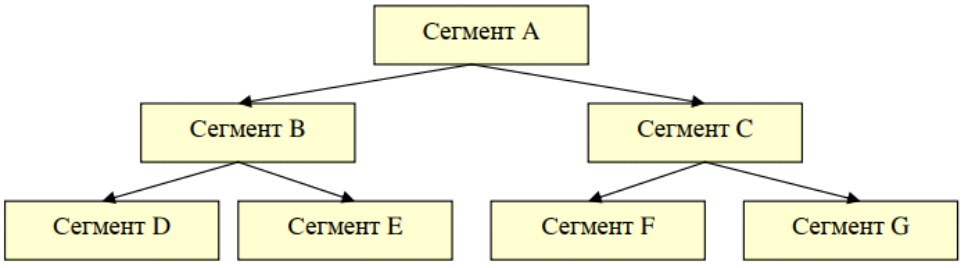


После успешного разрешения внешних ссылок получается загрузочный модуль. В начале загрузочного модуля находится таблица настройки - Relocation Table, необходимая для настройки относительных адресов при загрузке модуля.

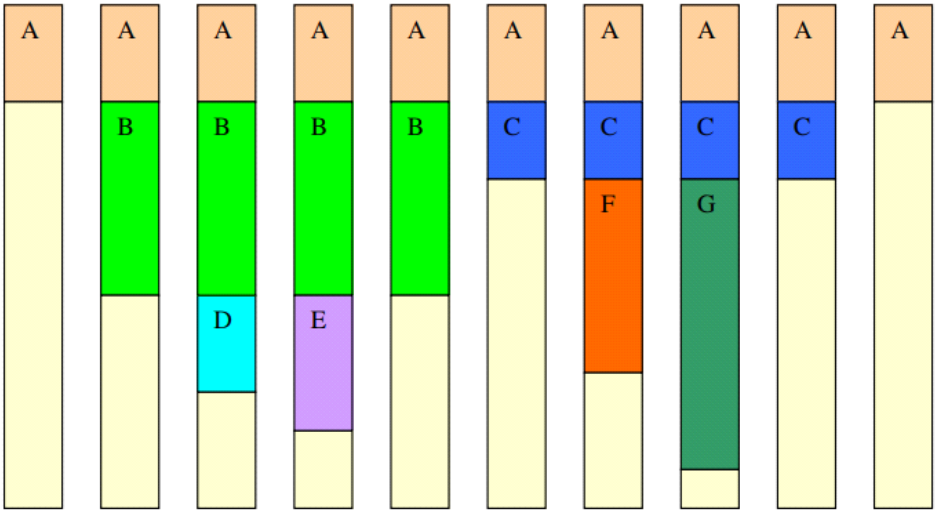
5.2.2. Модуль оверлейной структуры

Сейчас в системах не используются. Они использовались, когда из-за большого размера модуля он не влезал в память.
Для решения этой проблемы выделяется **корневой сегмент**, который всегда сидит в памяти. Остальные модули сидят на диске и подгружаются только тогда, когда это необходимо. Адресное пространство отработавших модулей перекрываются (overlay) новыми.

Пример иерархии:



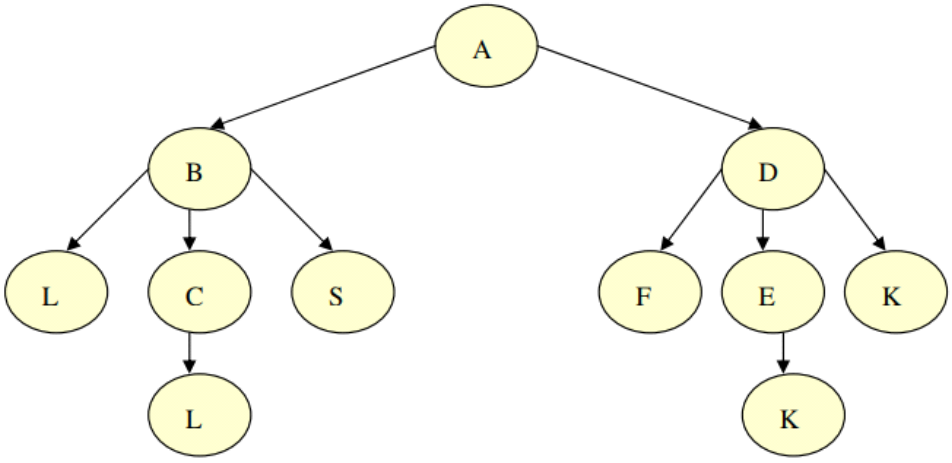
Пример изменения карты памяти при работе такого модуля:



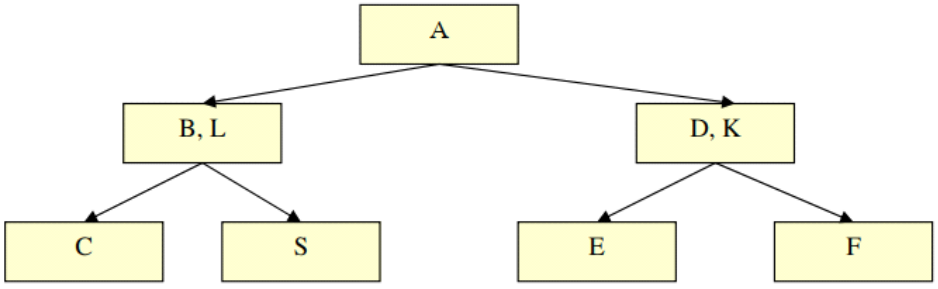
Как и все загрузочные модули, модули оверлейной структуры тоже строятся в форме объектных модулей. Программирование оверлейных

модулей на языках высокого уровня (например, Си) осложнено тем, что один файл с программным кодом независимо от размера транслируется в один объектный модуль (в отличие, например, от FORTRAN).

Планирование структуры такого модуля зависит от связей (связь - обращение из одной секции в другую) между секциями программы. Их можно представить в виде графа:



Эти секции объединяются в сегменты, причём каждая секция может находится только в одном сегменте. Ещё одно условие - секции, к которым поступает обращение из сегмента, должны находится либо в памяти, либо в сегментах, зависимых от данного.
Пример дерева сегментов для вышеприведённого графа:



Проблема в том, что построение подобного "хорошего" дерева не всегда возможно - иногда может получится, что все сегменты объединятся в один корневой и модуль выродится в модуль простой структуры.

5.2.3. Модуль динамической структуры

Используется до сих пор - в z/OS MVS и в Windows (DLL), в UNIX нет.
Структура модуля заранее неизвестна и зависит от данных, которые он будет обрабатывать.
В машинах IBM есть макрокоманды для управления динамической структурой: LOAD, DELETE, LINK, XCTL.

- LOAD загружает модуль и возвращает адрес точки входа. Если загрузка происходит несколько раз, то просто увеличивается специальный счётчик.
- DELETE - уменьшает счётчик загрузок на 1, и если он стал 0, то освобождает память.

- LINK - загружает модуль в ОП и сразу передаёт управление. Поле завершения работы модуль может быть выгружен
- XCTL - передаёт управление модуль без возврата. В этой команде отпала необходимость с появлением технологий виртуализации памяти.

Пример:

Модуль A:

```
… ;(1)
LOAD EP = B ;(2) EP - Entry Point
CALL B
…
LINK C
… ;(6)
DELETE EP=B ;(7)
```

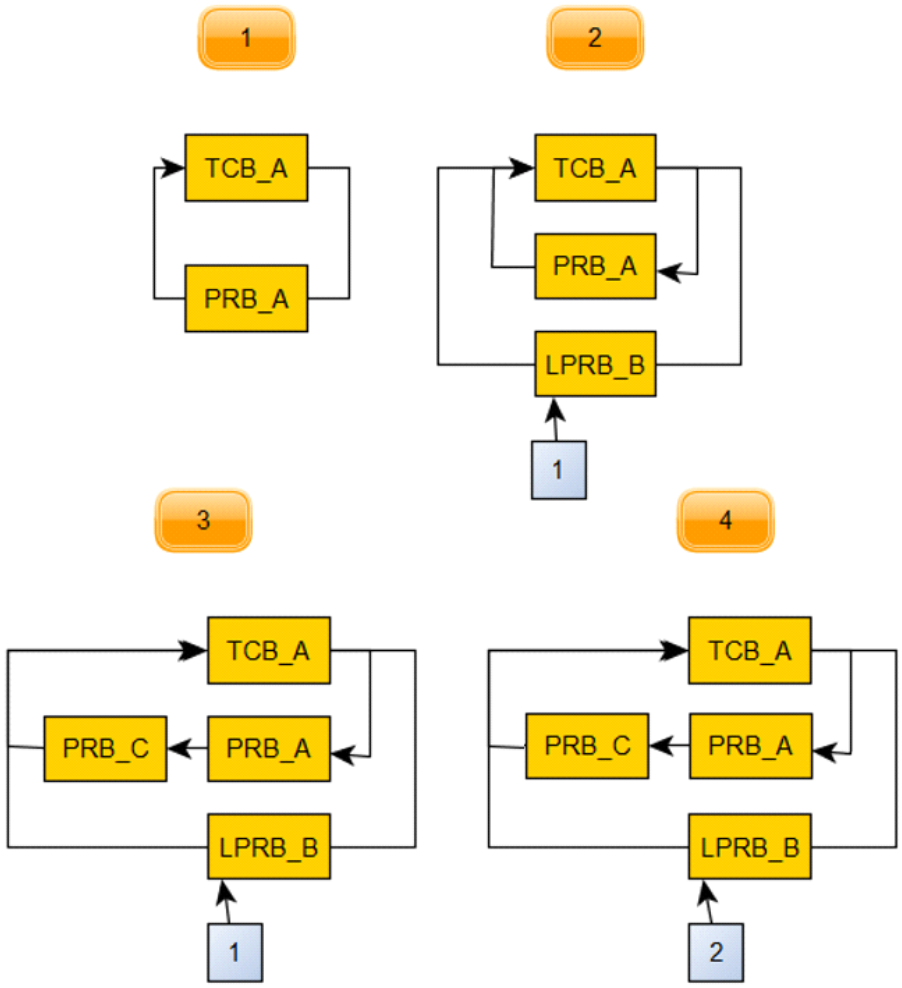
Модуль C:

```
… ;(3)
LOAD EP = B ;(4)
CALL B
DELETE EP = B ;(5)
```

Task Control Block (TCB) строится в области системных очередей, когда нужно начать выполнение задачи. К TCB присоединяется PRB.

Program Request Block (PRB) - управляющий блок, описывающий модуль. Из PRB в ОП строится список.

Load PRB (LPRB) - учитывает загруженные модули и строится при выполнении команды LOAD. Счётчик в нём при первой загрузке ставится на 1.



1. В ОП загружен TCB_A, к нему присоединён PRB_A
2. В ОП загружен модуль B, создан LPRB_B, его счётчик установлен на 1.
3. Загружен модуль C, ему передано управление.
4. Из модуля C загружен модуль B. Второй раз модуль B не загрузится, но счётчик в LPRB_B увеличится на 1. При вызове модуля B управление будет передано LPRB_B
5. Счётчик LPRB_B уменьшен на 1
6. Модуль C отработал и PRB_C удалён, структура принимает вид п.2
7. Выгружен модуль B. Структура принимает вид п.1

5.3. Организация программных модулей

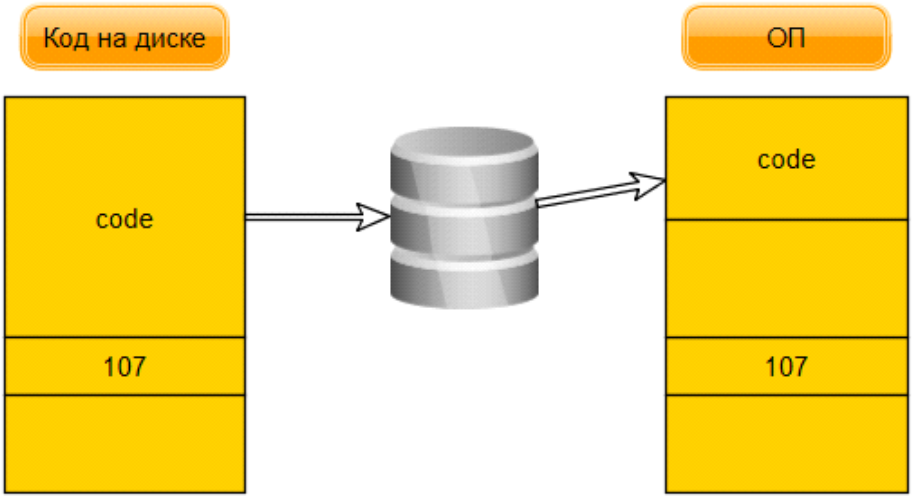
5.3.1. Однократно используемые модули

Появились из-за экономии памяти.

Пример - простой датчик псевдослучайных чисел

```
int rand(){
     static int n = 107;
     n = 25173*n + 13849
     n = n%65536;
     return n;
}
```

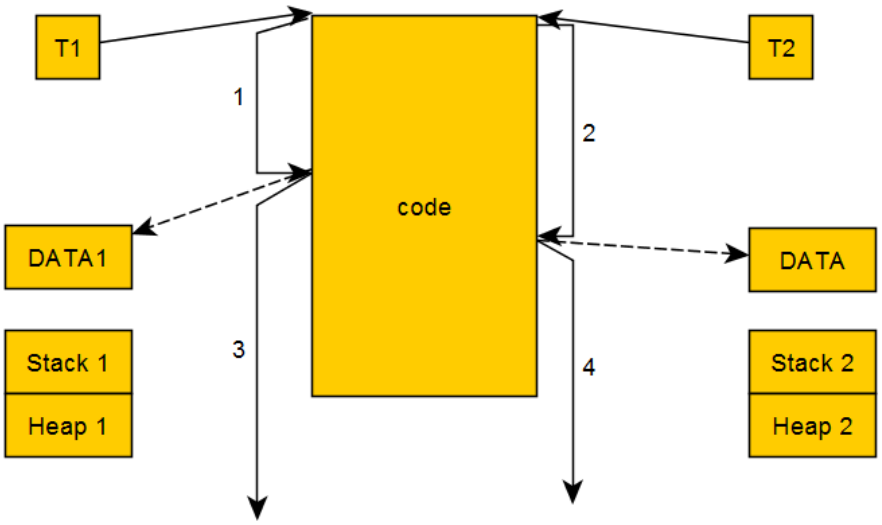
`Static` - не автоматическая переменная, инициализирующаяся только один раз при входе в функцию, а далее сохраняющая предыдущее значение



5.3.2. Повторно используемый модуль
Отличается тем, что инициализация переменных осуществляется во время выполнения. Таким образом, при каждом новом обращении с одинаковыми параметрами результат не изменится.
Пример: `for (i=1; i<n;i++){}`

5.3.3. Повторно входимый модуль (реентерабельный, реентрантный, чистая процедура). Особенность в том, что код такого модуля может выполнятся двумя и более процессами. Сейчас используются при написании многопотоковых приложений, функций ядра и т.п.
Пусть T1 и T2 - задачи или процессы, использующие один код. Пусть T1 начал выполнение кода, но прервался - закончился интервал времени. Аналогично T2. Опять инициализировался T1. Он должен начать с того же места, с которого остановился. И T2 тоже.

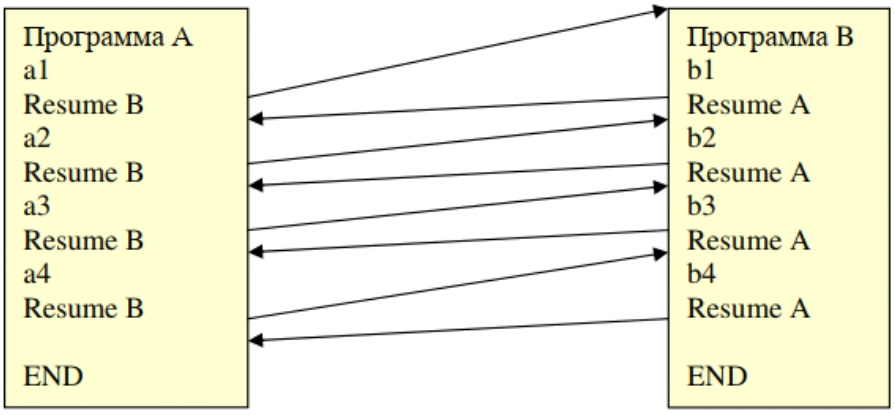
- Во-первых, получается, что у каждого процесса должны быть свои сегменты данных, стека и кучи. Сегменты определяются регистрами. Значит, нужно так написать код, чтобы в зависимости от задачи менялись регистры.
- Во-вторых, в коде не должно быть команд, которые модифицируют другие команды, не должно быть локальных переменных, изменяющихся в процессе выполнения. Из такого модуля нельзя вызывать нереентерабельные функции.



В случае обращения к файлам нужно запоминать DataControlBlock'и, хранящие положения в файлах и так далее. Способы написания реентерабельных модулей зависят от системы программирования.

5.3.4. Сопрограммы

Есть однозадачная маленькая система, которой нужно обеспечить мультипрограммный режим. Для решения такой задачи происходит моделирование логического параллелизма.



Для такой реализации нужны многовходовые модули (так, в C++ для этого есть оператор `entry`). Также нужно изготовить функцию, которая осуществляет переход на заданную точку входа и сохраняет адрес возврата.

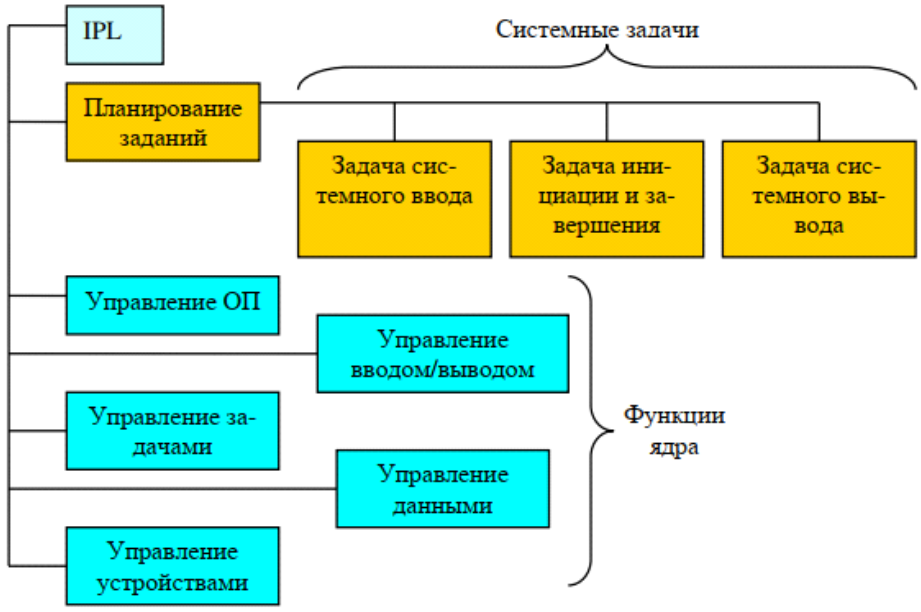
В современных системах, как правило, не используется.

Часть II. Организация и функционирование управляющей программы

13 марта 2018 г. 12:22

Управляющая программа представляет собой набор функций, которые резидентно размещаются в ОП и образуют **ядро (kernel)**. Инициализацию ядра проводит **загрузчик**. Ядро всегда работает в режиме супервизора, т.е. ему доступны все функции процессора.
Управляющая программа обеспечивает следующие основные функции:

- Управление ОП
- Управление процессами
- Управление устройствами
- Управление вводом/выводом
- Управление данными



Планирование заданий имеется только в пакетных системах. В диалоговых системах достаточно командного процессора.

Существуют два типа ядер:

- **Монолитные ядра (Monolithic Kernel)** - в таком ядре реализуются все основные функции ОС, и оно является единой программой.
  Во многих монолитных системах компиляция ядра осуществляется отдельно для каждого компьютера. Единственный способ в таком случае добавить (или убрать) новые компоненты в ядро - перекомпиляция.
  Пример - MS-DOS
- **Микроядра (Microkernel)** - в ядре остается лишь минимум функций, который обязательно должен быть реализован в режиме супервизора. Остальные, более высокоуровневые функции ядра реализуются в прикладном режиме.

Пример - Symbian

- **Гибридные (Hybrid)** - занимают промежуточное положение.
  Пример - BeOS

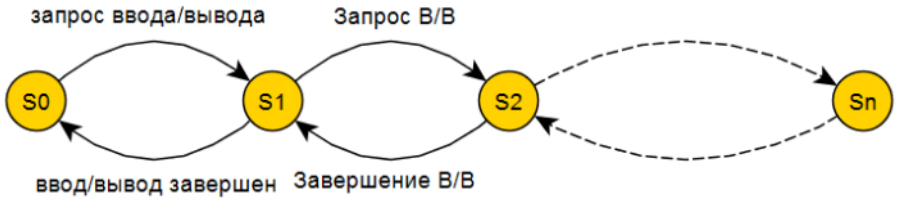
6. Понятие мультипрограммирования

13 марта 2018 г. 12:23

Тема 6. Понятие мультипрограммирования

Мультипрограммирование - такой режим работы системы, когда два или более процесса разделяют ресурсы системы.
N - **уровень мультипрограммирования** - чисто задач или процессов, которые выполняются в мультипрограммном режиме.
Основная задача - загрузить устройства системы как можно больше. Чтобы оценить эффективность, нужно узнать время простоя процессора.

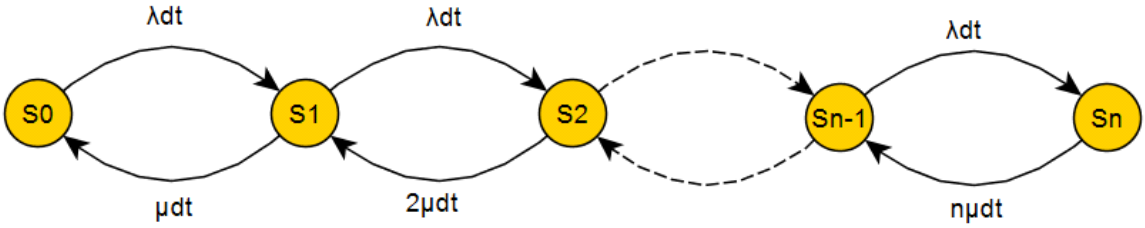
Построим простую модель.
Задачи в состоянии ожидания - **блокированные**. Готовые к работе задачи - **неблокированные**
S_0 — состояние системы, в котором все задачи готовы. Все задачи неблокированные
S_1 — одна задача блокирована
…
S_n — все n задач находятся в состоянии ожидания. ЦП простаивает.
Система переходит из состояния S_i в S_{i+1}, если какой-то процесс запросил ввод/вывод и тем перешёл в состояние ожидания. Обратный переход - завершение ввода/вывода и разблокировка процесса.



Перейдем к вероятностной системе:
P_i — вероятность нахождения системы в состоянии S_i.
Нас интересует вероятность P_n — вероятность, характеризующая простой процессора.
Для определения обычно используют закон Пуассона, так как он дает хорошие аналитические выражения.

$\boxed{\lambda dt}$ — вероятность блокирования процесса в результате запроса на ввод/вывод к моменту следующего наблюдения.
$\boxed{\frac{1}{\lambda}}$ — среднее время между запросами на ввод/вывод (мат. ожидание)
$\boxed{\mu dt}$ — вероятность завершения ввода/вывода к следующему моменту.
$\boxed{\frac{1}{\mu}}$ — мат. ожидание завершения ввода/вывода
События несовместны - перейти может в один момент только одна задача. Поэтому вероятности завершения ввода/вывода складываются и получается, что вероятность перехода из состояния S_{i+1} в S_i равна $(i+1)\mu dt$. Вероятность перехода в следующее

состояние, напротив, постоянна.



$\boxed{\omega = \frac{\lambda dt}{\mu dt + \lambda dt} = \frac{\lambda}{\lambda + \mu}}$ - характеризует, насколько часто задача задача занимается вводом/выводом (большая ω — счётная задача, маленькая - диалоговая).

Если λ растет, а μ —нет, это показывает частые запросы на ввод вывод. В этом случае машина уйдет в S_n. Наоборот, если μ растет относительно λ — устройства работают очень быстро, машина уйдет в состояние S_0.
Чтобы что-то понять, нужно рассмотреть равновесное состояние.

Обозначим $\boxed{p_{i,j}}$ — вероятность перехода и i в j. Тогда уравнение равновесия будет выглядеть следующим образом:
$p_{i,i+1} \cdot P_i = p_{i+1,i} \cdot P_{i+1}$
Подставив нужные величины, получим систему уравнений:

$$\begin{cases} \lambda dt P_0 = \mu dt P_1 \\ \lambda dt P_1 = 2\mu dt P_0 \\ \vdots \\ \lambda dt P_i = (i+1) \cdot \mu dt P_{i+1} \end{cases}$$

Из этой системы получим выражение для P_i через P_0

$$\begin{cases} P_1 = \frac{\lambda}{\mu} P_o \\ P_2 = \frac{\lambda}{2\mu} P_1 = \frac{1}{2}\left(\frac{\lambda}{\mu}\right)^2 P_0 \\ \vdots \\ P_i = \left(\frac{1}{i!}\right)\left(\frac{\lambda}{\mu}\right)^i P_0 \\ \vdots \\ P_n = \left(\frac{1}{n!}\right)\left(\frac{\lambda}{\mu}\right)^n P_0 \end{cases}$$

По свойству вероятности $\sum_{i=0}^{n} P_i = 1$.

$$P_0 = 1 - \sum_{i=1}^{n} P_i = 1 - \sum_{i=1}^{n} \left[\left(\frac{1}{i!}\right)\left(\frac{\lambda}{\mu}\right)^i P_0\right] \Rightarrow P_0 = \frac{1}{1 + \sum_{i=1}^{n}\left[\left(\frac{\lambda}{\mu}\right)^n \left(\frac{1}{i!}\right)\right]} = \frac{1}{\sum_{i=0}^{n}\left[\left(\frac{\lambda}{\mu}\right)^n \left(\frac{1}{i!}\right)\right]}$$

Подставляя в выражение для P_n полученное для P_0, получаем:

$$P_n = \frac{\left(\frac{1}{n!}\right)\left(\frac{\lambda}{\mu}\right)^n}{\sum_{i=0}^{n}\left[\left(\frac{\lambda}{\mu}\right)^n\left(\frac{1}{i!}\right)\right]}$$

После выражения $\frac{\lambda}{\mu} = \frac{\omega}{1-\omega}$ получится итоговое выражение

$$\boxed{P_n = \frac{\left(\frac{1}{1-\omega}\right)^n}{n! \cdot \sum_{i=0}^{n}\left[\left(\frac{\omega}{1-\omega}\right)^i \cdot \left(\frac{1}{i!}\right)\right]}}$$

Значения функции:

n\ω	25%	50%	75%
2	4,0%	20,0%	52,9%
3	0,4%	6,3%	34,6%
4	0,0%	1,5%	20,6%
5	0,0%	0,3%	11,0%

Поэтому в диалоговых системах практически не обращают внимание на загрузку ЦП. Более сложные модели лучшего результата не дают.

7. Управление основной памятью

20 марта 2018 г. 11:46

Тема 7. Управление основной памятью

Самостоятельно из ОС. Ч2

Функции управления памятью реализуются в ядре ОС. Реализация управления ОП специфична и зависит от возможностей аппаратуры и выбранной **стратегии распределения памяти**. Система управления ОП выполняет следующие функции:

- **Учёт ОП в ОС** - отслеживания занятых и свободных областей ОП
- Планирование запросов
- Выделение и освобождение ОП

7.1. Связывание загрузочного модуля с адресами в ОП

Логическое адресное пространство - совокупность программных адресов, которую занимает программа.
Физическое адресное пространство - совокупность абсолютных адресов физической памяти, в которых располагается программа
Система управление ОП отвечает за **отображение логического пространства в физическое**.

Статическое связывание - назначение корректных физических адресов в командах с прямой адресацией. Например, в коде вроде

```
mov AX, seg DATA
mov DS, AX
```

второй операнд первой команды неизвестен при компиляции и на его месте будут располагаться нули. При помещении загрузочного модуля в ОП загрузчик скорректирует этот адрес по таблице настроки.

Динамическое связывание - вычисление действительных адресов в процессе выполнения средствами процессора.
Логической пространство может быть **односегментным** и **многосегментным**, а физическое - **одноуровневым** и **многоуровневым**. Уровни памяти различаются по скорости доступа и объему. При многоуровневой организации используются системы виртуальной памяти, иногда называемые **страничными системами**.

7.2. Управление ОП в нестраничных системах

Используется, если физическая память одноуровневая. В таких системах всё логическое пространство загружается в физическую память. Это менее рационально, но просто для реализации.

7.2.1. Одиночное непрерывное распределение

После загрузки ОС вся оставшаяся память используется для задач. Управлением памятью занимаются программы.

Функция учёта заключается в построении списка **блоков MCB (Memory Control Block)**. Адрес первого элемента этого списка находится в области управляющей программы. В каждом элементе списка содержится информация о размере блока, занятости, владельце.
Такая стратегия реализуется в однопрограммных ОС. Поскольку в таких системах пользователь только один, защита памяти обычно не реализуется и управляющую программу можно легко "сломать".

Достоинства:

- Простота реализации

Недостатки

- Невозможность мультипрограммирования
- Нерациональное использование памяти
- Отсутствие защиты осложняет реализацию сложных проектов и отладку.

7.2.2. Распределение разделами

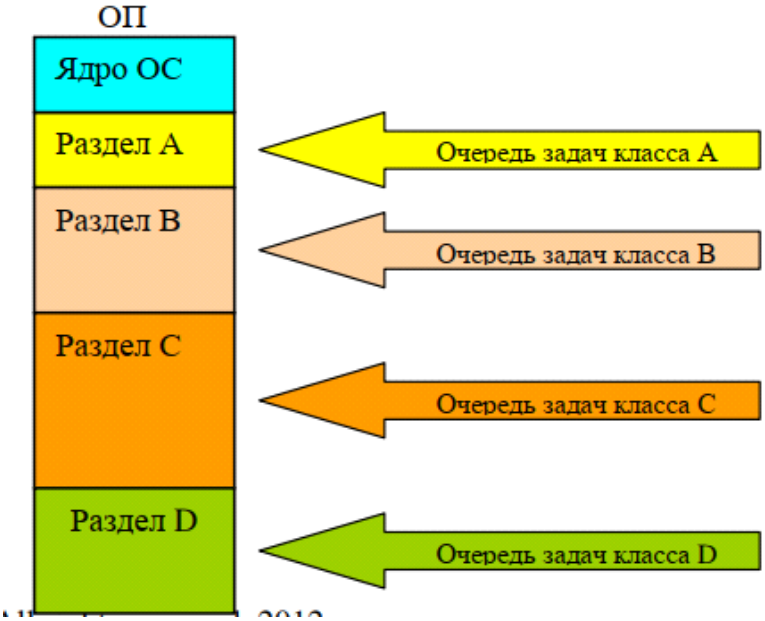
Для реализации мультипрограммирования задаче выделяется участок памяти - **раздел**. Доступ задачи к другим разделам должен быть исключен, для чего существуют два механизма:

- **Защита регистрами** - реализация в процессоре пары из **базового регистра**, в который загружается базовый адрес раздела и **регистра защиты**, в который загружается размер раздела. При вычислении адреса производится проверка на принадлежность к нужному адресному пространству
- **Защита ключами** - ОП выделяется небольшими блоками, с каждым из которых связан регистр защиты. При создании задачи в **Program State Word** записывается **код защиты**, с которым сравнивались регистры блоков при вычислении адреса

7.2.2.1. Распределение статическими разделами

После загрузки ОС вся свободная память делится на разделы

фиксированной длины, в которые загружаются задачи для выполнения. Такая стратегия использовалась в первых мультипрограммных ОС и в тех случаях, когда характеристики всех задач были заранее известны. Рассмотрим ОС IBM OS360 MFT. В ней ОП делилась на разделы - по одному разделу на каждый класс задач. Задачи всех заданий, принадлежащих данному классу, выполнялись в нужном разделе. Программист определял класс задачи параметром CLASS в операторе JOB. Класс определялся в зависимости от характеристик задачи.



Задача системного ввода помещала задание в очередь нужного класса. **Учёт состояния памяти** производился с помощью битового вектора - по одному биту на состояние каждого раздела, а планирование происходило с использованием очереди задач для каждого класса.

Преимущества

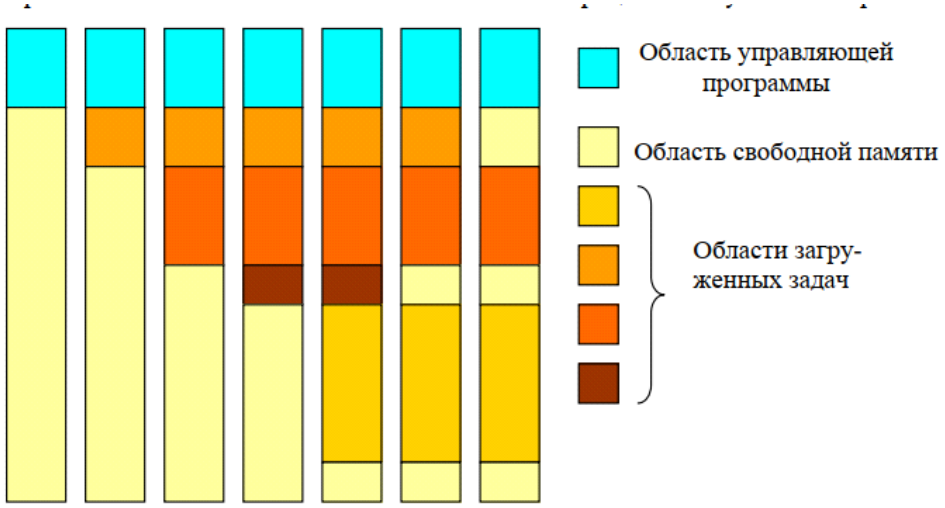
- Мультипрограммирование
- Простота реализации

Недостатки

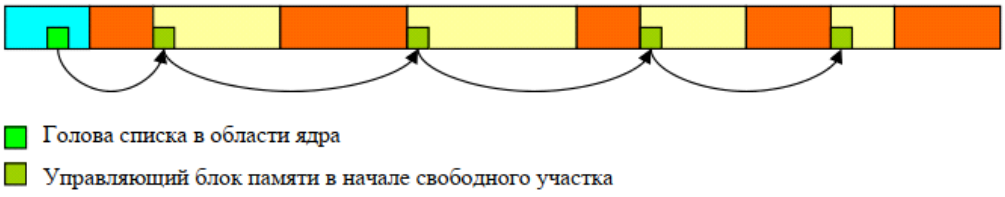
- Нерациональное использование памяти при различных неизвестных заранее задачах

7.2.2.2. Распределение динамическими разделами

При такой стратегии память выделяется по запросам, и в каждом таком запросе содержится размер требуемого участка памяти. При создании очередной задачи программа управления задачами создает запрос на выделение памяти, а система управления памятью находит свободный участок и создает раздел задачи, который уничтожается при завершении задачи.



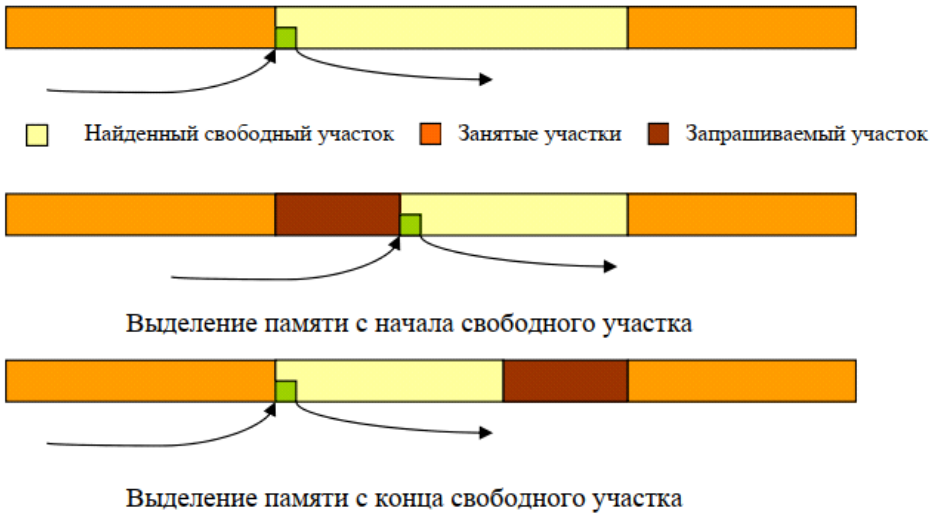
В таком случае для **учёта состояния памяти** также может использоваться **список МСВ**. МСВ помещается в начало каждого свободного участка, содержит адрес следующего МСВ и объем текущего участка. Информация о занятых участках хранится в блоке управления задачей. При завершении задачи участок освобождается, и список реорганизуется.



Учитываются именно свободные участки, так как в таком случае быстрее найти именно свободный участок для задачи.

Алгоритмы выделения памяти

Нужно выделить участок памяти, размер которого больше или равен размеру запроса.



Выделение памяти с начала свободного участка

Выделение памяти с конца свободного участка

Выделение с конца проще, так как в этом случае придется только изменить размер свободного участка.

Алгоритмы поиска свободного участка

1. **First fit**

   Алгоритм просматривает список и выбирает первый подходящий по

размеру. В начале списка группируются маленькие по размеру участки, а большие - в конце. Это уменьшает фрагментацию, но увеличивает время поиска

2. **The Best Fit**
   Алгоритм просматривает список и выбирает наиболее подходящий участок - такой, чтобы получившийся свободный был минимального размера.
   Если сортировать участки по размеру, то алгоритм First Fit будет работать как Best Fit. Но этот подход ещё больше увеличит время поиска.
   Такой способ оставляет нетронутыми большие участки и дает каждой задаче лучший участок, однако этим образует множество маленьких участков, что усиливает фрагментацию.
3. **The Worst Fit**
   Для каждой задачи выбирается максимальный по размеру участок. Также получается из First Fit сортировкой по размеру. В таком случае остаток после выделения памяти каждой задачи остается самым большим.

Алгоритмы освобождения памяти

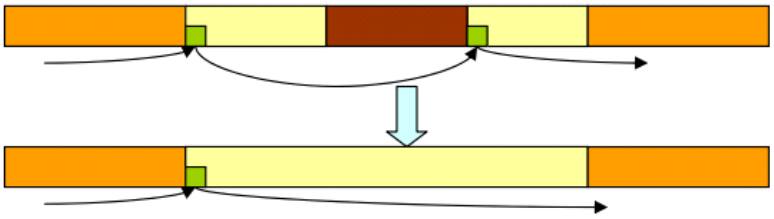


Рис. 23 Освобождение участка находящегося между двумя свободными

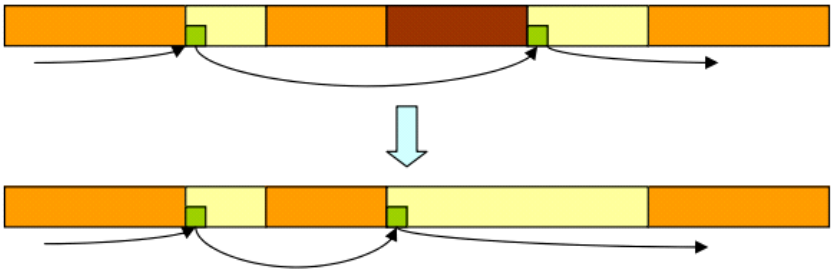


Рис. 24 Освобождение участка находящегося между свободным и занятым

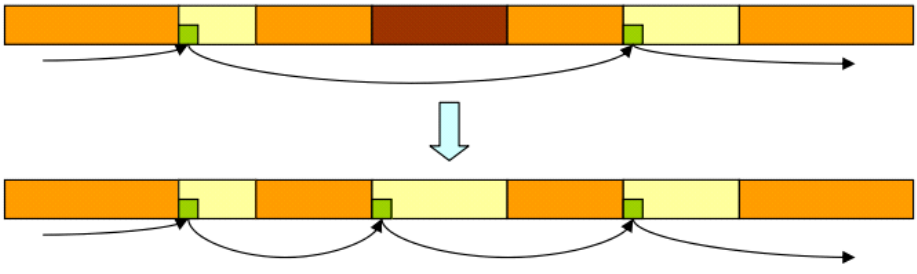


Рис. 25 Освобождение участка находящегося между двумя занятыми

Достоинства динамического выделения памяти:

- Мультипрограммирование
- Более рациональное использование памяти по сравнению со статическим
- Простота алгоритмов
- Низкие аппаратные затраты

Недостатки

- Фрагментация памяти
- Не может дать задаче больше памяти, чем есть в системе
- Загрузка ненужных частей кода в память.

7.2.3. Проблема фрагментации

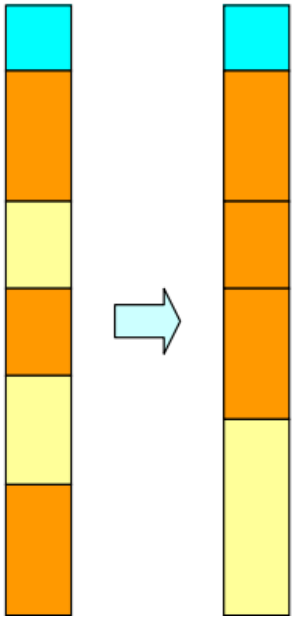
Фрагментация - такое состояние, при котором образуется много маленьких неиспользуемых свободных участков и не находится достаточно большого непрерывного участка для удовлетворения запроса задачи.

- **Внешняя фрагментация** - в ОП между задачами
- **Внутренняя фрагментация** - внутри области задачи

Для борьбы с внутренней фрагментацией используются библиотечные функции, а для борьбы с внешней - специальные методы управления ОП.

7.2.3.1. Распределение перемещаемыми разделами памяти

В этой стратегии решения все занятые разделы смещаются в одну область, чем образуют один непрерывный раздел свободной памяти.



Проблема в том, что при перемещении в программах необходимо корректировать множество адресно-зависимых элементов, например:

- Базовые регистры
- Команды с прямыми адресами
- Адреса в списках параметров
- Указатели и ссылки

Это очень трудоемкая операция. Для решения этой проблемы в программах стали делать относительными все адреса, кроме базовых

регистров.

Достоинства:

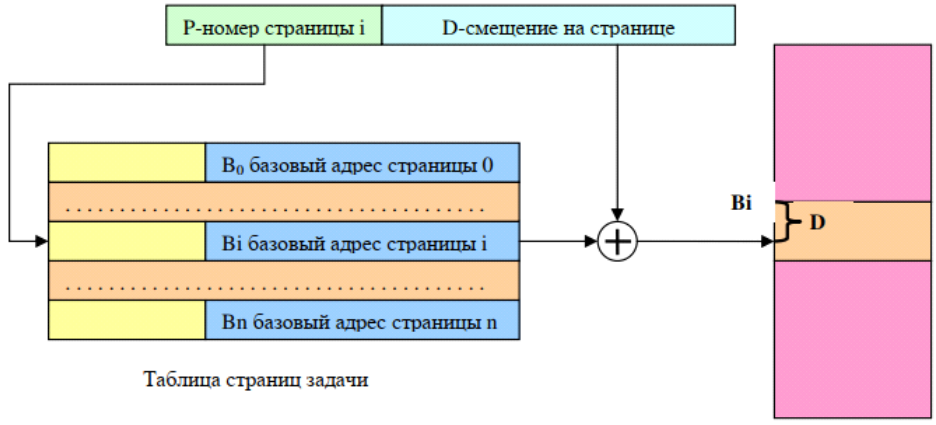
- Ликвидация фрагментации
- Увеличение уровня мультипрограммирования
- Увеличение производительности системы

Недостатки:

- Трудоёмкая перекомпоновка памяти
- Всё еще не может дать больше памяти, чем есть физически
- В память загружается всё адресное пространство задачи, что нерационально

7.2.3.2. Страничная организация памяти

Такая стратегия организации памяти отказывается от единства физического и логического пространства. Логическая и физическая память делится на **логические** и **физические страницы** равного размера. Адрес представляется в виде двух полей: поля номера страницы.
Для каждой задачи в ОП создается **таблица страниц задачи**. Элемент таблицы содержит базовый адрес физической страницы, в котором размещена соответствующая логическая. Для получения **действительного адреса** процессор по номеру страницы в поле адреса выбирает элемент таблицы и суммирует базовый адрес физической таблицы со смещением.



Для **учёта состояния памяти** - свободных физических страниц также создается битовый вектор.
Алгоритм **выделения памяти** работает следующим образом:

- Если число свободных физических страниц меньше числа логических страниц, то задача переводится в состояние ожидания
- Если свободных страниц физических страниц достаточно, что логические страницы размещаются в ОП и формируется таблица страниц задачи.

Достоинства:

- Отсутствие внешней фрагментации
- Отсутствие накладных расходов, связанных с перекомпоновкой

Недостатки:

- Внутренняя фрагментация
- Наличие нераспределённых физических страниц
- В память загружается всё адресное пространство задачи, что нерационально

7.3. Виртуальная память

У всех перечисленных стратегий работы с памятью есть недостаток - в память загружаются редко используемые участки кода. Чтобы увеличить эффективность использования памяти, нужно загружать только ту часть кода, которая необходима в данный момент.
Для этого в ОП загружается только часть адресного пространства, а другая часть остаётся во внешней памяти.

7.3.1. Управление виртуальной памятью по запросам

Память всё также делится на физические и логические страницы одинакового размера. При выполнении программы только часть логических страниц находится в ОП. При вычислении адреса страницы, находящейся во внешней памяти, возникает **страничное прерывание**, по которому ядро находит нужную страницу и загружает её в ОП.
Если в ОП нет места, то какая-то страница замещается по **алгоритму замещения**.
В различных системах реализация **области своппинга** различна - так, в Linux и UNIX на диске выделяется специальный раздел, а в Windows создается pagefile.sys

Для **учета состояния памяти** в такой стратегии необходимы следующие структуры:

- **Таблица физических страниц** - таблица, в которой отмечаются **физические состояния** страницы. Страница может быть **свободна**, **занята** или находится в **транзитном состоянии** (транзитное состояние введено для предотвращения конфликтов).
  Также в этой таблице находится **бит обращения**, который устанавливается, если к странице обращались.
- **Таблица страниц задачи** - создается для каждой задачи. Содержит:
  - **Бит состояния**, находится ли страница в ОП или во внешней памяти.
  - **Базовый адрес физической страницы**
  - **Бит использования страницы** - устанавливается, если была проведена запись.
- **Таблица внешних страниц задачи** - также создается для каждой задачи и содержит адреса страниц во внешней памяти.

Выделение памяти происходит либо при загрузке задачи в ОП, либо при страничном прерывании. При отсутствии свободных страниц кандидат на выгрузку определяется **алгоритмом замещения**

При **освобождении памяти** удаляются как страницы в ОП, так и внешние страницы.

Достоинства:

- Большая виртуальная память
- Более эффективное использование ОП
- Неограниченное мультипрограммирования

Недостатки:

- Рост накладных расходов
- Пробуксовка

7.3.1.1 Алгоритмы замещения страниц

От эффективности такого алгоритма зависит производительность системы. Эффективность определяется числом страничных прерываний. Моделирование производительности включает в себя следующие объекты

- P — **траектория страниц** - список страниц, в которых расположены используемые при выполнении программы адреса памяти
- M — **размер ОП**, измеряемый в физических страницах
- Алгоритм замещения

$\boxed{f = \frac{F}{|P|}}$ - **функция неудач**, где F — число страничных прерываний, $|P|$ — мощность траектории.

1. **Оптимальный алгоритм**
   Для удаления следует выбирать ту задачу, которой дольше всего не будет обращения. Единственный недостаток - невозможность реализации, так как траектория страниц заранее неизвестна.
   Пример использования:

P	4	3	2	1	4	3	5	4	3	2	1	5
M	4	4	4	4	4	4	4	4	4	2	2	2
		3	3	3	3	3	3	3	3	3	1	1
			2	1	1	1	5	5	5	5	5	5
F	+	+	+	+			+			+	+	

   $|P| = 12; F = 7 \Rightarrow f = \frac{7}{12} = 0{,}58(3)$

2. **FIFO (First In, First Out)**
   Для выгрузки выбирается страница, первой загруженная в память
   Тот же пример:

P	4	3	2	1	4	3	5	4	3	2	1	5
M	4	4	4	1	1	1	5	5	5	5	5	5
		3	3	3	4	4	4	4	4	2	2	2
			2	2	2	3	3	3	3	3	1	1
F	+	+	+	+	+	+	+			+	+	

   $|P| = 12; F = 9 \Rightarrow f = \frac{9}{12} = 0{,}75$
   Во-первых, недостаток алгоритма - нерациональность действия, так

как он может удалять страницы, которые будут использоваться.
Во-вторых, существует **аномалия страниц** - при увеличении размера ОП возможно увеличение числа страничных прерываний

P	4	3	2	1	4	3	5	4	3	2	1	5
M	4	4	4	4	4	4	5	5	5	5	1	1
		3	3	3	3	3	3	4	4	4	4	5
			2	2	2	2	2	2	3	3	3	3
				1	1	1	1	1	1	2	2	2
F	+	+	+	+			+	+	+	+	+	+

$$|P| = 12; |F| = 10 \Rightarrow f = \frac{10}{12} = 0{,}8(3)$$

3. **LRU (Least recently used)**
   Для выгрузки выбирается страница, которая дольше всего не использовалась.

P	4	3	2	1	4	3	5	4	3	2	1	5
M	4	4	4	1	1	1	5	5	5	2	2	2
		3	3	3	4	4	4	4	4	4	1	1
			2	2	2	3	3	3	3	3	3	5
F	+	+	+	+	+	+	+			+	+	+

$$|P| = 12; |F| = 10 \Rightarrow f = \frac{10}{12} = 0{,}8(3)$$

Результаты могут получатся и хуже, чем у FIFO, но зато у этого алгоритма отсутствует аномалия страниц

P	4	3	2	1	4	3	5	4	3	2	1	5
M	4	4	4	4	4	4	4	4	5	5	1	1
		3	3	3	3	3	3	3	4	4	4	5
			2	2	2	2	2	2	3	3	3	3
				1	1	1	5	5	1	2	2	2
F	+	+	+	+			+			+	+	+

$$|P| = 12; |F| = 8 \Rightarrow f = \frac{8}{12} = 0,(6)$$

Отсутствие аномалии можно доказать тем, что множество состоянии для столбцов при $|M| = 3$ включается в аналогичное для $|M| = 4$, и физически невозможна ситуация, когда прерывание возникает для второго случае и не возникает для первого.
Очевидный недостаток - большие накладные расходы. Поэтому в системах используется **бит обращения** - если было обращение, то бит ставится в 1, если все биты стали 1 - биты сбрасываются. Для удаления выбирается одна из нулевых страниц.

7.3.1.2. Явление пробуксовки

Пробуксовка - состояние системы, когда идёт интенсивный обмен страницами между ОП и внешней памятью, а процессор простаивает. Каждая задача ждёт, когда будет загружена нужная ей страница.
Эта проблема возникает при неограниченном уровне мультипрограммирования - на одну задачу приходится мало страниц физической памяти, что приводит к экспоненциальному росту числа страничных прерываний.

Стратегия рабочего множества позволяет преодолеть этот недостаток.

Идея заключается в том, что в ОП присутствуют те страницы, к которым задача обращается часто. В таком случае обмен страницами будет слабым.
Рабочее множество - такое множество страниц, которое нужно держать в ОП, чтобы в течении достаточно большого процессорного времени не произошло страничного прерывания. Если число свободных страниц меньше мощности рабочего множества, задача не загружается в ОП вовсе.

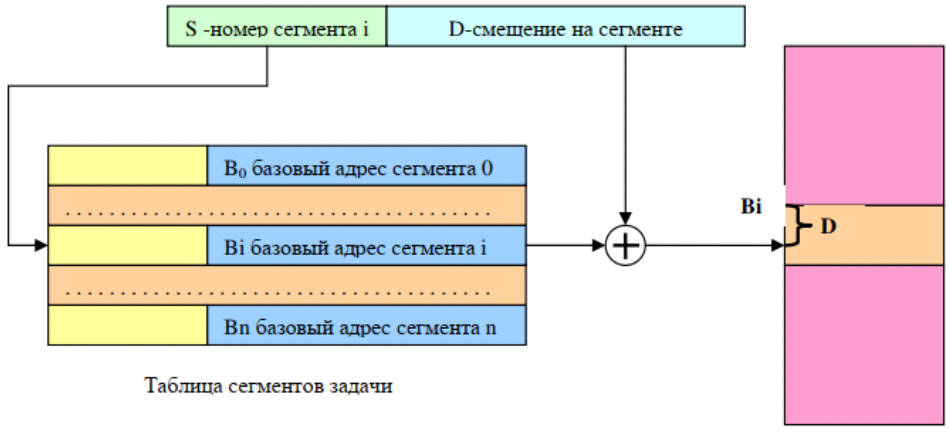
При разработке систем на основе такой стратегии необходимо руководствоваться **принципами локальности обращения:**

- **Локальность во времени -** если к данному слову памяти произошло обращение, то в ближайшее время с большой вероятностью обращение повторится
- **Локальность в пространстве** - если произошло обращение, то с большой вероятностью скоро произойдет обращение к соседним словам.

В алгоритмах оптимизации системы программирования должны пытаться повысить уровень локальности.

7.3.3. Управление виртуальной сегментной памятью по запросам

В вышеперечисленных стратегиях управления памятью логическое адресное пространство рассматривалось как односегментное, т.е. не разделялась память, используемая по-разному.
Сегмент - логическая группа информации - например, могут быть сегменты данных, кода и стека. Сегменты имеют переменную длину, что затрудняет работу.
При сегментно-страничной адресации программный адрес будет содержать номер сегмента S и смещение на сегменте D. Размер сегмента ограничен $|D| \leq 2^n - 1$, где $n -$ число разрядов, отводимое по смещение.



Аппаратные средства процессора должны поддерживать механизм динамического преобразования адресов и защиту сегментов.

Учёт состояния памяти:
Для каждой задачи создается **таблица сегментов.** Каждый её элемент содержит следующие поля

1. Базовый адрес сегмента в ОП
2. Бит состояния, показывающий, находится сегмент в ОП или во внешней памяти
3. Бит использования, показывающий, производилась ли запись в сегмент
4. Длина сегмента

Для учёта свободных участков создается **список свободных блоков**, а для учёта внешних сегментов - **таблица внешних сегментов**.

Запрос на **выделение памяти** поступает либо при активизации задачи, либо в момент выполнения по **сегментному прерыванию.**
При **освобождении памяти** нужно найти свободный участок - для этого могут быть использованы те же алгоритмы, что и в управлении динамическими разделами (7.2.2.2). Если свободный участок не найден, то происходит замещение.

Достоинства:

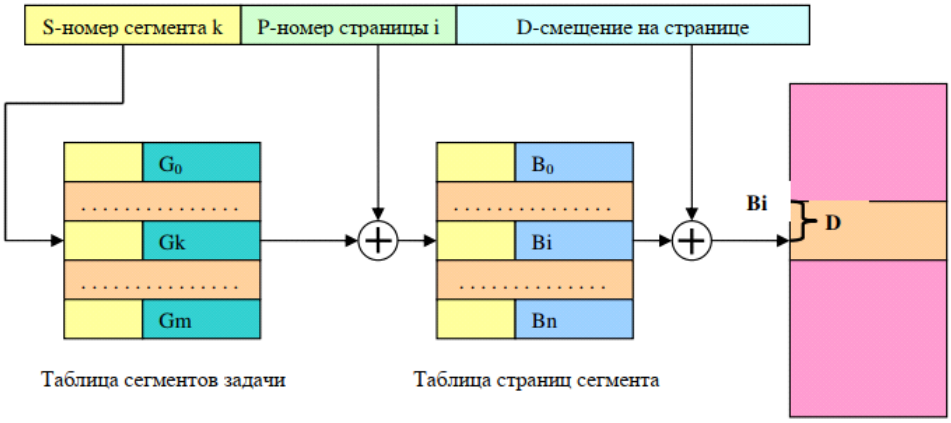
- Отсутствие фрагментации
- Большая виртуальная память
- Эффективное использование за счёт загрузки только используемых сегментов
- Упрощение разработки систем программирования, т.к. система управления памяти близка к логической структуре программы

Недостатки

- Высокие накладные расходы
- Ограничение размера сегмента
- Пробуксовка
- Большие сегменты снижают эффективность использования программы.

7.3.4. Управление виртуальной сегментно-страничной памятью

Основной недостаток сегментной памяти - переменный размер сегментов. Этот недостаток устраняется при использовании сегментно-страничного способа - логическое адресное пространство состоит из сегментов, а обмен между ОП и внешней памятью происходит страницами.
Таким образом, логическое пространство делится на сегменты, а сегменты - на страницы. Программный адрес состоит из **номера сегмента**, **номера страницы на сегменте** и **смещения на странице**.



Процессор поддерживает два прерывания - **сегментное** - когда в памяти отсутствует таблица страниц данного сегмента - и **страничное** - когда в ОП отсутствует страница.

Учёт состояния памяти:

1) **Таблица сегментов задачи** - связывает сегменты и страницы, принадлежащие сегментам. Каждый элемент содержит базовый адрес таблиц страниц данного сегмента, бит состояния, поля, связанные с защитой доступа и т.п.
2) **Таблица страниц сегмента** - устроена так же, как и таблица страниц в страничной памяти (7.2.3.2)
3) **Таблица физических страниц**
4) **Таблица внешних страниц**

Выделение и освобождения памяти осуществляется по алгоритмам управление виртуальной страничной памятью (7.3.1), используются те же алгоритмы замещения (7.3.1.1) и стратегия рабочего множества (7.3.1.2)

Достоинства:

- Большая виртуальная память
- Эффективное использование ОП за счёт загрузки части адресного пространства
- Неограниченный размер сегмента
- Отсутствуют недостатки, связанные с использование сегментов переменной длины
- Возможна защита сегментов

Недостатки:

- Пробуксовка
- Усложнение аппаратного обеспечения
- Высокие затраты памяти

8. Управление процессами и ресурсами

20 марта 2018 г. 11:57

Тема 8. Управление процессами и ресурсами

8.1. Процессы

Процесс - независимая работа, которая борется за ресурсы системы с другими независимыми работами.

Когда процесс возникает в системе, прежде всего создается его представитель - **Task Control Block**. В Windows и UNIX это называется **дескриптор процесса**. Там хранятся все данные, которые нужны управляющей программе, чтобы управлять этим процессом.
Каждый процесс создается для того, чтобы в его рамках выполнялась программа, но возможно и выполнение программы в рамках процесса или разделение программы разными процессами.

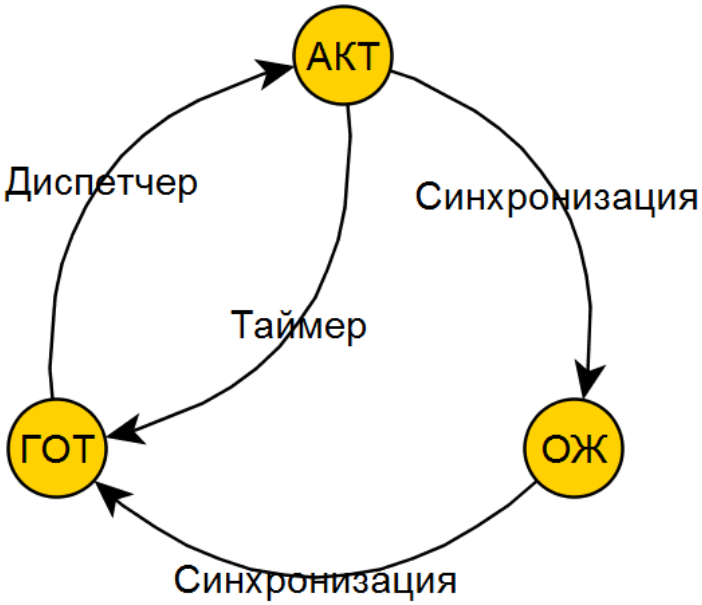
В разных системах ТСВ создаются по-разному.

- В z/OS и Linux к пулу управляющих блоков приписывается еще один блок.
- В UNIX есть "гнезда" для дескрипторов, при создании процесса ищется свободное гнездо и заполняется. При нехватке гнезд необходимо расширить область и перекомпилировать ядро.

Блок управления процессом:

1. **Идентификатор** (ID) процесса - уникальное целое беззнаковое число. Формируется в современных системах динамически
   Имя процесса - обычно имя программы, которая первой выполняется под этим процессом.
2. Информация о процессоре и полномочия процесса.
   Как правило сосредотачивается в **PSW** (Process State Word). К ней относится:
   - **Счетчик адреса команд** (СЧАК)**.** Указывает на следующую команду, с которой нужно начать, когда программа будет загружена в ЦП.
   - **Состояние процессора**
     - **Состояние супервизора** (SUP). Могут выполнятся все команды. Используется для системных процессов - например, система ввода/вывода.
     - **Пользовательское состояние** (PROB). Выполнение команд ограничено. Используется для пользовательских программ.
   - **Маскирования прерывания**. Некоторые программы ядра должны выполнятся таким образом, чтобы их никто не прерывал - например, программа управления памятью, методы

синхронизации. Чтобы программа стала непрерываемой, нужно маскировать все прерывания системой контроля машины.
    - **Ключ защиты памяти**. В мультипрограммном режиме память нужно защищать.
      Память выделяется блоками, к каждому блоку привязан регистр, хранящийся в PSW. Когда выполняется доступ памяти, PSW аппаратно сравнивается с блоком памяти, и - если совпадает - команда выполняется.
3. **Информация об используемой памяти**
   С точки зрения системы память занята, если она выделена под задачу. Но там может быть свободное место - например, если память выделена под кучу. Этой памятью управляет TCB.
4. **Информация по ресурсам процесса**. Информация о тех ресурсах, которые выделены данном процессу. Представляет собой указатели на различные блоки.
5. **Указатели на родственные процессы**. В мультипрограммных системах предоставляется возможность какому-либо процессу породить другой процесс. Можно, с помощью специальных функций, узнать ID родственного процесса, послать ему сообщение и т.п.
   Но такое обращение возможно только для родственных процессов. Чтобы можно было взаимодействовать с неродственными процессами, создается база данных с именами процессов и ссылками на все процессы. Каждый процесс регистрируется в этой базе.
6. **Информация о состояниях процесса**
   Все выделяемые состояния сводятся к 3-м:
   1. **Активное состояние.** Процесс находится в активном состоянии, если его программа выполняется на центральном процессоре.
   2. **Состояние ожидания**. Процесс ожидает завершения ввода/вывода или какого-то другого события.
   3. **Состояние готовности**. У процесса есть все ресурсы, чтобы занять ЦП, но ЦП занят чем-то другим. Процесс стоит в очереди готовых процессов.



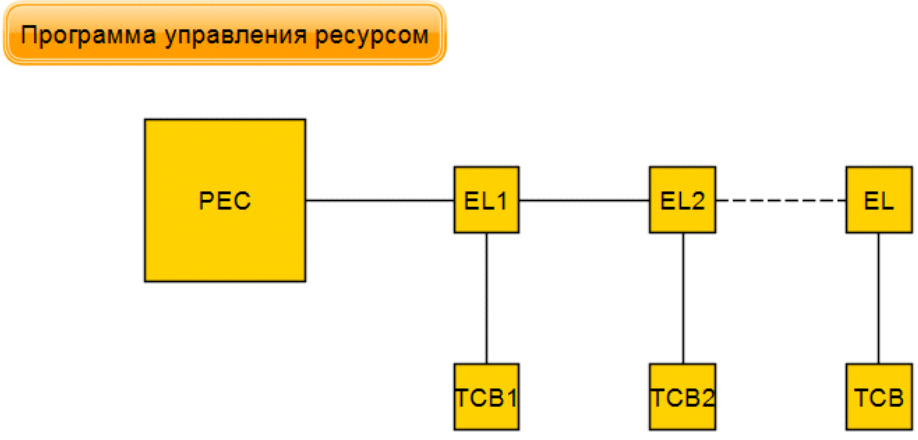
7. **Приоритет**. Влияет на место в очереди готовых процессов.

8.2. Ресурсы

Ресурсы - объекты, которые запрашиваются, используются, освобождаются и потребляются процессами. Ресурсом может всё что угодно, в том числе и программа.

Виды ресурсов:

- **Неразделяемые ресурсы** - монопольно используемые ресурсы. Предоставляется процессу, и до тех пор, пока процесс от него не откажется, принадлежит этому процессу. Пример - магнитная лента.
- **Разделяемые ресурсы**
  - **Совместно используемые**. Пример - область памяти, которая используется всеми процессами только для чтения.
  - **Повторно используемые** - разделяемые на последовательной основе. Это такой ресурс, который процесс запросил, получил в использование, использовал и освободил. Пример - чтение с диска или использование файлов для записи.
    Для управления таким ресурсом к нему ставится очередь.



Процесс EL1 в данный момент активен и использует ресурс. Когда он закончит, будет вызвано нужное прерывание. Ресурс будет

освобожден, EL1 исключается из очереди, а следующий процесс EL2 из состояния ожидания выводится в состояние готовности и вступает во владение ресурсом.

- **Потребляемые ресурсы** - такие ресурсы, который один процесс производит, а другой потребляет. После потребления в системе не существуют. Пример - сообщения, сигналы.

8.3. Планирование процессов

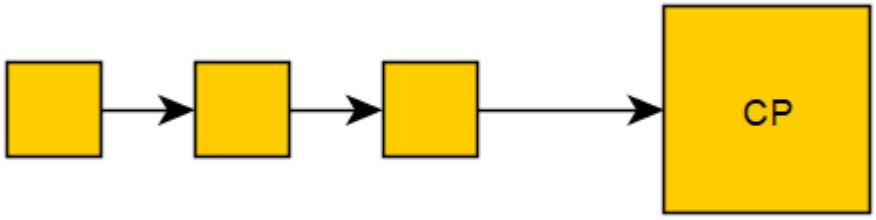
Шаги планирования:

1. Назначение приоритета
   В z/OS:
   ```
   //JOB …, PRTY=m
   …
   //EXEC …, PRTY=n
   ```
2. Постановка процесса в очередь готовых
3. Диспетчеризация

8.4. Алгоритмы диспетчеризации

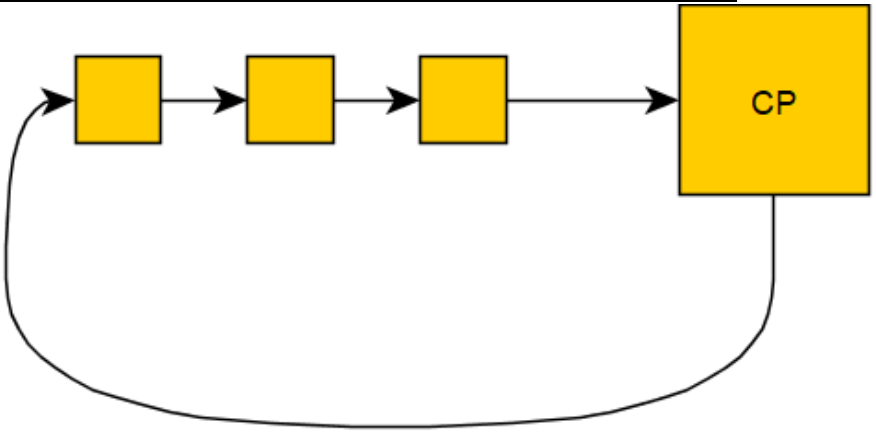
8.4.1. FIFO (First In, First Out)

Первый пришедший процесс обслуживается первым.



Данный алгоритм редко используется сам по себе, но (например) возможно использование очередей с приоритетами - в таком случае процессы с одинаковыми приоритетами будут обслуживаться по принципу FIFO.
FIFO подходит для пакетных систем, но не для диалоговых систем и не для систем реального времени.

8.4.2. Равномерное циклическое квантование



Диспетчеризация происходит подобно FIFO, но каждый процесс получает

временной квант, определяющий время, доступное процессу на ЦП. После истечения времени процесс становится в конец очереди.
Существуют 2 модификации:

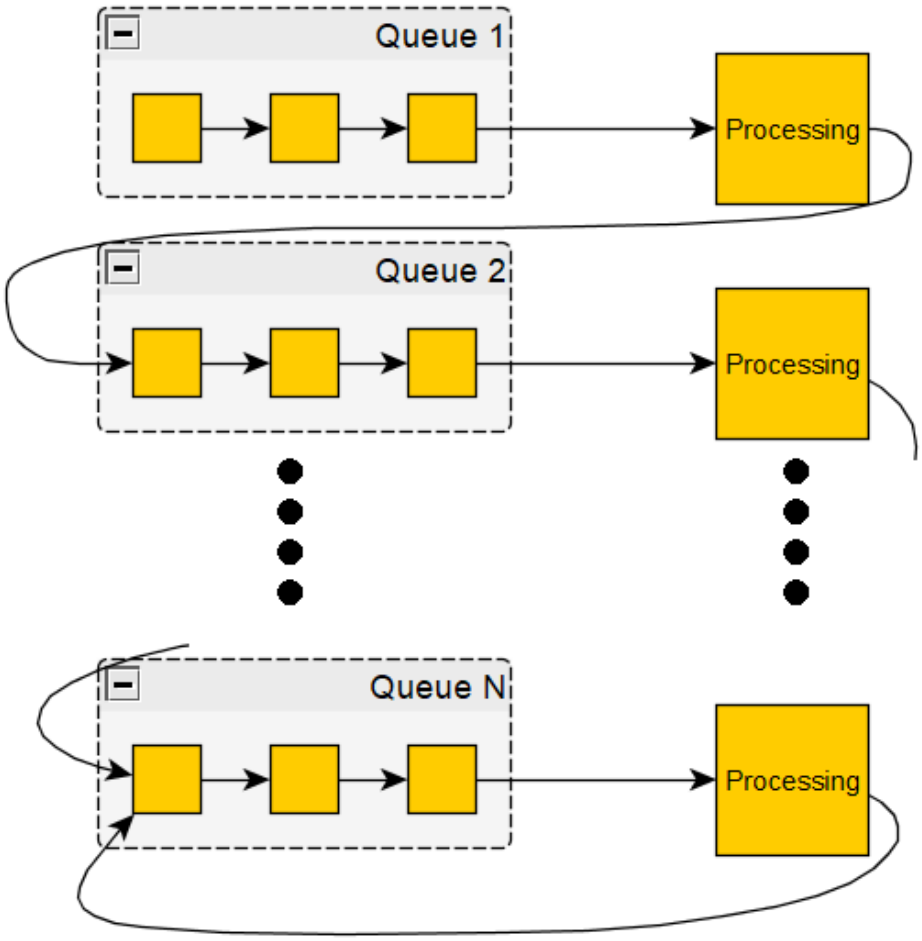
1. **Модификация Коффмана** - каждая задача, находящаяся в очереди, получает дополнительный квант времени, если поступает новая
2. Каждая задача получает число квантов, равно числу задач в очереди

Недостаток этого метода - балансирование между вырождением в FIFO и высокими накладными расходами.

8.4.3. Алгоритмы с обратной связью

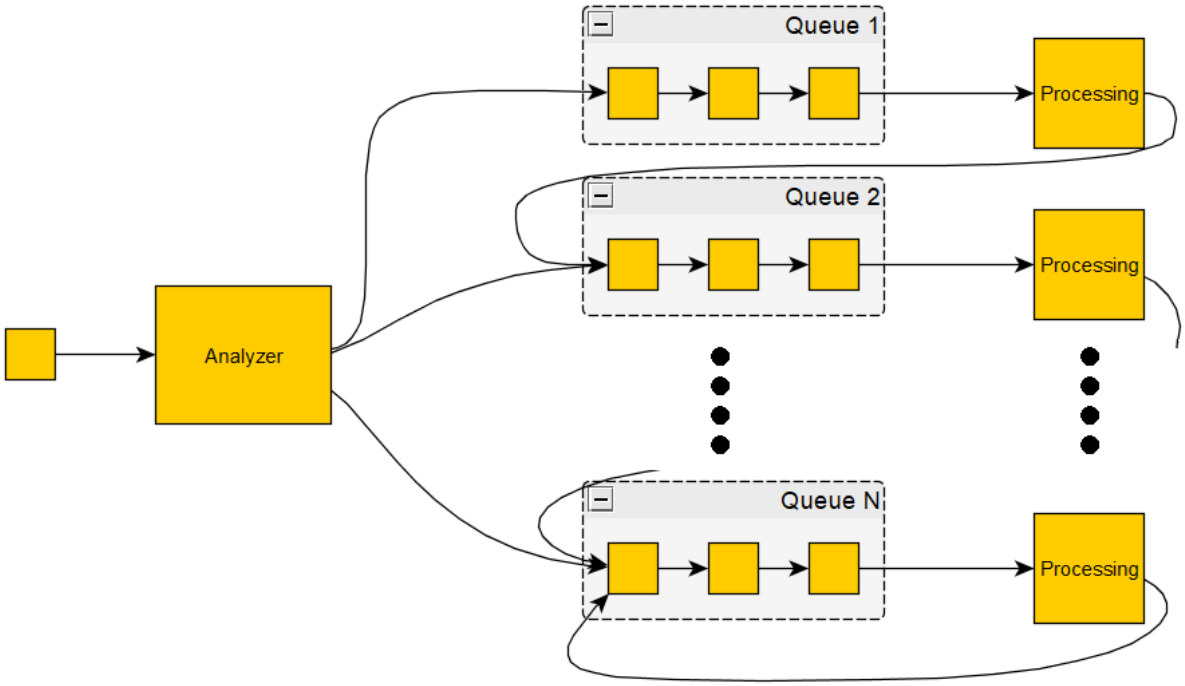
Такие алгоритмы определяют приоритет процесса по характеру использования им кванта времени.

8.4.3.1. FB_N – обратная очередь с N очередями



В таком алгоритме квант времени, предоставляемый задаче, постоянен. Если задача не отработала за время $\Delta t = q$, она перемещается в следующую очередь. В нижней очереди процесс циркулирует до самого завершения.
Таким образом, чем больше времени ЦП занимает задача, тем меньше становится её приоритет. Это обеспечивает быстрое выполнение диалоговых задач.

8.4.3.2. Алгоритм Корбато



Этот алгоритм похож на предыдущий, но отличается тем, что процессы здесь предварительно сортируются по очередям, а в каждой очередь квант времени дается $\Delta t = 2^n q$, т.е. экспоненциально зависящий от номера очереди.
Распределение может проходить, например, по формуле Корбато: $i = \log_2 \frac{l_p}{l_{tk}} + 1$, где l_p – длина программы в байтах, l_{tk} – число байт, которые могут быть переданы между ОП и внешней памятью за Δt.

8.4.4. Алгоритмы, использующиеся в ОСРВ

В отличие от диалоговых и пакетных систем, главное требования при разработке ОСРВ - предсказуемость, в частности - ограниченное время отклика. Для каждого процесса ставятся "дедлайны", и если процесс не может быть выполнен, не пропустив ни одного дедлайна, он должен быть отклонён.

8.4.4.1. Статическое расписание

Расписание работы составляется заранее, планировщик следует расписанию. Это просто и надёжно, но негибко и отвязано от внешнего мира

8.4.4.2. Ограниченное равномерное циклическое квантование

Для каждой задачи устанавливается ограниченное число раз выполнения, после чего задача перемещается в фоновую очередь, из которой задача помещается в основную, когда основная освободится

8.4.4.3.

Задачи распределяются на 16 или 32 уровня. Сначала выполняется задача на уровне $2^n - 1$, потом на уровне ниже.

9. Взаимодействие и синхронизация процессов

3 апреля 2018 г. 11:42

Тема 9. Взаимодействие и синхронизация процессов

9.1. Параллелизм

Наиболее эффективный способ увеличение производительности - распараллеливание.

Виды параллелизма

- **Естественный параллелизм** - есть несколько физических процессоров, на которых может выполняться несколько процессов.
  - **Параллелизм независимых ветвей** - программы делятся на независимые ветви, которые могут выполнятся параллельно.
  - **Параллелизм объектов или данных** - совокупность данных обрабатывается параллельно одной программой
- **Логический параллелизм** - есть смесь задач, элементы которой поочерёдно становятся активными.

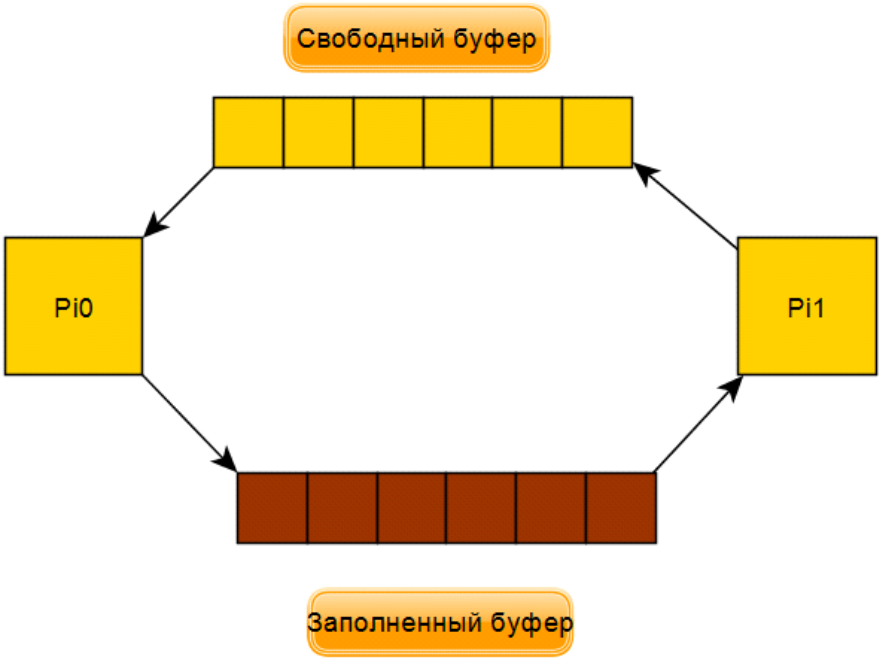
Логический параллелизм не приносит никаких дивидендов в плане быстродействия, но является удобным способом построения больших систем.

9.2. Проблемы синхронизации процессов

9.2.1. Проблема критической секции. Если какие-то части кода работают с общим **критическим ресурсом** (которым может владеть только один процесс), эти части (**критические секции**) должны синхронизироваться на основе взаимного исключения:

- Если процесс находится в критической секции, то никакой другой процесс не может находится в этой секции
- Процесс не может бесконечно долго находится в критической секции
- Процесс не может бесконечно долго ждать входа в критическую секцию

9.2.2. Проблема "Поставщик - Потребитель"



Если процесс Pi0 работает быстро, то он быстро исчерпает свободные буферы и уйдет ждать освобождения буфера. Если же Pi1 работает быстрее, он быстро исчерпает все заполненные буферы и будет ждать.

Код синхронизации для Pi0:

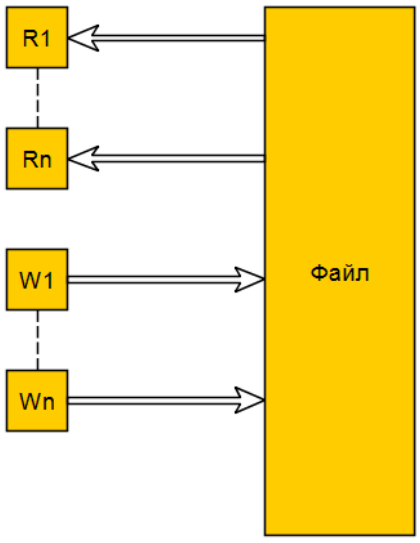
```
M: if (E == 0) goto M
    E--
    <ввод>
    F++
    goto M
```

Код синхронизации для Pi!:

```
M: if (E == 0) goto M
    F--
    <вывод>
    E++
    goto M
```

Если свободных буферов нет, то код остается в первой строчке ждать. Этот механизм называется **занятое (активное) ожидание** - потому что занимается квант времени процессора. Поэтому код такого вида не используется - создан механизм, не запускающий процесс, пока не выполнится условие, но суть от этого не меняется.

9.2.3. Проблема "Читатели - Писатели"



Если файл и ряд процессов, читающих и записывающих в файл. Читатели могут читать файл одновременно, но писатель может быть только один и только когда файл не читается.

Синхронизация:

R — число читателей, читающих файл.

$W = 1 \Rightarrow$ писатель пишет

$W = 0 \Rightarrow$ нет

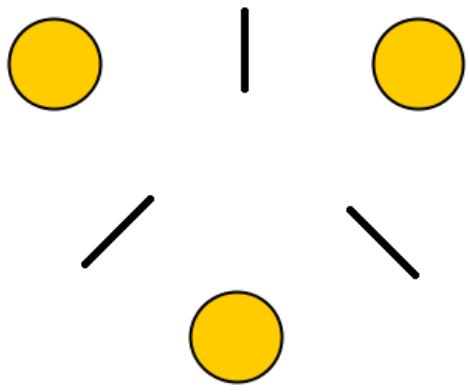
Код синхронизации для писателя:

```
Gi: if ((W!=0) && (R!=0)) goto Gi
    W = 1
    <Запись в файл>
    W = 0
end
```

Код синхронизации для читателя:

```
Ri: if (W!=0) goto Ri
    R = R + 1;
    <чтение из файла>
    R = R - 1
end
```

9.2.4. "Китайские обедающие философы"



Есть три философа. Для того, чтобы поесть, философу нужны две

палочки. Когда две палочки свободны, философ начинает есть. Если двух палочек нет, философ беседует.

9.3. Механизмы синхронизации

Все механизмы синхронизации должны быть непрерываемы во избежание появления проблемы критической секции.

9.3.1. Синхронизация в UNIX

В ядре UNIX есть следующие функции:

```
wakeup(&runout);
sleep(&runout, Prior);
```

Когда один процесс заснул (не выполнилось какое-то условие), то другой процесс дает wakeup, и все процессы, заснувшие по данному событию, просыпаются.

Но может сложиться ситуация, когда sleep выполнится после wakeup. Поэтому в UNIX создали событие &lbolt, и диспетчер через определенные интервалы времени делает

```
wakeup(&lbolt);
```

Поэтому процесс, который синхронизируется, должен работать следующим образом:

```
while (cond)
    sleep(&lbolt, Prior);
```

Это и есть активное ожидание. Подобные решения в UNIX вызваны желанием обеспечить как можно меньшую зависимость от железа. Цена этого - более низкая скорость работы.

9.3.2. Синхронизация с помощью "семафора"

Изобретен Дейкстрой.

Для ресурса создается семафор $S \geq 0$. Для него определены две операции:

```
P(S):
{
    if (S == 0) sleep(&S);
    else if (S > 0){
        S = S - 1;
    }
}
V(S):
{
    S = S + 1;
    wakeup(&S);
}
```

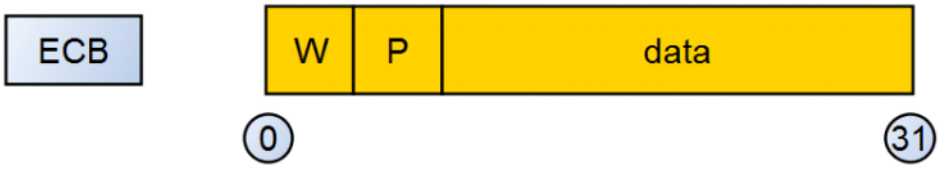
Первая операция вызывается при доступе к ресурсу. Она либо уменьшает семафор на 1, либо, если уменьшать некуда, переводит процесс в состояние ожидания, связанное с событием S.

V(S) вызывается при освобождении ресурса - она увеличивает семафор и будит процессы, заснувшие в ожидании ресурса.

Для решения проблемы критической секции с помощью семафора S связывается с критическим ресурсом. В начальный момент $S = 1$.
CS — выполнение критической секции
$P_1: P(S); CS; V(S)$
…
$P_i: P(S); CS; V(S);$

9.3.3. Синхронизация в z/OS - POST/WAIT

Механизм синхронизации на основе события. События представлены структурами данных - **ECB (Event Control Block)**. Эти блоки используются макросами **POST, WAIT** и **EVENTS**. POST сигнализирует о завершении события, WAIT и EVENTS сообщают, что задача не может продолжаться, пока не произойдут какие-то события.



W - Wait Bit, P - Post Bit. В данных может хранится адрес Request Block или код завершения.
Когда ECB создается, W=P=0. WAIT ставит W = 1, а POST - P = 1, W = 0.
WAIT может работать и со списком из ECB.

```
A:
ATTACH EP=B, ECB=ECB_B
ATTACH EP=C, ECB=ECB_C
ATTACH EP=D, ECB=ECB_D
…
WAIT 3, ECBLIST = ECB_B

{B,C,D}:
POST ECB_{B,C,D}, RC_{B,C,D}
```

WAIT в A переведет задачу A в состояние ожидания. 3 - общее число ожидаемых сообщений.

При выполнении $ATTACH$ образуются $TCB_{\{B,C,D\}}$. RB — Request Block. Вид списка ECB:

[ECB_B]	1	0	&RB_A
[ECB_C]	1	0	&RB_A
[ECB_D]	1	0	&RB_A

RC - код завершения процедуры.

В списке ECB после выполнения задачи B будет установлен флаг POST для B, а на место адреса RB_A установится код завершения B.

[ECB_B]	1	1	RC_B

[ECB_C]	1	0	&RB_A
[ECB_D]	1	0	&RB_A

В RB у TCB-блока A есть счётчик ожидаемых событий. Когда выполнится POST процедуры B, в счётчик будет записано $3 - 1 = 2$ и список будет иметь вид:

[ECB_B]	0	1	RC_B
[ECB_C]	1	0	&RB_A
[ECB_D]	1	0	&RB_A

Когда завершится задание C, список будет иметь следующий вид:

[ECB_B]	0	1	RC_B
[ECB_C]	0	1	RC_C
[ECB_D]	1	0	&RB_A

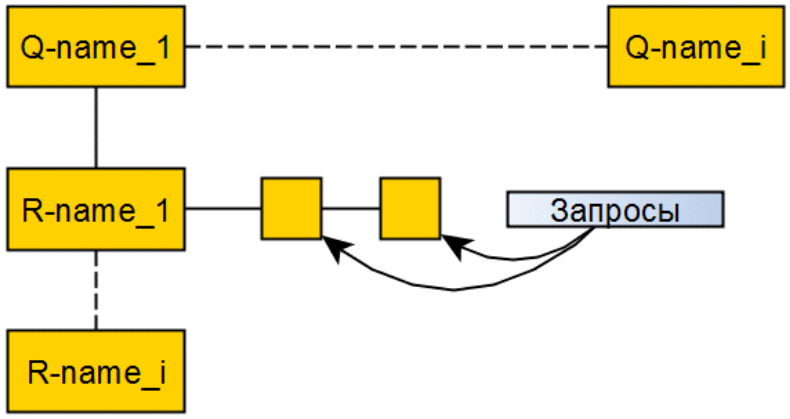
В счётчик в RB_A будет установлено 1.
Список после выполнения D:

[ECB_B]	0	1	RC_B
[ECB_C]	0	1	RC_C
[ECB_D]	0	1	RC_D

После выполнения D счётчик ожидаемых событий будет иметь значение 0 и POST в D переведет задачу A из состояния ожидания в состояние готовности.

9.3.4. Синхронизация в Z/OS - ENQ/DEQ

Способ синхронизации для повторно используемых ресурсов.
P_1
…
$ENQ\ QNAME, RNAME$ //Запрос ресурса
⋮ //Владение ресурсом
$DEQ\ QNAME, RNAME$ //Освобождение ресурса



DEQ выкидывает элемент из очереди и переводит процесс из состояния ожидания в состояние готовности.

9.3.5. Синхронизация в QNX - посредством обмена сообщениями

Сообщения - потребляемый ресурс. Один процесс его производит, другой потребляет.

```
P_1:
char BUF[128] //Посылаемое сообщение
SEND (id, BUF) //Посылает сообщение по id процесса
```

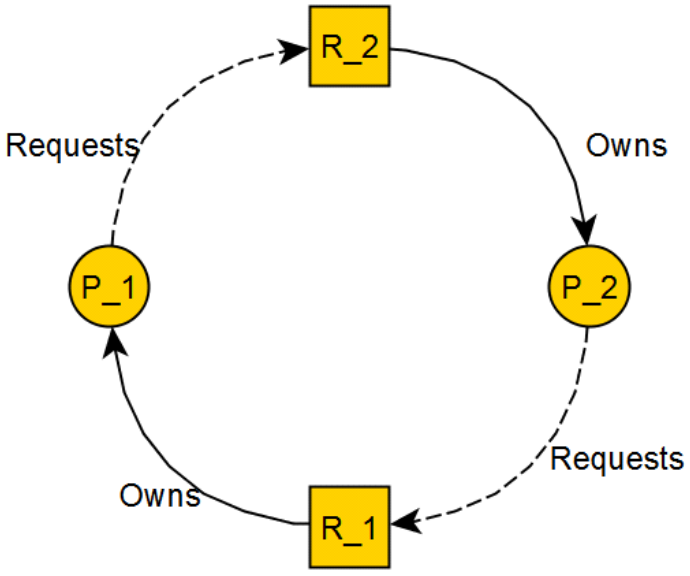
Процесс переходит в состояние ожидания и ждет ответа (В QNX). В UNIX процесс переходит в состояние ожидания, когда MailBox переполнился.

```
P_2:
id_S = RECEIVE (id, BUF) //Если RECEIVE выполнилось
   до SEND, P_2 переходит в состояние ожидания. id_S
   устанавливается равным id процесса-отправителя.
REPLY (id_S, BUF) //Освобождает процесс P_1
id_S = CRECEIVE (id, BUF) //Не останавливает
   процесс, если сообщение не пришло
```

9.4. Тупиковые ситуации

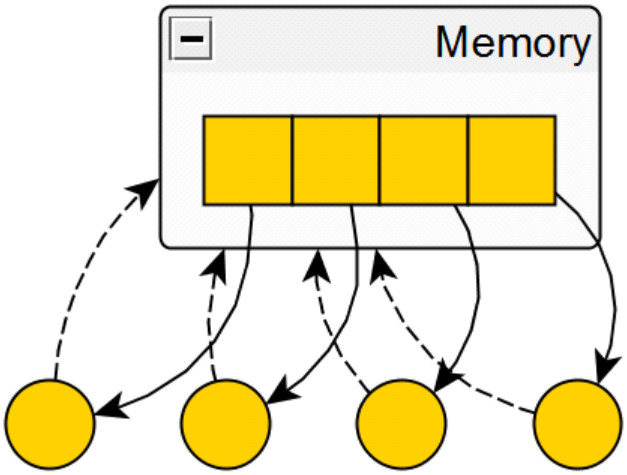
Пусть в ОС есть два процесса P_1, P_2 и два критических ресурса R_1, R_2. Допустим, происходит следующее:

1. P_1 запрашивает и берёт R_1
2. P_2 запрашивает и берёт R_2
3. P_1 запрашивает R_2 и уходит в состояние ожидания
4. P_2 запрашивает R_1 и уходит в состояние ожидания



Система зависла - **тупик**.

Если ресурс многоединичен (например, как ОП), тоже возможен тупик - например, если все процессы выпросят всю свободную память и попросят ещё.



Для предотвращения такой ситуации в универсальных системах разрешается запрашивать память 1 раз. Некоторые системы завершают программу при нехватке памяти. В UNIX и Linux при нехватке памяти процесс выгружается в swap-область и становится в очередь.

Ещё один пример тупика: пусть есть три сообщения S_1, S_2, S_3

```
P_1: wait(S_3); … ; send(S_1);
P_2: wait(S_1); … ; send(S_2);
P_3: wait(S_2); … ; send(S_3);
```

При запуске всё зависнет. Чтобы этого избежать, нужно сначала посылать сообщения, а потом ждать.

9.4.1. Решение проблемы тупика

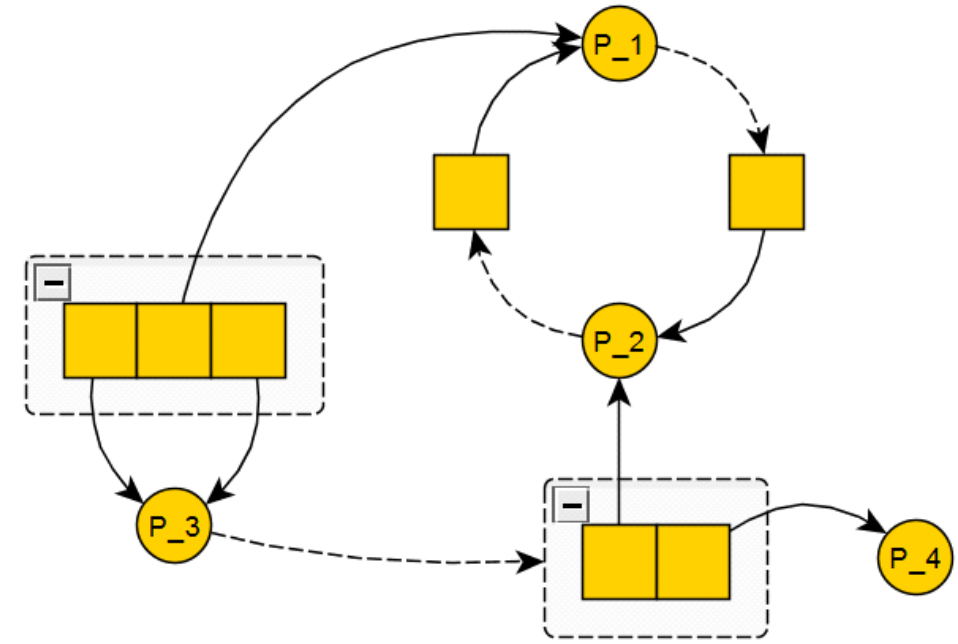
1. Распознавание тупика
2. Восстановление тупика
3. Предотвращение тупика

9.4.1.1. Распознавание тупика

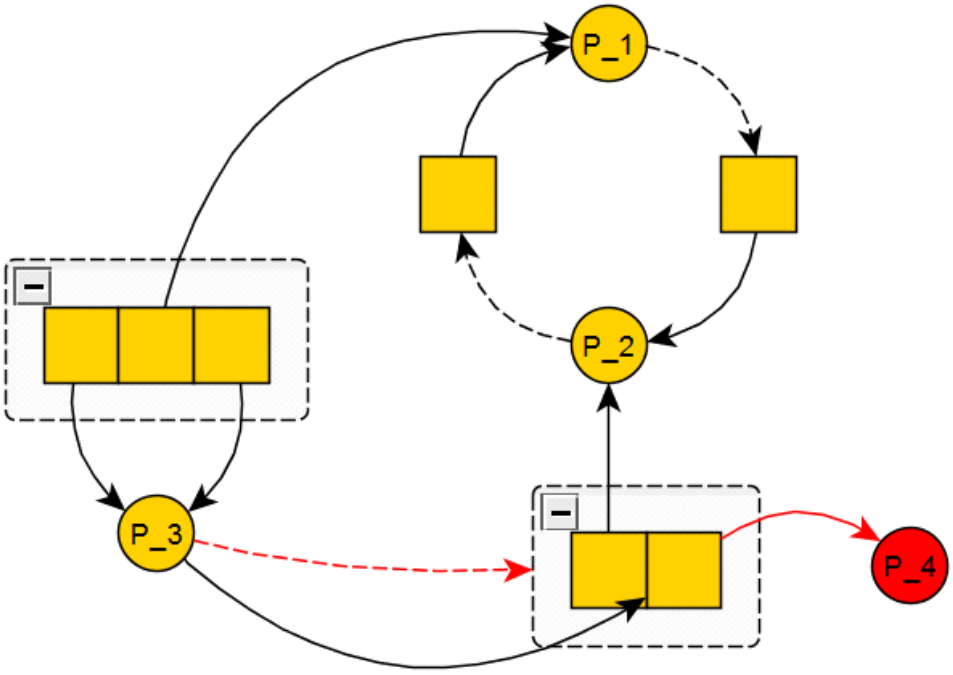
Теорема Холта. Необходимое (но не достаточное) условие существования тупика - цикл в графе. В **модели Холта** используется следующий метод:

1. Активный процесс освобождает ресурсы.
2. Если какой-то процесс стал активным, назад на шаг 1.
3. Если в системе остались процессы, то тупик.

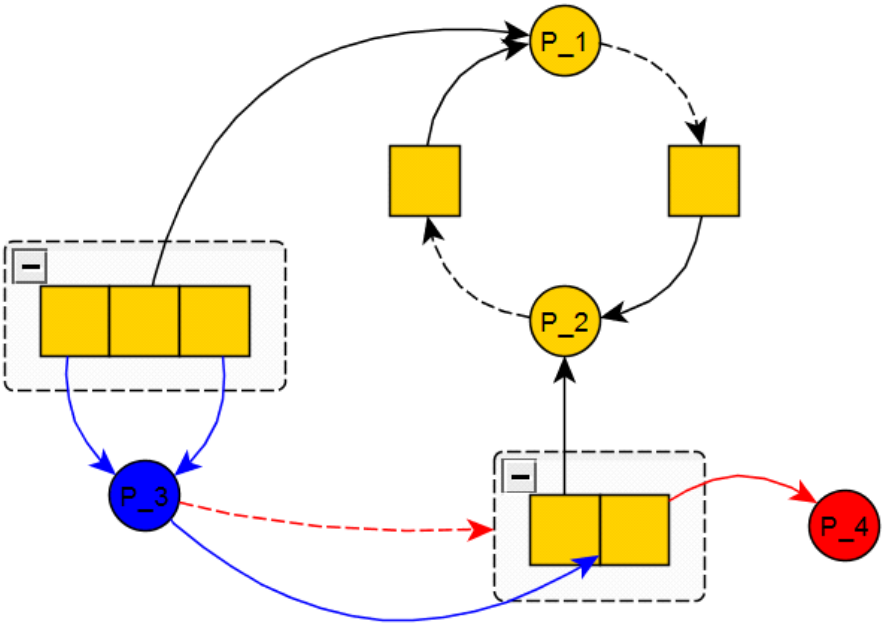
Пример. Состояние системы:



Шаг 1. Удаляется процесс P_4



Шаг 2. Удаляется процесс P_3



На третьем шаге удалить нечего. Тупик найден.

9.4.1.2. Восстановление тупика

Для восстановления тупика можно убить процесс или отнять ресурсы у ждущего процесса. Но этот способ создает массу осложнений, например, с модифицирующимися ресурсами. Кроме того, это нельзя делать в универсальных системах.

9.4.1.3. Предотвращение тупика

Для предотвращения тупика необходимо построить систему таким образом, чтобы тупики никогда не возникли. Есть разные способы, как этого добиться.

1. **Предварительное описание ресурсов (z/OS)**. Недостаток в том, что для этого нужно знать, какие ресурсы потребуются задаче. Также это уменьшит рациональность использования ресурсов.
   Для реализации такого подхода создается отдельный модуль **(initializer)**, собирающий ресурсы. Если все ресурсы собрать не получается, то задача уходит в состояние ожидания и не запускается.
2. **Контроль за распределением ресурса**.
   Возможно построение модели Холта перед запуском программы, но из-за медленности работы на практике не применяется.
   Иногда используется разбиение ресурсов на классы и удовлетворение запросов в порядке возрастания классов.

10. Управление устройствами

Tuesday, May 8, 2018 11:44

Тема 10. Управление устройствами

10.1. Функции управления устройствами

1. **Отслеживание состояния устройств**. Каждому устройству сопоставляется **UCB (Unit Control Block)**. Он хранит характеристики устройства, тип, состояние - offline/online, ready/not ready, free/busy и очередь запросов.
2. **Планирование** - определение, какому процессу, в какой момент времени и на какой срок выдать устройство. Здесь выделяются три типа устройств:
   - Монопольно используемые устройства
   - Последовательно используемые устройства
   - Виртуальные устройства
3. **Управление устройствами** - непосредственное выделение, построение таблиц, отметка в UCB - блоке и освобождение. Способы управления зависят от устройства.

10.2. Характеристика устройств

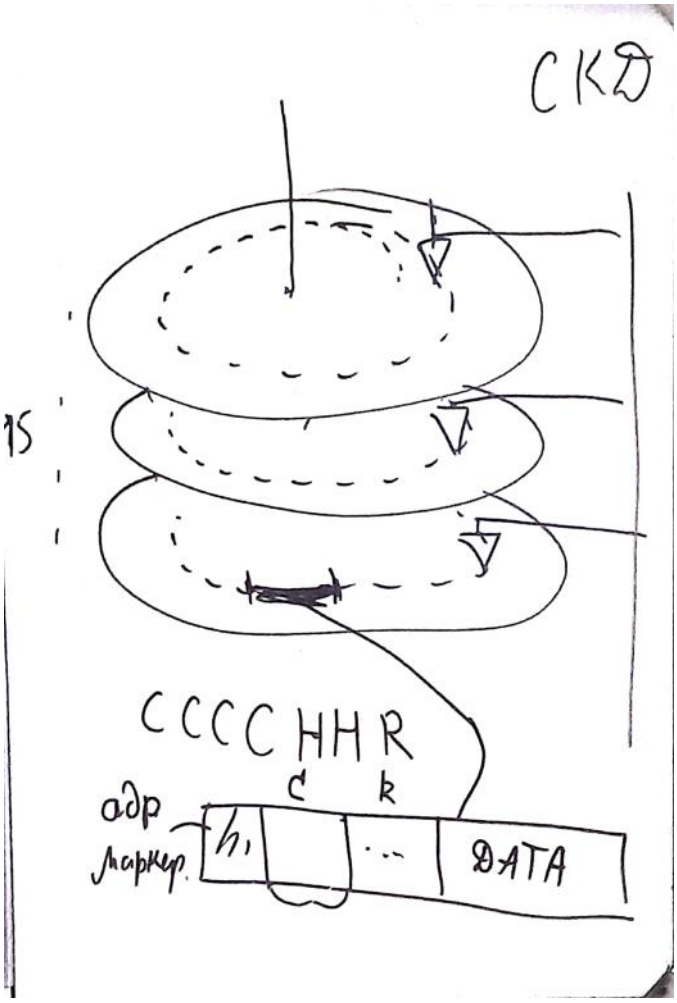
Все устройства можно поделить на следующие группы:

1. Устройства ввода\вывода (не имеют файловой структуры)
2. Запоминающие устройства (имеют файловую структуру)
   a. Запоминающие устройства последовательного доступа.
   b. Запоминающие устройства прямого доступа.

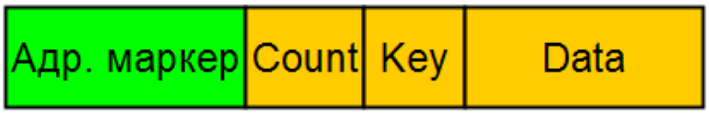
10.2.1. Магнитная лента

HOT, EOT - Head/End of Tome, HOF/EOF - Head/End of File

10.2.2. Диск



Диски расположены друг над другом. Ряд концентрических окружностей на диске - **дорожки**. Дорожки, располагающиеся друг над другом, образуют **цилиндр**.
Адрес формируется из адреса цилиндра, номера дорожки (трека) в цилиндре, номера записи в треке - *CCCCHHR*
В FBA-устройствах все треки поделены на одинаковое число записей (512 байт) - **блоков**
Запись на дорожке устроена следующим образом:



Адресный маркер - говорит, что начинается запись на дорожке
Count - число информации в записи
Key - ключ для индексных наборов данных
Блок устроен также, только без ключа

10.3. Организация работы с внешними устройствами

Существует два способа ввода/вывода - **прямой** и **косвенный**.

10.3.1. Прямой ввод/вывод - основан на шине.

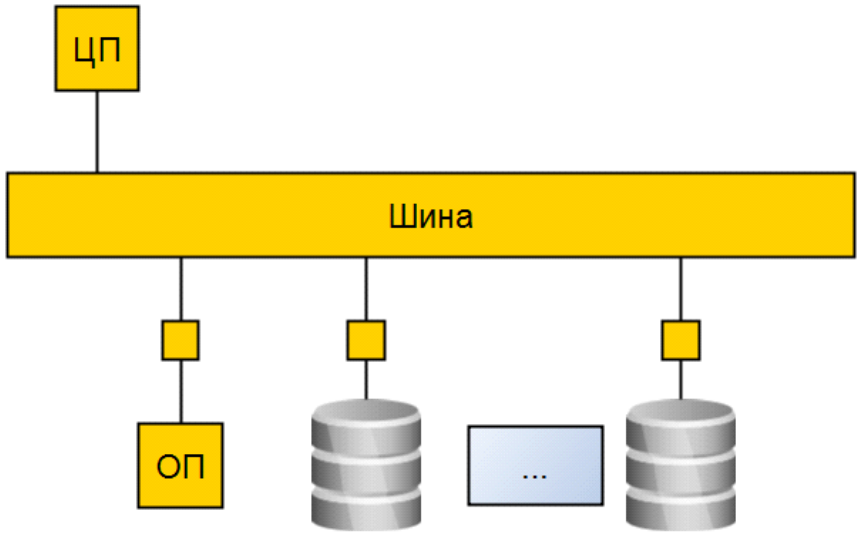
Шина имеет разные типы линий:

- Управляющие
- Адресные

- Для передачи данных

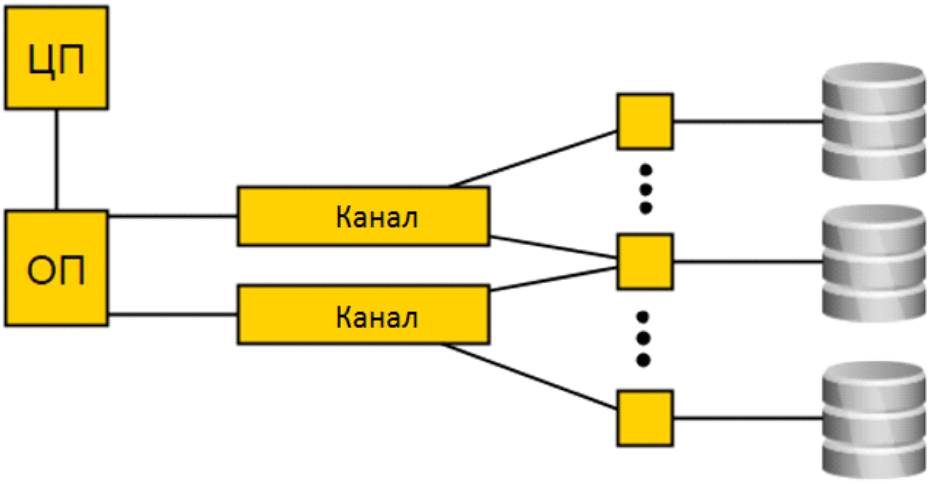
Арбитр шины определяет тип прерывания, запускающегося при завершении работы с шиной. Более высокий приоритет имеют устройства, подключенные ближе к шине.

Программа в ОП - **драйвер устройства** - выполняется на ЦП. Драйвер получает адрес устройства и захватывает шину, по адресной шине устанавливает связь с контроллером, по управляющей шине отдает команды устройству.
У каждого драйвера есть свой вектор прерывания. При подключении устройства контроллер инициирует прерывание нужного драйвера.



Такой ввод/называется прямым, так как программа запущена на ЦП и занимается вводом/выводом. В это время другие устройства не могут осуществлять ввод/вывод и пользоваться шиной.
Чтобы освободить ЦП, было сделано **устройство прямого доступа к памяти**, которое освобождает процессор от соответствующих операций. Шина всё равно будет занята.
Основная проблема в таком случае - как найти подходящий драйвер.

10.3.2. Косвенный ввод/вывод - вводом/выводом занимается специальное периферийное устройство - **канал**



В ОП лежит канальная программа. Достоинства такого метода в том, что процессор свободен во время ввода\вывода. Кроме того, обеспечивается дополнительная надежность системы за счёт дублирования каналов. Но такое исполнение намного сложнее и дороже.

10.4.Реализация функций управления устройствами

10.4.1.Монопольно используемые устройства - устройства, которые могут использоваться только одним пользователем.
например, магнитная лента.
Учёт - в UCB-блок устройства ставится флаг - занят/свободен.
Планирование - устройство занимается на всё время.

10.4.2. Разделяемые устройства - устройства, которые могут использоваться несколькими пользователями попеременно.
Учёт - UCB-блок, таблица каналов, таблица контроллеров, таблица логических путей.
Планирование - при запросе ввода/вывода таблицы связываются непосредственно с задачей, которую запрашивает ввод-вывод - аллоцирование наборов данных

10.4.2.1. Методы планирования разделяемых устройств

При маленьких устройствах обычно используется очередь FIFO. Это не вызывает проблем в большинстве случаев.
При устройствах большого объема делают оптимизации. Рассмотрим некоторые примеры алгоритмов.

Оптимизация по времени поиска цилиндра
К диску идут запросы, содержащие адрес цилиндра, на чтение нужной информации. Запросы упорядочиваются определённым способом, чтобы обеспечить минимальное движение считывающей головки.
Оценка эффективности оптимизации - дисперсия времени доступа - разные агенты должны иметь равные права доступа к диску.
Для этого есть следующие алгоритмы:

1. **SSTF (Short Seek Time First)**. Из запросов выбирается ближайший к текущему положению головки. Основная проблема - дискриминация крайних дорожек.
2. **SCAN** - сканирование. Выбирается также ближайший запрос к текущему положению головки, но только в одном направлении. Этот алгоритм обладает намного меньшей дисперсией, чем SSTF.
3. **N-step SCAN**. Те запросы, которые пришли после начала движения в заданном направлении, не рассматриваются. Это сокращает дисперсию ещё сильнее.
4. **C-SCAN** - запросы обрабатываются только в одном направлении, в другом - просто выводятся.

Оптимизация по времени ожидания записи

У каждой записи есть свой номер. Если номера будут в разном порядке, то будет много лишнего вращения - цилиндр крутится только в одном направлении.

Если диски большие, то поиск цилиндра - приоритетная оптимизация. Но есть диски с фиксированными головками или барабаны. В них приоритетнее оптимизация по записи.

10.4.3. Виртуальные устройства

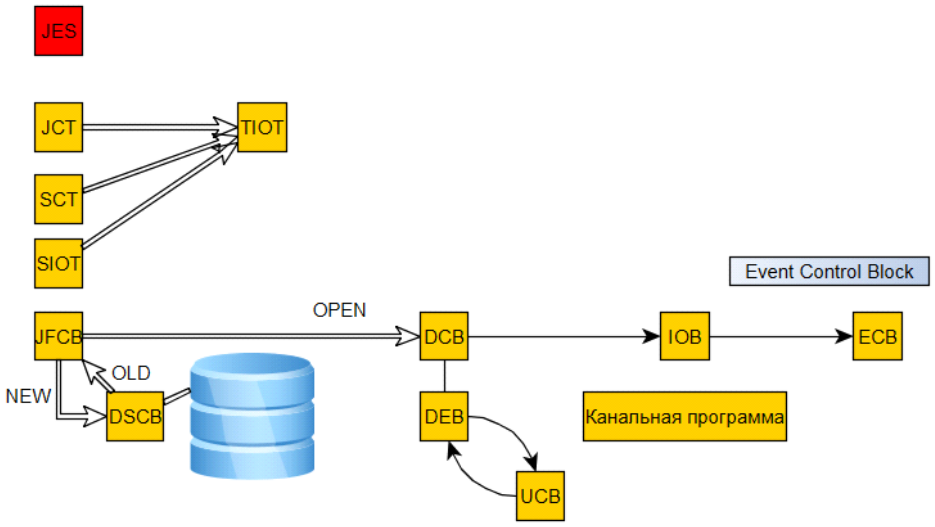
- использование какого-то другого устройства вместо того, которое имеем в виду. Пример - печать в файл вместо принтера. Могут быть виртуальные диски - использование куска памяти с интерфейсом, как у диска.

Первая виртуализация - виртуализация ввода с перфокарт. Записи с перфокарт записывались на диск, и уже с диска ридер считывал, как с перфокарты. Это повысило производительность

11. Организация ввода/вывода в ОС

Tuesday, May 8, 2018 12:55

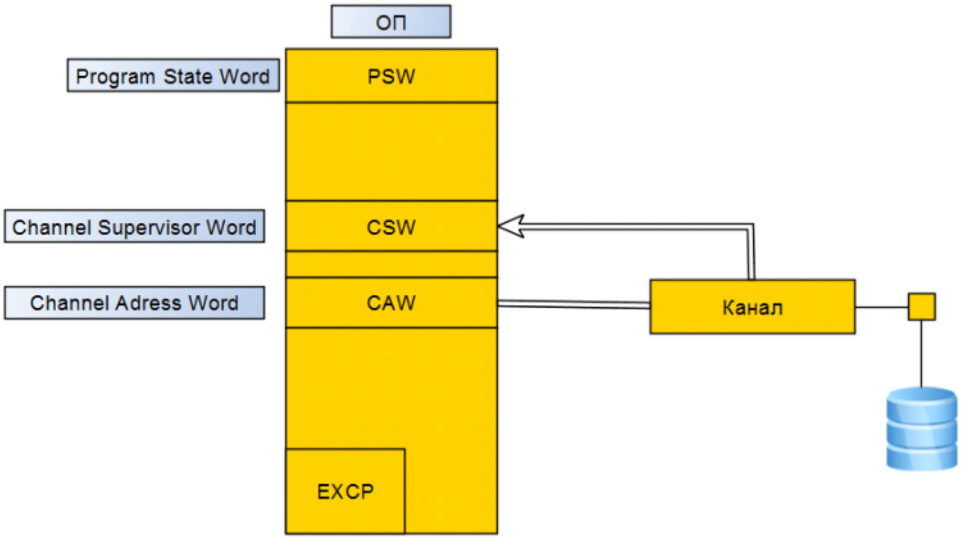
Тема 11. Организация ввода/вывода в ОС
11.1. Организация ввода/вывода в z/OS



- TIOT - Task IO Table
  - JCT - Job Control Table
  - SCT - Step Control Table - таблица управления шагов задания
  - SIOT - Step IO Table - таблица ввода/вывода шага
- JFCB - Job File Control Block - описывает файл в задаче
- DSCB - Data Set Control Block - блок управления файлом
- DCB - Data Control Block - описывает данные в файле - тип данных и тип доступа
  - DEB - Data Extended Block - помимо этого, хранит реальный адрес устройства
  - UCB - Unit Control Block
- IOB - IO Block
- ECB - Event Control Block

Для каждого DD-оператора аллокируется набор данных на диске.
Если диспозиция оператора OLD, то на диске отыскивается DSBD (Data Set Control Block) ("дескриптор" файла), и эта информация перекачивается в JFCB. Если диспозиция NEW, то DSCB создается.
В программе находится DCB (Data Control Block). Макрокоманда OPEN берет информацию из JFCB и отыскивает расположение информации, после чего создается DEB (Data Extend Block).

10.2. Физическая система ввода вывода



EXCP-супервизор (выполняет канальную программу) имеет 2 входа - запрос на ввод/вывод и прерывание от канала "ввод/вывод закончен".

При запросе на ввод/вывод:

1. Проверяется корректность IOB (Input/Output Block)
2. Создается элемент запроса
3. Просматриваются управляющие блоки устройств и определяется, доступны ли устройства. Определяется логический путь.
4. Программа проверки каналов по таблице слов логических каналов определяет физический канал
5. В таблице устройств выбирается модуль начала ввода/вывода, которому передается управление.
6. Если операция не может быть начата, то ставится в очередь, иначе - выполняется.

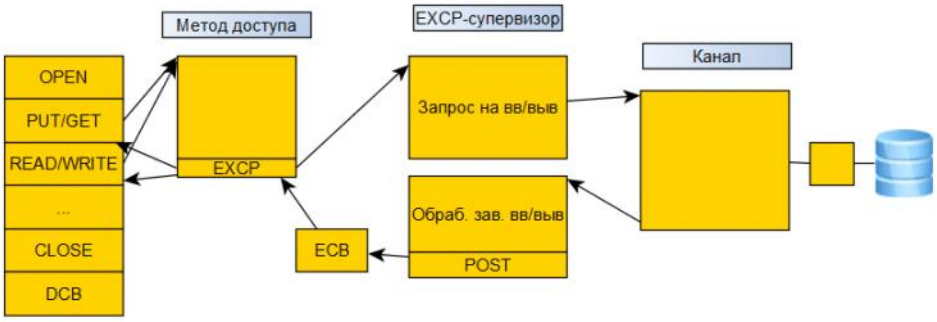
При окончании ввода/вывода:

EXCP-супервизор получает управление по прерыванию от канала.

1. По прерыванию от отыскивает UCB устройства, с которого закончился ввод/вывод
2. Анализируются байты состояния. Если всё хорошо, он посылает POST, после чего задача, поставившая запрос, узнает о том, что ввод/вывод завершен.
3. Супервизор запускает запросы на ввод/вывод для всех запросов, которые может запустить, после чего отдает управление диспетчеру.

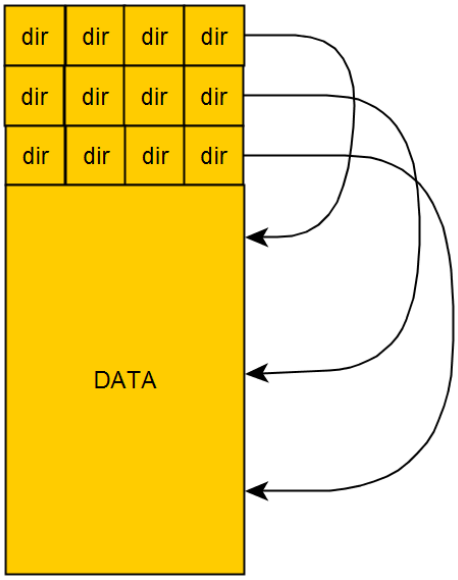
10.3. Логическая система ввода/вывода (z/OS)

Метод доступа - это сочетание способа доступа к данным с определённым типом организации данных.



10.4. Типы организации данных

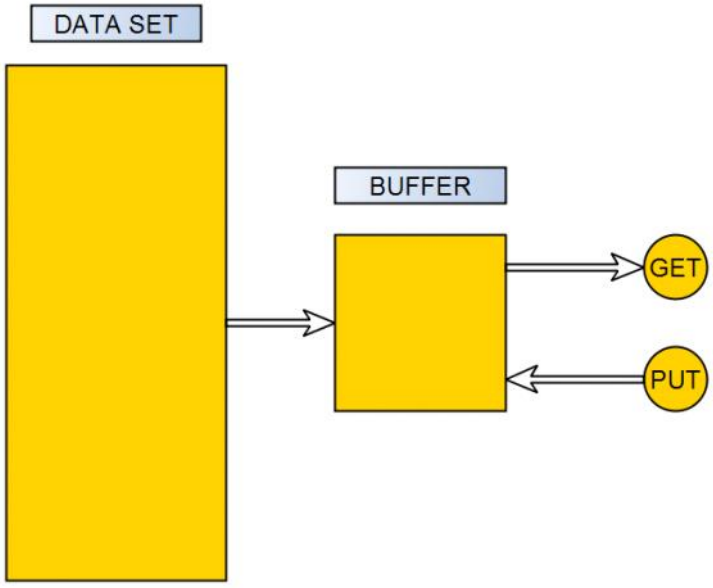
- **Последовательные (SQ)** - можно прочитать записи только по порядку. Адрес n-ой записи неизвестен.
- **Библиотечные (PDS)**
  Данные делятся на элементы, элементы хранятся в директориях.



  - PDSE (Extended)
  - VSAM

Способы доступа к наборам данных

1. **С очередями**
   Макрокоманды GET и PUT.
   Программа метода доступа управляет буферным пулом.
   Как только буфер заполнился, он автоматически загружается в набор данных.



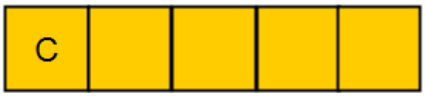
Применим только тогда, когда адрес следующей записи можно вычислить по предыдущему.
Эти макрокоманды делают все необходимые операции сами.

2. **Базисный**
   Макрокоманды READ и WRITE.
   По READ читается блок, все необходимые операции (разблокирование и т.п.) нужно сделать самому. WRITE пишет блок, поэтому блок нужно подготовить. В этом случае нет необходимости писать канальную программу, и этот способ применим к любым наборам данных.

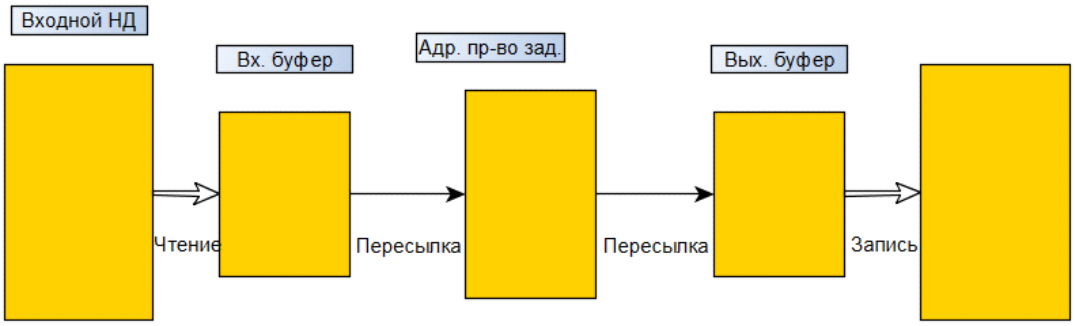
QSAM - последовательный метод доступа с очередями
BSAM - последовательный базисный
BPAM -

10.5. Буферизация

Буферный пул



В начале управляющий байт, затем - различные буферы
Используется потому, что основные затраты времени - на движение головки, а не на запись. Запись нескольких килобайт или байт практически не различается по времени. Поэтому логично делать запись большими блоками.



Виды буферизации

- **Буферизация с упреждением** - когда буферы заполняются ещё до первой команды READ, а при записи - буферы разгружаются по заполнению
- **Буферизация по требованию** - только по требованию

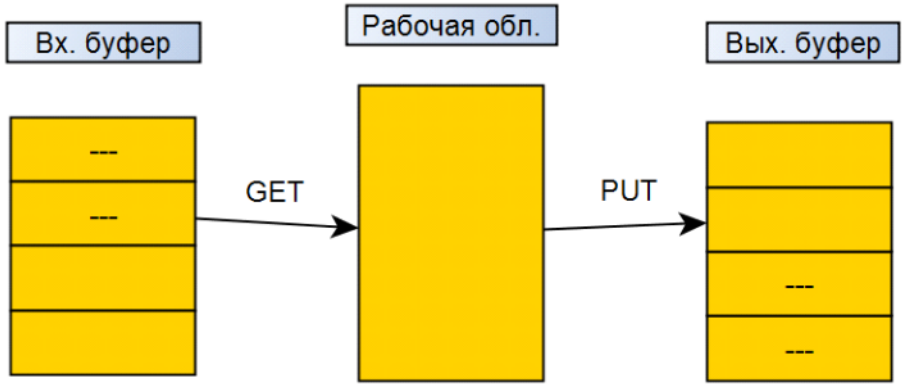
Пересылки

- **Режим перемещения** - содержимое буфера непосредственно перемещается в рабочую область или наоборот
- **Режим указания** - макрокоманда даст адрес заполненного буфера при GET и адрес свободного буфера при PUT
- **Режим подстановки (замещения)** - форматы и размеры входного и выходного буфера должны быть одинаковы. Полезно при "перегонке" файла из одного в другой.

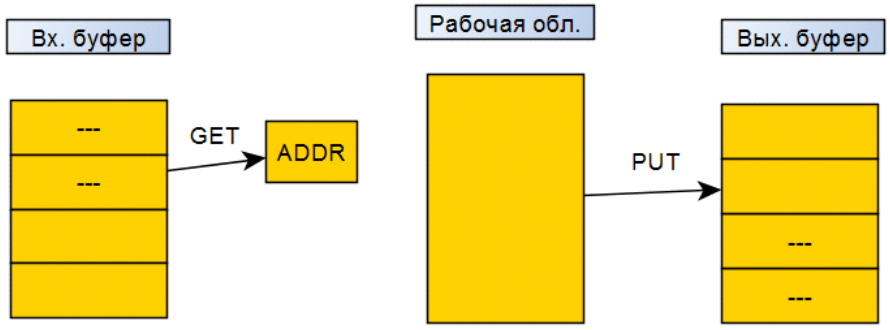
10.5.1. Простая буферизация

Режимы подстановки

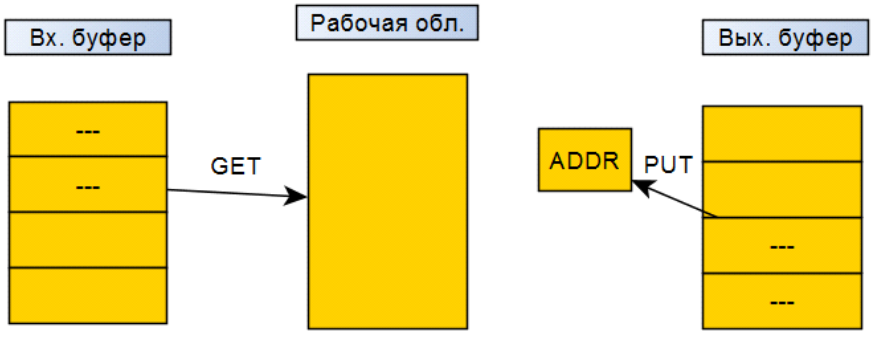
GET - перемещение, PUT - перемещение



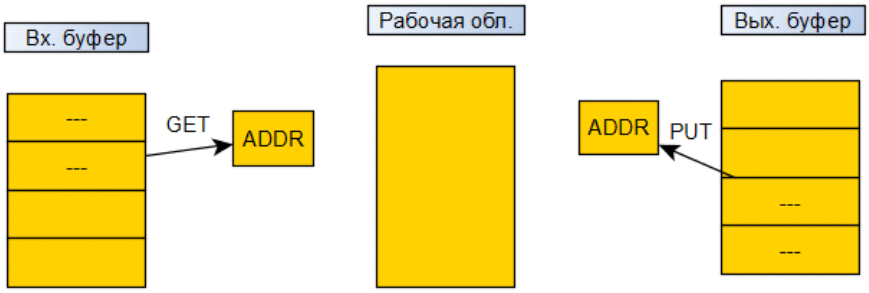
GET - указание, PUT - перемещение



GET - перемещение, PUT - указание

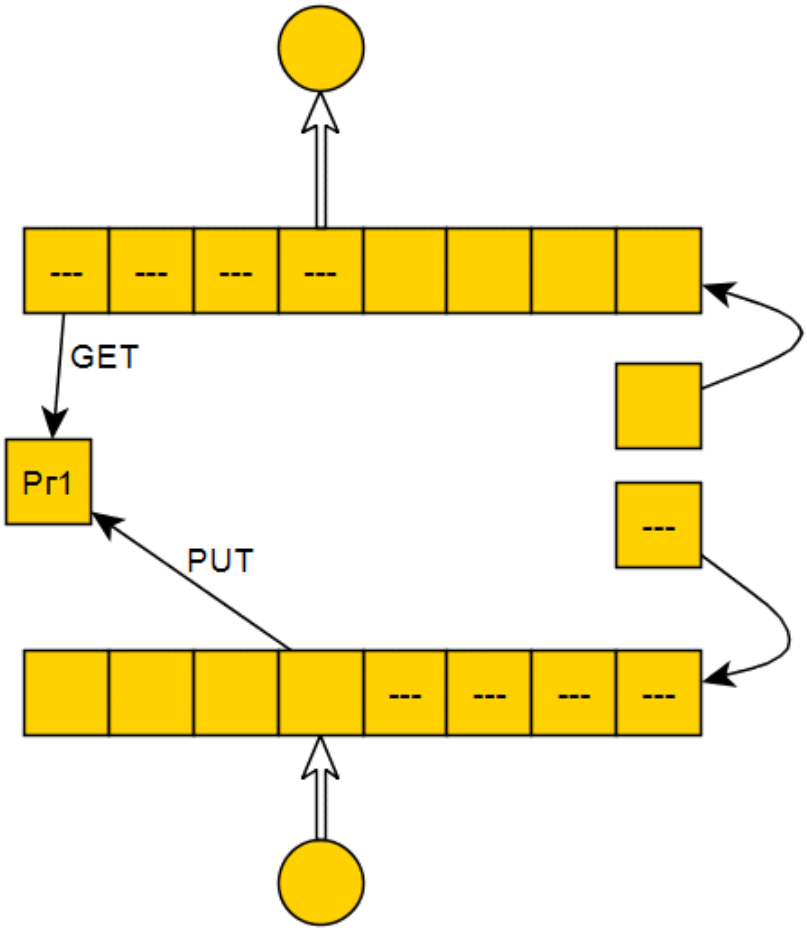


GET - указание, PUT - указание

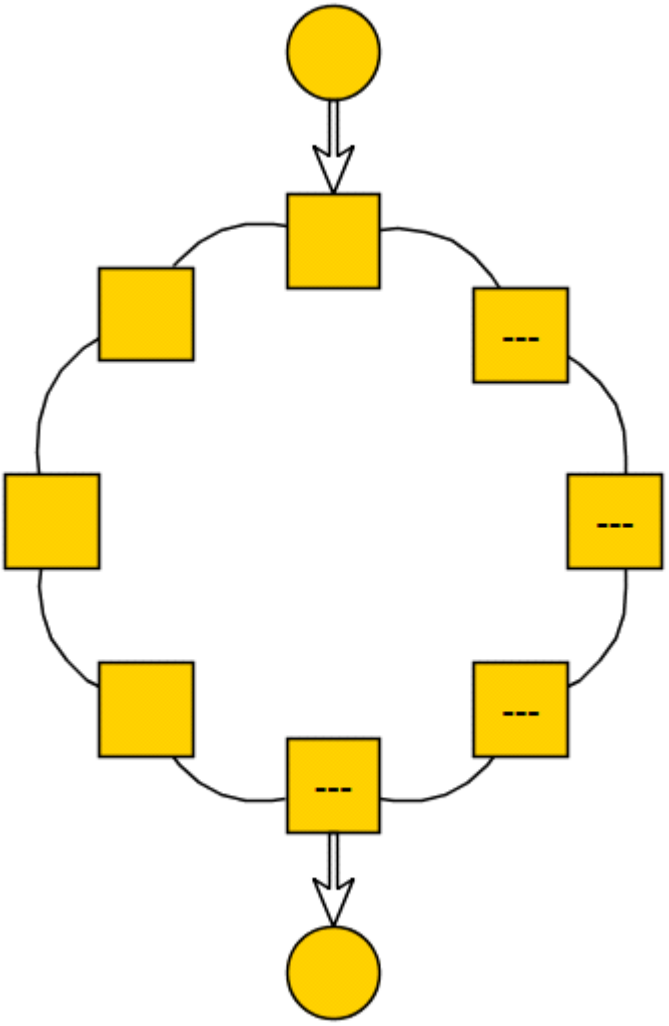


10.5.2. Обменная буферизация

GET и PUT работают в режиме подстановки, рабочая область не требуется.



GET
Pr1
PUT



10.6. Организация ввода/вывода в UNIX

В файловой системе UNIX есть каталог root/dev, в котором лежит описание всех драйверов. Все это считывается в ОП и держится в буфере, чтобы не итерироваться по файловой системе.

В этой структуре есть **адресный дескриптор** файла:

`i_addr[0]`

- `d_major` - определяет тип устройства и соответствующий драйвер
- `d_minor` - номер устройства, передающийся драйверу

Все устройства в UNIX делятся на 2 класса: **байт-ориентированные** и **блок-ориентированные** (запоминающие)

Адреса драйверов для байт-ориентированных устройств находится в таблице CdevSW, а для блок-ориентированных в BdevSW. По этим таблицам находится адрес нужного драйвера.

В контексте открытия или создания файла создается структура u:

`u`

- `u_offile` - указывает на дескриптор файла
- `u_count` - для байт-ориентированных устройств
- `u_offset` - номер следующего блока устройства (для блок-ориентированных устройств)

`u_base` - для байт-ориентированных устройств

В дескрипторе содержится:

- Тип открытия файла - чтение/запись
- Счётчик числа ссылок к дескриптору
- Указатель позиции в файле (меняется при чтении или записи)

12. Управление данными

Tuesday, May 15, 2018 13:08

Тема 12. Управление данными

С введением IBM360 единицей адресуемой памяти стал байт.
Запись - последовательность байт.
Физическая запись - представление записи на диске. Обычно в физическую запись пишется блок логических записей.
Логические записи

1. **Фиксированный** (F) тип. Каждая запись имеет одинаковый размер
2. **Запись переменной длины** (V)
   В z/OS идет сначала поле счётчика, потом данные, в Open System выбран символ, который является EOL.
3. **Неопределённая запись** (U)**.** Пишется всё подряд, иногда используется для загрузочных модулей

Блокирование записей
Для F-типа определяется размер блока, кратный размеру записи. Для V-типа в блоке обычно остается свободное место, т.к. неизвестно, насколько заполнится блок.
В Open System длина блока - 512 байт. Такими блоками запись идёт на диск. В z/OS это длину блока определяет программист

	Структурированность	Интерпретированность
Файл	Нет	Нет
Набор данных	Да	Нет
Базы данных	Да	Да

Управление данными на внешних запоминающих устройствах

Данными нужно управлять, т.к.:

1. Пользователи должны быть изолированы от машинно-зависимых объектов
2. Должно быть разграничение доступа к разным данным
3. Алгоритмы управления данными должны быть централизованы
4. Необходимо управлять системной информацией

Функции:

1. Учет местонахождения информации на носителе
В Z/OS у на диске есть VTOC - Volume Table of Contents. В нулевом цилиндре на нулевой дорожке есть указание на адрес VTOC.
Во VTOC'е находятся DSCB - Data Set Control Block (в UNIX это индексный файлы и дескрипторы файла)

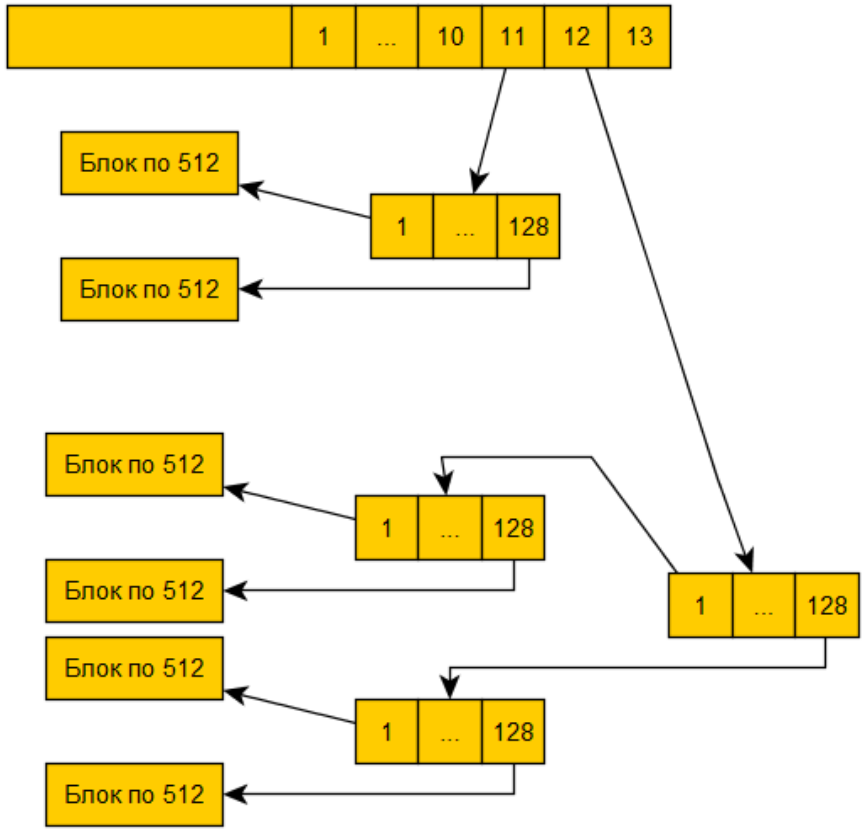
В UNIX 1-й блок - IPL - **Initial Program Load**, а второй - **суперблок**. В нём обозначена версия системы и прочая полезная информация.

Структура суперблока:

- s.size
- s.nfree - число свободных блоков в s.free
- s.free[NICFREE] - массив номеров свободных блоков
- s.nind - число индексных дескрипторов
- s.inode[NICINODE] - массив индексных дескрипторов
- s.flock - флаг, запрещающий работу с блоков
- s.ilock - запрещает работу с индексным дескриптором
- s.fmode - если данные модифицированы
- s.ronly

Индексный дескриптор файла

С 1-го по 10-й - ссылки на блоки по 512 байт



2. Размещение информации

- **Системная** информация
- **Общепользовательская** информация
- **Персональная** информация

3. Выделение информационного ресурса

Существует ряд таблиц, отвечающих за работу с ресурсом.

4. Освобождение информационного ресурса

Уничтожение этих таблиц

Работа с файлами

После команды `FILE * fp; if (fp = fopen(…)) …`

Fp будет указывать на FCB, хранящийся в адресном пространстве программы

Разграничение полномочий доступа

UIC = [gr, num] - код идентификации пользователя, получается при входе пользователя в систему.

Все пользователи разделены на группы:

1. **Системный пользователь (SYS) -** Обычно UIC для системного пользователя что-то вроде [255, 255]
2. **Член группы (GR)** - если группа совпадает с группой собственника
3. **Собственник файла (OWN)** - UIC совпадает с UIC собственника
4. **Кто угодно (WORLD)** - ничего не совпадает

В файле заданы атрибуты для каждой группы пользователя

	R	W	…
SYS	+		
GR	+	+	
OWN	+	+	+

Получение доступа к информационному ресурсу

Когда fopen пытается открыть файл, ему нужно получить индексный дескриптор этого файла. Но этот файл уже может быть открыт.

Поэтому в системной области указаны активные дескрипторы, и fopen по этим дескрипторам может проверить, открыт ли файл. Если файл открыт, нужно посмотреть, открыт файл на чтение или на запись.

- Если на запись - файл открыть нельзя
- Если на чтение - файл можно открыть на чтение

Если функция не нашла дескриптор, то файл не открыт, и функция идет в индексный файл и ищет дескриптор. Если дескриптор найдется, функция записывает дескриптор в активные и устанавливает нужные флаги, после чего вытаскивает информацию из дескриптора и заполняет FCB.

В FCB хранится указатель на текущее положение в файле - информация о позиции в файле принадлежит программе для возможности одновременного чтения.