

# Операционные системы. Экзамен.

Рязанова Наталья Юрьевна

2019

# Содержание

<b>1 билет</b>	<b>6</b>
1.1 ОС – определение ОС. Ресурсы вычислительной системы. Режимы ядра и задачи: переключение в режим ядра – классификация событий. Процесс, как единица декомпозиции системы, диаграмма состояний процесса с демонстрацией действий, выполняемых в режиме ядра. . . . .	6
1.2 Три режима работы компьютера на базе процессоров Intel (X86). Адресация аппаратных прерываний в защищенном режиме: таблица дескрипторов прерываний (IDT) – формат дескриптора прерываний, типы шлюзов. Пример заполнения IDT из лабораторной работы. . . . .	7
<b>2 билет</b>	<b>10</b>
2.1 Классификация операционных систем. Особенности ОС определенных типов. Виртуальная машина и иерархическая машина – декомпозиция системы на уровня иерархии, иерархическая структура Unix BSD, архитектуры ядер ОС – определение, примеры. . . . .	10
2.2 Три режима работы вычислительной системы с архитектурой X86: особенности. Реальный режим: линия A20 – адресное заворачивание. Перевод компьютера в защищенный режим. Линия A20 в защищенном режиме: включение и выключение линии A20 (код из лабораторной работы). . . . .	11
<b>3 билет</b>	<b>14</b>
3.1 Прерывания: классификация. Последовательность действий при выполнении запроса ввода-вывода. Обработчики аппаратных прерываний: виды и особенности. Функции обработчика прерываний от системного таймера. . . . .	14
3.2 Защищенный режим: назначение системных таблиц – глобальной таблицы дескрипторов (GDT), таблицы дескрипторов прерываний (IDT), теневых регистров (структуры, описывающие дескрипторы GDT и IDT и заполнение дескрипторов в лабораторной работе по защищенному режиму). . . . .	17
<b>4 билет</b>	<b>19</b>
4.1 Тупики: Обнаружение тупиков для повторно используемых ресурсов методом редукции графа, способы представления графа, алгоритмы обнаружения тупиков. Пример анализа состояния системы метод редукции графа. Методы восстановления работоспособности системы. . . . .	19
4.2 Задача «Обедающие философы» – модели распределение ресурсов вычислительной системы. Множественные семафоры UNIX: системные вызовы, поддержка в системе, пример использования из лабораторной работы «производство-потребление». . . . .	20
<b>5 билет</b>	<b>25</b>
5.1 Виртуальная память: распределение памяти страницами по запросам, схема с гиперстраницами, обоснование использования данной схемы. Управление памятью страницами по запросам в архитектурах x86 – расширенное преобразование (PAE) – схема преобразований. Анализ страничного поведения процессов: свойство локальности, рабочее множество. . . . .	25
5.2 Задача «Производство-потребление»: алгоритм Эд. Дейкстры, реализация на семафорах UNIX (код из лабораторной работы). . . . .	27
<b>6 билет</b>	<b>31</b>
6.1 Понятие процесса. Процесс как единица декомпозиции системы. Диаграмма состояний процесса с демонстрацией действий, выполняемых в режиме ядра. Планирование и диспетчеризация. Классификация алгоритмов планирования. Примеры алгоритмов планирования, соотнесенные с типами ОС. Процессы и потоки. Типы потоков. . . . .	31
6.2 Обеспечение монопольного доступа к разделяемым данным в задаче «читатели-писатели» : реализация на базе Win32 API (пример кодов лабораторной работы «читатели-писатели» для ОС Windows). . . . .	35
<b>7 билет</b>	<b>38</b>
7.1 Управление виртуальной памятью: распределение памяти сегментами по запросам: схема преобразования виртуального адреса, способы организации таблиц сегментов, стратегии выбора разделов памяти для загрузки сегментов, алгоритмы и особенности замещения сегментов. . . . .	38
7.2 Управление памятью сегментами по запросам в архитектуре X86. Тип организации таблиц сегментов. Формат дескриптора сегмента в таблицах дескрипторов сегментов (GDT и LDT) (код и заполнение дескрипторов GDT из лабораторной работы по защищенному режиму). . . . .	40
<b>8 билет</b>	<b>43</b>

8.1	Взаимоисключение и синхронизация процессов и потоков. Семафоры: определение, виды. Семафор, как средство синхронизации и передачи сообщений. Семафоры UNIX: примеры решения задач с помощью семафоров: «Производство-потребление» и «Читатели-писатели» в UNIX (пример реализации в лабораторной работе). . . . .	43
8.2	Аппаратные прерывания: задачи обработчика прерываний от системного таймера в защищенном режиме. . . . .	48
<b>9</b>	<b>билет</b>	<b>50</b>
9.1	Виртуальная память: управление памятью страницами по запросу – три схемы преобразования; реализация страничного преобразования в компьютерах на базе процессоров Intel (x86): стандартное преобразование и PAE в защищенном режиме – схемы, размеры таблиц и их количество на каждом этапе преобразования. . . . .	50
9.2	Unix: концепция процессов; иерархия процессов, процессы «сироты», процессы «зомби», демоны; примеры из лабораторной работы (5 программ). . . . .	52
<b>10</b>	<b>билет</b>	<b>55</b>
<b>11</b>	<b>билет</b>	<b>56</b>
11.1	Параллельные процессы: взаимодействие, обоснование необходимости монопольного доступа к разделяемым переменным, способы взаимоисключения. Мониторы: определение; примеры – простой монитор и монитор кольцевой буфер. . . . .	56
11.2	Средства межпроцессорного взаимодействия (IPC) операционной системы UNIX System V: очереди сообщений и программные каналы – сравнение, примеры (для программных каналов пример из лабораторной работы с сигналами). . . . .	57
<b>12</b>	<b>билет</b>	<b>62</b>
12.1	ОС с монолитным ядром. Переключение в режим ядра. Диаграмма состояний процесса и переход из одного состояния в другое – причины каждого перехода. Диаграмма состояний процесса в UNIX. Переключение контекста. Система прерываний. . . . .	62
12.2	Задача: читатели-писатели – монитор Хоара, решение с использованием семафоров Unix и разделяемой памяти, пример реализации из лабораторной работы. . . . .	64
<b>13</b>	<b>билет</b>	<b>67</b>
13.1	Виртуальная память: управление памятью страницами по запросу – три схемы. Алгоритмы вытеснения страниц: демонстрация особенностей на модели траектории страниц. Рабочее множество – определение, глобальное и локальное замещение. Флаги в дескрипторах страниц, предназначенные для реализации замещения страниц. . . . .	67
13.2	Синхронизация и взаимоисключение параллельных процессов в распределенных системах: централизованный и распределенный алгоритмы, алгоритмы Token-ring; сравнение алгоритмов. Транзакции: определение, особенности, двухфазный протокол фиксации. . . . .	69
<b>14</b>	<b>билет</b>	<b>73</b>
14.1	Процессы: взаимодействие процессов в распределенных системах; централизованный и распределенный алгоритмы, синхронизация часов (алгоритм Лампорта); RPC – механизм . . . . .	73
14.2	Аппаратные прерывания: типы аппаратных прерываний; особенности. Прерывания от устройств ввода-вывода: назначение и аппаратная реализация. Прерывание от системного таймера в защищенном режиме. Пример кода обработчика прерывания от системного таймера из лабораторной работы по защищенному режиму. . . . .	75
<b>15</b>	<b>билет</b>	<b>78</b>
15.1	Межпроцессорное взаимодействие в Unix System V (IPC): сигналы, программные каналы, семафоры и разделяемая память; примеры использования из лабораторных работ. . . . .	78
15.2	Синхронизация и взаимоисключение параллельных процессов в распределенных системах: централизованный и распределенные алгоритмы – сравнение. . . . .	83
<b>16</b>	<b>билет</b>	<b>85</b>
16.1	Взаимодействие параллельных процессов: проблемы; монопольный доступ и взаимоисключение; взаимодействие параллельных процессов в распределенных системах – особенности; централизованный алгоритм, распределенный алгоритм; синхронизация логических часов (алгоритм Лампорта). . . . .	85
16.2	Процессы в UNIX: системные вызовы fork(), exec(), wait(), signal() – примеры из лабораторных работ. . . . .	86

<b>17 билет</b>	<b>93</b>
17.1 Виртуальная память: распределение памяти страницами по запросам, свойство локальности, рабочее множество, анализ страничного поведения процессов. Схема страничного преобразования в процессорах Intel (X86) PAE – размеры таблиц дескрипторов. . . . .	93
17.2 Прерывания от системного таймера в защищенном режиме: функции (по материалам лабораторной работы). . . . .	95
<b>18 билет</b>	<b>96</b>
18.1 Процессы: взаимодействие параллельных процессов – монопольный доступ и взаимоисключение; программная реализация взаимоисключения – флаги, алгоритм Деккера, алгоритм Лампорта. . . . .	96
18.2 Защищенный режим: перевод компьютера в защищенный режим – реализация – пример кода из лабораторной работы. . . . .	98
<b>19 билет</b>	<b>102</b>
19.1 Процессы Unix: создание процесса в ОС Unix и запуск новой программы. Примеры программ из лабораторных работ, демонстрирующих эти действия. Системные вызовы wait() и pipe(): назначение, примеры из лабораторных работ. Процессы «сироты», «зомби» и «демоны». . . . .	102
19.2 Взаимодействие параллельных процессов: мониторы – определение; монитор Хоара «читатели-писатели», реализация в для ОС Windows – пример из лабораторной работы. . . . .	107
<b>20 билет</b>	<b>112</b>
20.1 Процессы: взаимодействие параллельных процессов – монопольный доступ и взаимоисключение; программная реализация взаимоисключения – примеры, семафоры – определение, виды семафоров, примеры использования множественных семафоров из лабораторных работ «производство-потребление» и «читатели-писатели». . . . .	112
20.2 Приоритетное планирование в ОС Windows (лабораторная работа). . . . .	118
<b>21 билет</b>	<b>121</b>
21.1 Взаимодействие параллельных процессов: монопольное использование – реализация; типы реализации взаимоисключения. Мониторы – определение, примеры: простой монитор, монитор «кольцевой буфер» и монитор «читатели-писатели». Пример реализации монитора Хоара «читатели-писатели» для ОС Windows. . . . .	121
21.2 Процессы Unix: создание процесса в ОС Unix и запуск новой программы. Примеры из лабораторной работы (код). . . . .	126
<b>22 билет</b>	<b>129</b>
22.1 Процессы: взаимодействие параллельных процессов – монопольный доступ и взаимоисключение; аппаратная реализация взаимоисключения, спин-блокировка – реализация. . . . .	129
22.2 Процессы: бесконечное откладывание, зависание, тупиковая ситуация – анализ на примере задачи об обедающих философах и примеры аналогичных ситуаций в ОС. Множественные семафоры в Linux: системные вызовы и поддержка в ОС Linux; примеры из лабораторных работ. . . . .	130
<b>23 билет</b>	<b>133</b>
<b>24 билет</b>	<b>134</b>
24.1 Процессы: взаимодействие параллельных процессов – монопольный доступ и взаимоисключение; алгоритм Лампорта «Булочная» и «Логические часы» Лампорта. . . . .	134
24.2 Процессы: бесконечное откладывание, зависание, тупиковая ситуация – анализ на примере задачи об обедающих философах и примеры аналогичных ситуаций в ОС. Множественные семафоры в Linux: системные вызовы и поддержка в системе; пример из лабораторной работы «производство-потребление». . . . .	135
<b>25 билет</b>	<b>140</b>
<b>26 Дополнительные вопросы</b>	<b>141</b>
26.1 XMS . . . . .	141
26.2 Методы организации ввода-вывода: программируемый, с прерываниями, прямой доступ к памяти. . . . .	141
26.3 Кэши TLB и данных. . . . .	141
26.4 ОС с монолит. ядром. Переключение в режим ядра. Система прерываний. Точные и неточные прерывания. . . . .	141
26.5 Спецификация XM ( XMS ): Conventional, HMA, UMA, EMA. . . . .	141
26.6 Управление памятью: выделение памяти разделами фиксированного размера, выделение памяти разделами переменного размера, стратегии выделения памяти, фрагментация памяти. . . . .	142

26.7 Тупики: Обнаружение тупиков для повторно используемых ресурсов методом редукции графа, способы представления графа и методы восстановление работоспособности системы. . . . .	142
26.8 Последовательность операций при выполнении аппаратного прерывания. Прерывания точные и неточные . . . . .	145
26.9 EMS . . . . .	146
26.10 Классификация структур ядер ОС. Особенности ОС с микроядром. Модель клиент-сервер. Три состояния процесса при передаче сообщений. Достоинства и недостатки микро-ядерной архитектуры.	146
26.11 Тупики: определение тупиковой ситуации для повторно используемых ресурсов, четыре условия возникновения тупика, обход тупиков - алгоритм банкира. . . . .	149
26.12 Прерывание int 8h (реальный режим) - функции. . . . .	150

# 1 билет

**1.1 ОС – определение ОС. Ресурсы вычислительной системы. Режимы ядра и задачи: переключение в режим ядра – классификация событий. Процесс, как единица декомпозиции системы, диаграмма состояний процесса с демонстрацией действий, выполняемых в режиме ядра.**

**ОС** – это комплект программ, которые совместно управляют ресурсами вычислительной системы и процессами, использующими эти ресурсы при вычислениях.

**Ресурс** – это любой из компонентов вычислительной системы и предоставляемые ею возможности.

Основной задачей ОС – выделение процессам ресурсов.

## 1.1.1 Главные ресурсы программы

1. Процессорное время
2. Объем памяти
3. Ключи защиты
4. Системные таблицы
5. Реентерабельные коды – переход чистой процедуры (которая не изменяет саму себя)

**Процесс** – программа в стадии выполнения.

**Процесс** – главная абстракция системы. UNIX декларирует следующим образом: процесс – часть времени выполняет собственный код, и тогда он выполняется в **режиме задачи**, а часть времени выполняет реентерабельный код операционной системы, и тогда он выполняется в **режиме ядра**.

## 1.1.2 Три события, переводящих систему в режим ядра

1. Системные вызовы (программные прерывания). Процесс переключается в режим ядра и выполняется реентерабельный код системы, после обслуживания переключается в режим задачи.

Ни одна операционная система не позволяет процессу на прямую обратиться к устройствам ввода/вывода. Если бы обращались напрямую, то такая система была бы незащищенной. Для обслуживания системного вызова, процесс должен перейти в режим ядра, в режиме ядра выполняется реентерабельный код ОС.

Реентерабельный код – код чистой процедуры (чистая процедура не модифицирует саму себя (вынесены все данные)). Коды ядра работают с системными таблицами. Процедуры ядра сами являются ресурсами, которые могут запрашивать процессы, разные процессы могут находиться в разных точках одной и той же процедуры.

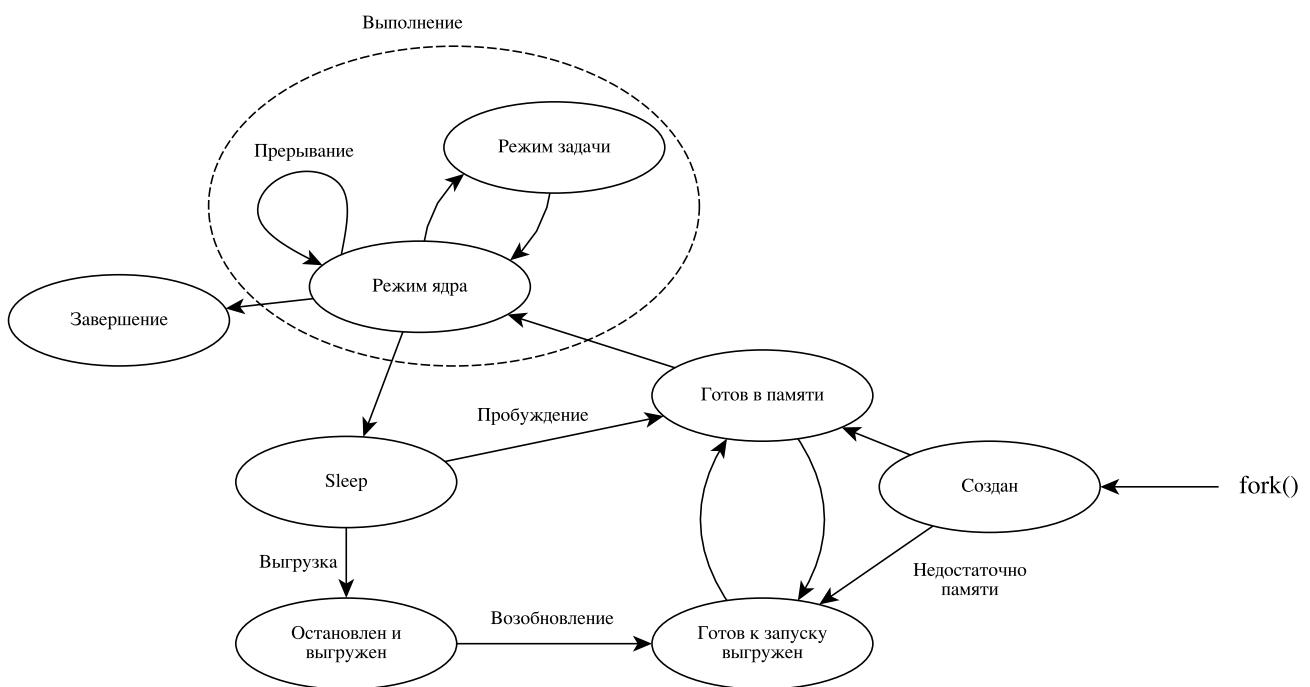
### 2. Исключения (exception)

Системные вызовы и исключения – синхронные события по отношению к вашему процессу. Возникают в процессе выполнения программного кода. Исключения возникают в результате ошибок в программме (не устранимым, например – деление на ноль). Устранимое – страничное прерывание, необходимая страница будет загружена в память и программа продолжит выполнение.

### 3. Аппаратные прерывания (прерывания, поступившие от устройств)

Асинхронные события в системе. Также имеют различный характер в системе. Самое массовое – прерывания от внешних устройств. Второй тип – прерывание от системного таймера, которое выполняется по тику (выделенный импульс) 18,3 раз в секунду. Третий тип – прерывание от действий оператора (`ctrl+alt+del`, `ctrl+C` = завершение процесса в unix)

### 1.1.3 Диаграмма состояний процесса



## 1.2 Три режима работы компьютера на базе процессоров Intel (X86). Адресация аппаратных прерываний в защищенном режиме: таблица дескрипторов прерываний (IDT) – формат дескриптора прерываний, типы шлюзов. Пример заполнения IDT из лабораторной работы.

### 1.2.1 Три режима работы компьютера

Компьютеры на базе процессоров Intel X86 могут работать в 3 режимах:

#### 1. Реальном

Реальный режим поддерживается аппаратно, может работать аналогично процессорам архитектуры x86 (16 разрядов, с 20-разрядный адрес (шина данных) – сегмент/смещение), используется сегментная адресация памяти.

Минимальная адресная единица памяти – байт.  $2^{20} = FFFF = 1024 \text{ Кб} = 1\text{Мб}$  (объем доступного адресного пространства).

Необходим для обеспечения функционирования программ, разработанных для старых моделей, в новых моделях микропроцессоров.

Однопроцессорный режим под управлением MS-DOS (нет многозадачности).

Компьютер начинает работать в реальном режиме.

Максимально возможный размер сегмента в реальном режиме - 64КБ.

## 2. Защищенном

Защищенный режим – 32х разрядные многопроцессный/многозадачный режим, поддерживающий виртуальную память. В защищенном режиме 4 уровня привилегий, ядро ОС – на 0-м. Используется страничная организация памяти (повышает уровень защиты задач друг от друга и эффективность их выполнения). Объем адресуемой памяти – 4 Гб.

### 3. Специальном режиме защищенного режима V86(virtual)

Специальный режим защищенного режима – многозадачный режим, в котором как задачи выполняются ОС реального режима (таких виртуальных машин может быть несколько).

У каждой задачи собственное адресное пространство размером 1МБ.

Процессор фактически продолжает использовать схему преобразования адресов памяти и средства мультизадачности защищённого режима.

В виртуальном режиме используется трансляция страниц памяти (в эту область могут быть отображены произвольные страницы памяти). Это позволяет в мультизадачной операционной системе создавать несколько задач, работающих в виртуальном режиме.

Задача исполняется с самыми низкими привилегиями в кольце 3. Когда в такой задаче возникает прерывание, процессор переключается из виртуального режима в защищённый. Поэтому все прерывания отображаются в операционную систему,ющую в защищённом режиме.

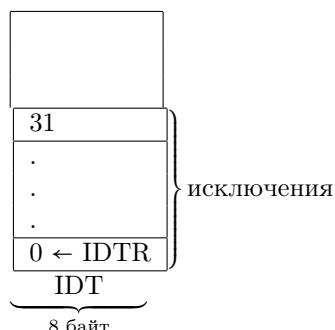
Прерывания обрабатываются обычными обработчиками ОС защищенного режима.

### 1.2.2 Адресация аппаратных прерываний

В защищенном режиме адресация обработчиков выполняется с помощью таблицы дескрипторов прерываний. IDT содержит дескрипторы трех типов (gate).

- **Interrupt Gate** – аппаратные прерывания
  - **Task Gate** – программные прерывания, системные вызовы
  - **Trap Gate** – исключения

В процессоре есть регистр IDTR, который содержит начальный адрес в таблице дескрипторов прерываний.



Селектор – сегментный регистр в защищенном режиме. Для обращения к дескрипторам. С помощью селектора получаем доступ к сегменту кода, описанному в GDT. Из дескриптора сегмента кода, берем начальный адрес сегмента.

### 1.2.3 Пример заполнения ИДТ из лабораторной работы

```
intr struc ; Структура для описания дескрипторов прерываний
    offs_l    dw 0      ; Смещение обработчика, нижняя часть (биты 0..15)
    sel        dw 0      ; Селектор сегмента команд
    rsrv       db 0      ; Зарезервировано
    attr       db 0      ; Атрибуты
    offs_h    dw 0      ; Смещение обработчика, верхняя часть (биты 16..31)
intr ends
```

```
IDT label byte ; Таблица дескрипторов прерываний IDT
trap1 intr 13 dup (<0, SEL_32bitCS,0, 8Fh, 0>) ; Первые 32 элемента таблицы
; (отведены под исключения)
```

```
trap13 intr <0, SEL_32bitCS,0, 8Fh, 0> ; Исключение общей защиты
trap2 intr 18 dup (<0, SEL_32bitCS,0, 8Fh, 0>) ; Первые 32 элемента таблицы
                                                ; (отведены под исключений)
int08 intr <0, SEL_32bitCS,0, 8Eh, 0> ; Дескриптор прерывания от таймера
int09 intr <0, SEL_32bitCS,0, 8Eh, 0> ; Дескриптор прерывания от клавиатуры
idt_size = $-IDT ; Размер IDT

idtr dw idt_size-1 ; Лимит IDT
dd ? ; Линейный адрес IDT

idtr_real dw 3FFh,0,0 ; содержимое регистра IDTR в реальном режиме
```

## 2 билет

- 2.1 Классификация операционных систем. Особенности ОС определенных типов.**  
Виртуальная машина и иерархическая машина – декомпозиция системы на уровня иерархии, иерархическая структура Unix BSD, архитектуры ядер ОС – определение, примеры.

### Классификация операционных систем.

#### 1. Однопрограммная пакетной обработки

В оперативной памяти может быть только 1 прикладная программа.

#### 2. Мультипрограммной пакетной обработки

В оперативной памяти одновременно много программ. Загрузка – перфокартами. Программа располагается в памяти целиком; процессорное время выделяется по принципу приоритетов.

#### 3. Системы разделения времени

В оперативной памяти одновременно находится большое число программ, процессорное время квантуется, чтобы обеспечить гарантированное время ответа системы. Время ответа системы  $\leq 3$  секунд.

#### 4. Системы реального времени

Реальное время в операционных системах – это способность операционной системы обеспечить требуемый уровень сервиса в определенный промежуток времен, величина которого определяется особенностями работы внешнего по отношению к вычислительной системе устройства или процесса.

В отличие от систем общего назначения ОСРВ обеспечивает ответ системы (или сервис системы) за определенный промежуток времени, то есть обслуживание запроса. Это всегда запрос или внешний по отношению к системе или внешнего процесса.

2 процесса реального времени в наших компьютерах – видео и аудио.

Жесткое реальное время – это когда интервал установлен жестко и не может быть превышен.

Мягкое реальное время – возможность небольших отклонений от величины интервала.

#### 5. Персональные операционные системы

### Виртуальная машина – кажущаяся, возможная.

Виртуальная машина – набор команд и функций, необходимых пользователю для получения сервиса операционной системы

**Иерархическая машина (МЕДНИКА-ДОНОВАНА)** – ОС разбивается на функции и определяется место этих функций по удаленности от аппаратной части.



Управление процессами (нижний уровень) – выделение кванта времени.

Управление процессами (верхний уровень) – создание, уничтожение, взаимодействие при помощи сообщений.

Управление информацией – хранение, уничтожение файлов, файловая система.

**Интерфейс – функции, которые нижние уровни предоставляют верхним уровням.**

- прозрачные - позволяет обращаться через уровни
- полупрозрачные - какие-то обращения через уровни возможны (какие-то только к низлежащим уровням)
- непрозрачные - обращения возможны только к низлежащим уровням

Таблица 1: Структура ядра ОС UNIX 4.4 BSD

Системные вызовы					Сис. выз.		
Управление терминалом	1	2	3	4	Сокеты	Обработка сигналов	Создание и завершение процесса
Необратимый телетайп	Обработанный телетайп	Файловая система	Виртуальная память	Сетевые протоколы	Маршрутизация	Планирование процессов	
Драйверы символьных устройств	Дисциплины линий связи	Буферный КЭШ	Страницочный КЭШ	Драйверы блочных устройств	Драйверы сетевых устройств	Диспетчеризация процессов	

- 1- Символьный уровень
  - 2- Именование файлов
  - 3- Отображение адреса
  - 4- Страницочные прерывания
- ///- Аппаратные и эмулируемые прерывания

### Архитектура ядер операционных систем

Существует два типа ядер:

1. Монолитные ядра
2. Микроядра

#### Монолитное ядро

Это программа имеющая модульную структуру, то есть состоящая из подпрограмм. Системы Windows, Unix, Linux имеют монолитные ядра. Unix имеет минимизированное ядро (вынесен графический интерфейс).

#### Микроядро

В микроядерной архитектуре все компоненты являются самостоятельными программами, которые выполняются в собственных адресных пространствах. В силу этого взаимодействие между компонентами ОС выполняется с помощью посылки и приема сообщений, причем механизм посылки и приема сообщений обеспечивается специальным модулем ядра, который называется микроядро. Mach – первая ОС с микроядром. Классическим примером микроядерной ОС является Symbian OS.

## 2.2 Три режима работы вычислительной системы с архитектурой X86: особенности. Реальный режим: линия A20 – адресное заворачивание. Перевод компьютера в защищенный режим. Линия A20 в защищенном режиме: включение и выключение линии A20 (код из лабораторной работы).

Компьютеры на базе процессоров Intel X86 могут работать в 3 режимах:

1. Реальном

Реальный режим поддерживается аппаратно, может работать аналогично процессорам архитектуры x86 (16 разрядов, с 20-разрядный адрес (шина данных) – сегмент/смещение), используется сегментная адресация памяти.

Минимальная адресная единица памяти – байт.  $2^{20} = FFFF = 1024 \text{ Кб} = 1\text{Мб}$  (объем доступного адресного пространства).

Необходим для обеспечения функционирования программ, разработанных для старых моделей, в новых моделях микропроцессоров.

Однопроцессорный режим под управлением MS-DOS (нет многозадачности).

Компьютер начинает работать в реальном режиме.

Максимально возможный размер сегмента в реальном режиме - 64КБ.

## 2. Защищенным

Защищенный режим – 32х разрядные многопроцессорный/многозадачный режим, поддерживающий виртуальную память. В защищенном режиме 4 уровня привилегий, ядро ОС – на 0-м. Используется страничная организация памяти (повышает уровень защиты задач друг от друга и эффективность их выполнения). Объем адресуемой памяти – 4 Гб.

## 3. Специальном режиме защищенного режима V86(virtual)

Специальный режим защищенного режима – многозадачный режим, в котором как задачи выполняются ОС реального режима (таких виртуальных машин может быть несколько).

У каждой задачи собственное адресное пространство размером 1МБ.

Процессор фактически продолжает использовать схему преобразования адресов памяти и средства мультизадачности защищённого режима.

В виртуальном режиме используется трансляция страниц памяти (в эту область могут быть отображены произвольные страницы памяти). Это позволяет в мультизадачной операционной системе создавать несколько задач, работающих в виртуальном режиме.

Задача исполняется с самыми низкими привилегиями в кольце 3. Когда в такой задаче возникает прерывание, процессор переключается из виртуального режима в защищённый. Поэтому все прерывания отображаются в операционную систему,рабатывающую в защищённом режиме.

Прерывания обрабатываются обычными обработчиками ОС защищенного режима.

При включении компьютера, он находится в реальном режиме. **Порт линии A20** заземлен.

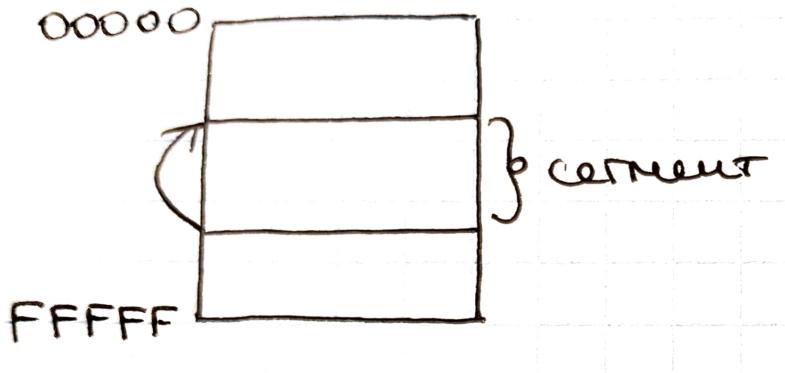
### Адресное заворачивание

В реальном режиме 20 адресных линий нам доступно  $2^{20} = 1 \text{ Мб}$ .

16-разрядные регистры. Смещение не превышает 64 Кб.

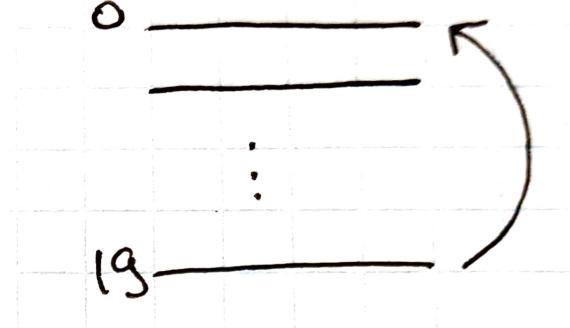
#### 1 тип

Тогда FFFF+1=0000, начинает указывать на начало сегмента.

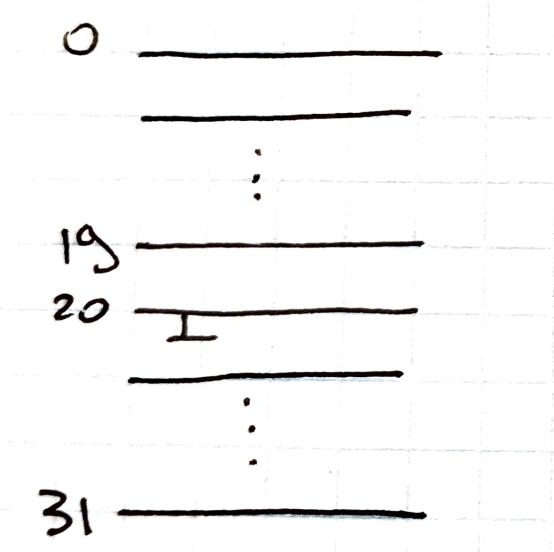


#### 2 тип

FFFFF+1=00000. Адрес начинает указывать на начало.



В наших компьютерах 32 адресных линии. В реальном режиме при превышении 1 Мб значение не теряется.



Если в реальном режиме открыть линию A20 (address line), то станет доступным еще 64Кб памяти.

Если мы переходим в защищенный режим (линия A20 принудительно обнулена). Для того чтобы были доступны все адреса надо открыть A20. Если не открыть, то будет битая память.

#### Переключение в защищенный режим

1. Открыть адресную линию A20.
2. Подготовить в оперативной памяти глобальную таблицу дескрипторов GDT. В этой таблице должны быть созданы дескрипторы для всех сегментов, которые будут нужны программе сразу после того, как она переключится в защищенный режим. Впоследствии, находясь в защищённом режиме, программа может модифицировать GDT (если, она в нулевом кольце защиты).
3. Подготовить в оперативной памяти таблицу дескрипторов прерываний IDT.
4. Для обеспечения возможности возврата из защищённого режима в реальный необходимо записать адрес возврата в реальный режим в область данных BIOS по определенному адресу, а также записать в CMOS память в ячейку 0Fh код. Этот код обеспечит после выполнения сброса процессора передачу управления по адресу, подготовленному нами в области данных BIOS по этому адресу.
5. Запомнить в оперативной памяти содержимое сегментных регистров, которые необходимо сохранить для возврата в реальный режим, в частности, указатель стека реального режима.
6. Запретить все маскируемые и немаскируемые прерывания. Сохранить маски прерываний. Перепрограммировать контроллер прерываний.
7. Загрузить регистр GDTR и IDTR.
8. Перейти в защищенный режим (установить бит PE — нулевой бит в управляющем регистре CR0 в 1).
9. Загрузить новый селектор в регистр CS.
10. Загрузить сегментные регистры селекторами на соответствующие дескрипторы.
11. Разрешить прерывания.

#### Открыть A20

```
mov al,0D1h
out 64h,al
mov al,0DFh
out 60h,al
```

или

```
in al,92h
or al,2
out 92h,al
```

#### Закрыть A20

```
mov al,0D1h
out 64h,al
mov al,0DDh
out 60h,al
```

### 3 билет

**3.1 Прерывания: классификация.** Последовательность действий при выполнении запроса ввода-вывода. Обработчики аппаратных прерываний: виды и особенности. Функции обработчика прерываний от системного таймера.

#### 3.1.1 3 типа прерываний (составляют систему прерываний):

##### 1. Системные вызовы (программные прерывания)

- вызов, когда требуется сервис системы (ввод/вывод, обращение к внешнему устройству)
- клавиатура, мышь, вторичная память (флешки, диски)
- ни одна система не позволяет напрямую процессом обращение к устройствам ввода-вывода, если разрешить такие обращения, то систему защитить невозможно.
- Система предоставляет соответствующие функции (примитивы, т. к. низкоуровневые (ядро) действия) (API)
- синхронные события.

##### 2. Исключительные ситуации

- прерывание выполняемой программы при возникновении исключения
- (исправимые – страничное прерывание; неисправимое – деление на 0)
- синхронные события.

##### 3. Аппаратные прерывания

- Бывают нескольких типов:
  - Прерывания системного таймера (важнейшие функции)
  - Прерывание от внешних устройств (возникают по завершению операции ввода/вывода)
  - Действия оператора (ctrl-alt-del)
- Асинхронные события, возникают независимо от каких-либо действий, которые выполняются в системе.

#### 3.1.2 Ввод-вывод

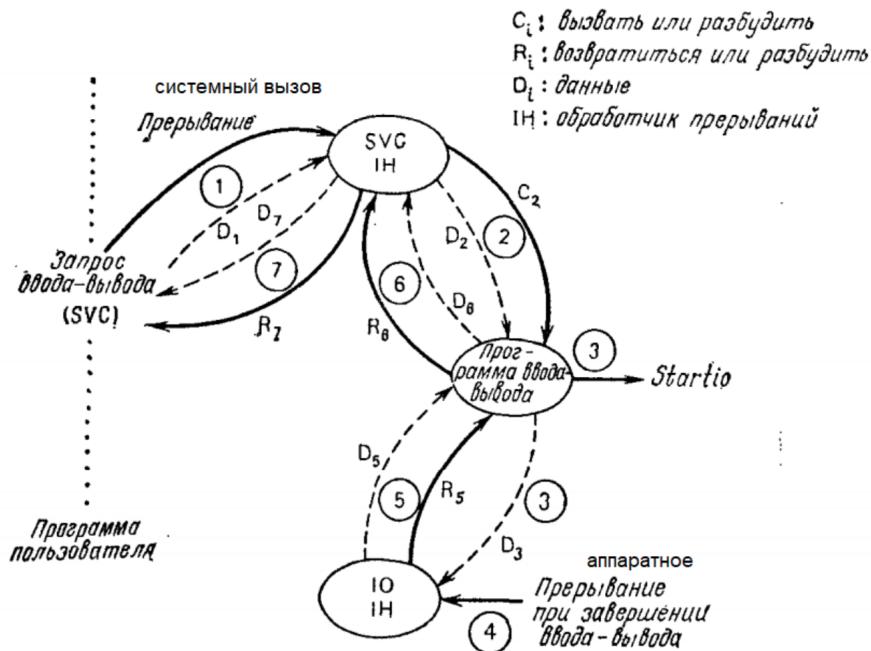
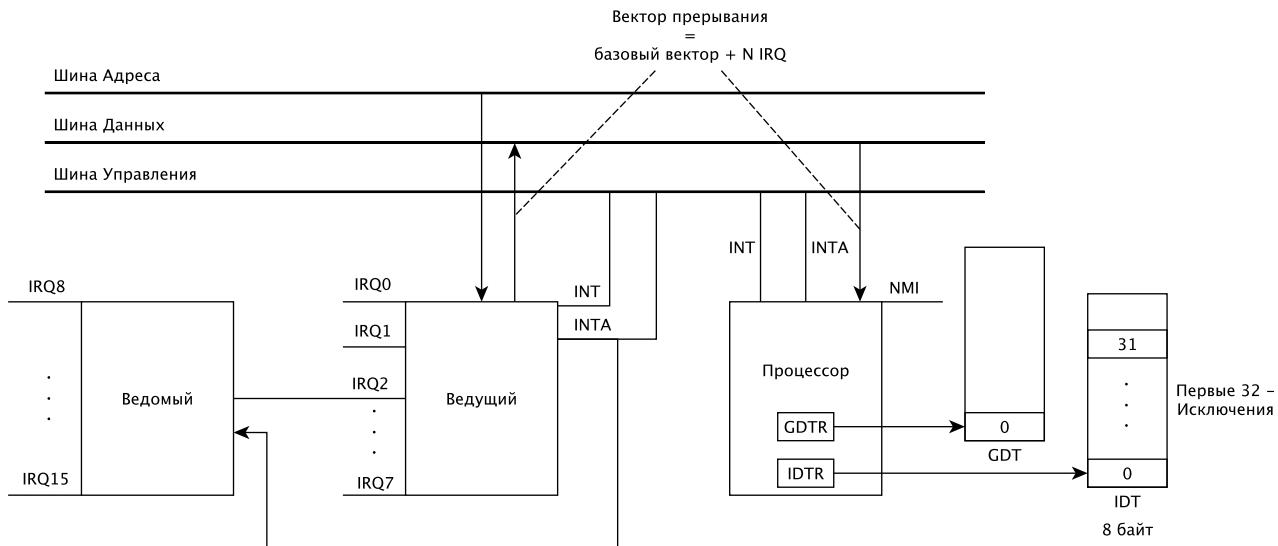


Рис. 1: Управление прерываниями в последовательности ввода-вывода

1 – системный вызов

D1 – передача данных

D5 – данные сохраняются в буфере ядра  
 ИО ИН – выполняется на высоком уровне привилегий (должно быстро завершиться)  
 Программа ввода/вывода – драйвер  
 ОС в стадии выполнения – супервизор.  
 Обработчик системного вызова вызывает драйвер – код, который имеет много точек входа.  
 Драйвер предназначен для управления внешним устройством, его инициализация.  
 По шине данных посыпается информация контроллеру.  
 Любой обработчик аппаратного прерывания входит в драйвер и является его точкой входа.



IRQ1 – сигнал от клавиатуры.

Когда завершается операция ввода/вывода, в буфер клавиатуры записывается код, контроллер отправляет его на ножку IRQ1. Формируется запрос INT и по шине управление поступает в процессор.

В конце цикла выполнение каждой команды процессор проверяет наличие сигнала прерывания и переходит в обработчик, прежде посыпая через INT A сигнал.

Получив INT A контроллер формирует вектор прерывания, который по шине данных поступает в процессор. Вектор используется для адресации обработчика.

У контроллера прерываний есть порт, значит он адресуется.

### 3.1.3 Обработчики аппаратных прерываний

Существуют 2 подхода к реализации обработки прерываний в системе:

- запрет прерываний на время выполнения обработчика прерывания. Запрет прерываний на длительное время в системе – невозможен. Поэтому обработчики прерываний делятся на быстрые и медленные, а медленные делятся на две части: верхнюю и нижнюю половину. Обработчик таймера может инициализировать последующее выполнение планировщика выполнения. Верхняя половина считывает в буфер информацию из контроллера и заканчивается инициализация последующего выполнения нижней половины обработчика. В Unix поддерживаются несколько типов вторых половин. В Windows тоже самое, но через DPC.

- вложенные прерывания.

PSW - аппаратный контекст.

Механизм прерывания реализуется аппаратно (шаги):

- текущее PSW записывается в место старого PSW
- новое PSW загружается в качестве текущего
- по завершению обработки прерывания старое PSW загружается в место текущего, что дает возможность продолжить выполнение прерванной программы.

### 3.1.4 Функции обработчика прерывания системного таймера (int 8h)

Реальный режим

## 1. Инкремент счетчика реального времени

В DOS есть область данных BIOS там находится по известному адресу счетчик реального времени (оперативная память). В компьютере есть энергонезависимая память CMOS микросхема (там и находится счетчик реального времени, при включении компьютера копируется в область данных BIOS).

## 2. Посылка в порт дисковода команды на отключение моторчика дисковода.

## 3. Вызов пользовательского прерывания 1Ch.

### **Защищенный режима**

У первых компьютеров были только дисководы без винчестеров. Чтобы что-то прочитать нужно было разворачивать дискету. А это временные затраты! Было решено возложить функцию отключения моторчика на таймер 2с. На каждом тике он декрементируется. Если 0, то в порт посыпается команда выключения.

INC и DEC поддерживаются аппаратно. Поэтому выполняются быстро.

Вызов пользовательского прерывания INT 1CH - чистая заглушка, чтобы программисты не вставляли большой код в прерывание.

## **Windows**

### **По тику**

- Инкремент счётчика системного времени
- Декремент счетчиков отложенных задач
- Декремент остатка кванта текущего потока.
- Активизация обработчика ловушки профилирования ядра

### **По главному тику**

- Инициализация диспетчера настройки баланса (освобождение объекта «событие» каждую секунду)

### **По кванту**

- Инициализация диспетчеризации потоков (посредством добавления соответствующего объекта DPC в очередь)

## **Unix/Linux**

### **По тику**

- Счет тиков аппаратного таймера
- Декремент кванта текущего потока.
- Инкремент времени использования процессора
- Наблюдение за списком отложенных вызовов.

### **По основному тику**

- Добавление в очередь на выполнение функций, относящихся к работе планировщика-диспетчера (напр: пересчет приоритетов)
- Пробуждение системных процессов, таких, как swapper и pagedaemon (процедура wakeup перемещает дескрипторы процессов из очереди «спящих» в очередь «готовых к выполнению»)
- Обработка сигналов тревоги; декремент будильников и измерение времени работы процесса

### **По кванту**

- Посылка текущему процессу сигнала SIGXCPU, если израсходован выделенный ему квант процессорного времени.

### 3.2 Защищенный режим: назначение системных таблиц – глобальной таблицы дескрипторов (GDT), таблицы дескрипторов прерываний (IDT), теневых регистров (структуры, описывающие дескрипторы GDT и IDT и заполнение дескрипторов в лабораторной работе по защищенному режиму).

Защищенный режим (protected mode) 32-разрядный, многопоточный, многопроцессный, 4 уровня привилегий, доступно 4 Гб виртуальной памяти (для Pentium-64Гб). Параллельные вычисления могут быть защищены программно-аппаратным путем. В защищенном режиме 4 уровня привилегий. Ядро системы находится на 0-м уровне. Создан для работы нескольких независимых программ. Для обеспечения совместной работы нескольких задач необходимо защитить их от взаимного влияния, взаимодействие задач должно регулироваться. Программы, разработанные для реального режима, не могут функционировать в защищенном режиме. (Физический адрес формируется по другим принципам.)

В защищенном режиме адресация выполняется с помощью дескрипторов прерываний (IDT).

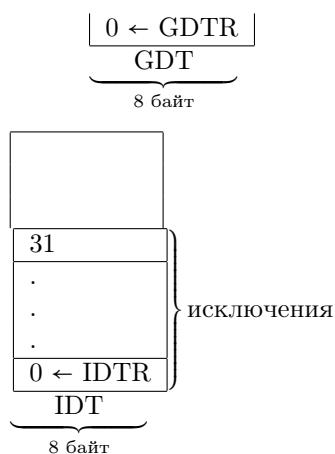
GDT (global descriptor table) – таблица, которая описывает сегменты системы, общие сегменты (сегменты ОП)

На начальный адрес GDT указывает GDT Register (32 разрядный). В системе только одна GDT. 0 дескриптор обязательно пустой.

LDT (local descriptor table) – таблица, которая описывает адресное пространство процесса. В LDT Register находится смещение до соответствующего дескриптора в GDT, описывающего сегмент, в котором находится LDT. Таблиц LDT столько, сколько процессов.

IDT (interrupt descriptor table) – таблица, предназначенная для хранения адресов обработчиков прерываний. Базовый адрес IDT помещен IDT Register.

Для доступа к дескрипторам используются селекторы, записываются в сегментные регистры.



На каждой команде мы обращаемся к памяти, а то и несколько раз. При этом каждый раз будет выполняться преобразование адреса. Поэтому для быстро существуют теневые регистры.

В процессоре для каждого из сегментных регистров имеется так называемый теневой регистр дескриптора, который имеет формат дескриптора. Теневые регистры недоступны программисту; они автоматически загружаются процессором из таблицы дескрипторов каждый раз, когда процессор загружает соответствующий сегментный регистр. Таким образом, в защищенном режиме программист имеет дело с селекторами, т.е. номерами дескрипторов, а процессор – с самими дескрипторами, хранящимися в теневых регистрах. Именно содержимое теневого регистра (в первую очередь, линейный адрес сегмента) определяет область памяти, к которой обращается процессор при выполнении конкретной команды.

В реальном режиме теневые регистры заполняются не из таблицы дескрипторов, а непосредственно самим процессором. В частности, процессор заполняет поле базы каждого теневого регистра линейным базовым адресом сегмента, полученным путем умножения на 16 содержимого сегментного регистра, как это и положено в реальном режиме. Поэтому после перехода в защищенный режим в теневых регистрах находятся правильные линейные базовые адреса, и программа будет выполнять правильно, хотя с точки зрения правил адресации защищенного режима содержимое сегментных регистров лишено смысла.

Тем не менее после перехода в защищенный режим прежде всего следует загрузить в используемые сегментные регистры (и, в частности, в регистр CS) селекторы соответствующих сегментов. Это позволит процессору правильно заполнить все поля теневых регистров из таблицы дескрипторов. Пока эта операция не выполнена, некоторые поля теневых регистров (в частности, границы сегментов) могут содержать неверную информацию.

При переходе из защищенного режима в реальный необходимо записать в теневые регистры FFFF(limit) – максимально возможное смещение, определяющее размер сегмента реального режима 64 Кб.

#### 3.2.1 Пример из лабораторной работы

```

descr struc ; Структура для описания дескриптора сегмента
    limit    dw 0      ; Граница (биты 0..15)
    base_l   dw 0      ; База, биты 0..15
    base_m   db 0      ; База, биты 16..23
    attr_1   db 0      ; Байт атрибутов 1
    arrt_2   db 0      ; Граница(биты 16..19) и атрибуты 2
    base_h   db 0      ; База, биты 24..31
descr ends

intr struc ; Структура для описания дескрипторов прерываний
    offs_l   dw 0      ; Смещение обработчика, нижняя часть (биты 0..15)
    sel      dw 0      ; Селектор сегмента команд
    rsrv    db 0      ; Зарезервировано
    attr    db 0      ; Атрибуты
    offs_h   dw 0      ; Смещение обработчика, верхняя часть (биты 16..31)
intr ends

GDT label byte ; Таблица глобальных дескрипторов
gdt_null descr <> ; Нулевой дескриптор
gdt_flatDS descr <0FFFFh,0,0,92h,0CFh,0>
                    ; переключение в 32-х битную модель памяти flat
                    ; с лимитом в 4 Гб и страничной адресацией
gdt_16bitCS descr <RM_seg_size-1,0,0,98h,0,0>
                ; 16-битный 64-килобайтный сегмент кода с базой
                ; RM_seg
gdt_32bitCS descr <PM_seg_size-1,0,0,98h,0CFh,0>
                ; 32-битный 4-гигабайтный сегмент кода с базой PM_seg
gdt_32bitDS descr <PM_seg_size-1,0,0,92h,0CFh,0>
                ; 32-битный 4-гигабайтный сегмент данных с базой PM_seg
gdt_32bitSS descr <stack_l-1,0,0,92h,0CFh,0>
                ; 32-битный 4-гигабайтный сегмент данных с базой stack_seg
gdt_size = $-GDT ; размер нашей таблицы GDT+1байт (на саму метку)
; сегмент видеобуфера рассчитывается как смещение

gdtr dw gdt_size-1 ; Лимит GDT
       dd ? ; Линейный адрес GDT

; Имена для селекторов
SEL_flatDS equ 8
SEL_16bitCS equ 16
SEL_32bitCS equ 24
SEL_32bitDS equ 32
SEL_32bitSS equ 40

IDT label byte ; Таблица дескрипторов прерываний IDT
trap1 intr 13 dup (<0, SEL_32bitCS,0, 8Fh, 0>)
            ; Первые 32 элемента таблицы (отведены под исключения)
trap13 intr <0, SEL_32bitCS,0, 8Fh, 0> ; Исключение общей защиты
trap2 intr 18 dup (<0, SEL_32bitCS,0, 8Fh, 0>)
            ; Первые 32 элемента таблицы (отведены под исключения)
int08 intr <0, SEL_32bitCS,0, 8Eh, 0> ; Дескриптор прерывания от таймера
int09 intr <0, SEL_32bitCS,0, 8Eh, 0> ; Дескриптор прерывания от клавиатуры
idt_size = $-IDT ; Размер IDT

idtr dw idt_size-1 ; Лимит IDT
       dd ? ; Линейный адрес IDT

idtr_real dw 3FFh,0,0 ; содержимое регистра IDTR в реальном режиме

```

## 4 билет

**4.1 Тупики:** Обнаружение тупиков для повторно используемых ресурсов методом редукции графа, способы представления графа, алгоритмы обнаружения тупиков. Пример анализа состояния системы методом редукции графа. Методы восстановления работоспособности системы.

**Повторно-используемые ресурсы** – количество в системе постоянно и при использовании они не изменяются (или редко): аппаратура (ОП, ЦП), реenterабельные коды, системные таблицы (изменения в них могут вноситься только супервизором), процедуры ОС (так как они являются реenterабельными).

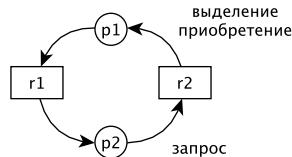
**Потребляемые ресурсы** – количество в ОС переменно и произвольно: сообщения. Процесс может создать любое количество сообщений. Процесс получения сообщения заканчивается его уничтожением.

**Тупик** – ситуация, возникающая в результате монопольного использования разделяемых ресурсов, когда процесс, владея ресурсом, запрашивает другой ресурс, занятый непосредственно или через цепочку запросов другими процессами, ожидающими освобождение ресурса, занятого 1-м процессом.

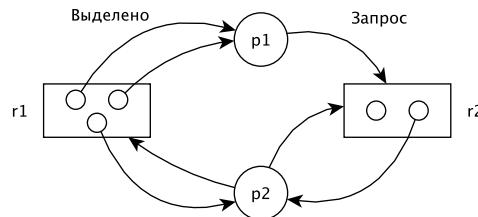
### Методы борьбы с тупиками

1. Недопущение тупиков (в системе создаются такие условия, когда тупик в принципе невозможен)
2. Обход тупиков тупик (возможен, но выполняются действия, которые предотвращают возникновение тупика)
3. Обнаружение тупиков (тупики возможны и они возникают, если они возникли, то из надо обнаружить, чтобы восстановить работоспособность системы)

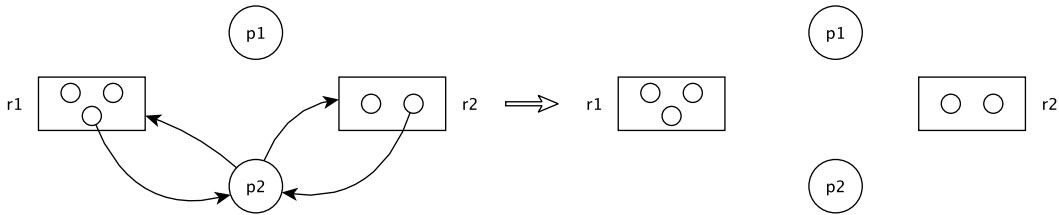
Обнаружение тупиков происходит с помощью графовой модели. Система может быть написана двудольным направленным графом. Два непересекающихся множества вершин (вершины процессов и вершины ресурсов). При этом вершины графа соединяются дугами причём никакая дуга не соединяет вершины одного подмножества. Дуга направленная из вершины принадлежащей множеству ресурсов вершине принадлежащей подмножеству процессов называется **выделением или приобретением ресурсов**. Дуга направленная из вершины относящейся к процессам к ресурсам называется **запросом**.



Обнаружение тупиков по графовой модели выполняется методом редукции графа. Этот метод основан на том эгоистическом предположении, что незаблокированный в тупике процесс может запрашивать приобретать любые нужные ему ресурсы, а затем освобождать их. Освободившиеся ресурсы могут быть выделены или распределены другим процессам, которые их ожидают.



Процессу 1 выделены ресурсы 1 и он запрашивает ещё ресурсы 2. Процессу 2 выделено 1 и 2 и он запрашивает ещё одну 1 и одну 2. Данный график может быть сокращён по вершине p1, поскольку он запрашивает ресурс 1 а он у системы есть. И в результате он освободит занимаемые им единицы ресурсов. Теперь и процесс 2 может запросить ресурсы и завершиться.



Данный граф является полностью сокращаемым. Значит система не находилась в тупике.

### Метод прямого обнаружения

Для представления графа используются матрица распределений и матрица запросов.

$$S \xrightarrow{i} T$$

Кол-во единиц j-го ресурса выделенное i-му процессу.

$$|b_{ij}| - \text{матрица выделений}$$

$$|a_{ij}| - \text{матрица запросов}$$

Вектор свободных единиц ресурсов.

$$\mathcal{F} = (\dots, f_i, \dots)$$

Для повторно используемых ресурсов в системе хранится информация о типах.

Кол-во выделенных + кол-во свободных = кол-во единиц данного ресурса в системе.

Матрица распределения			Матрица запросов			Свободные ресурсы		
P1	1 2 3		P2	1 2 3		P1	1 2 3	
1	0 1 1		1	1 1 0		1	0 1 1	
2	1 3 0		2	0 0 1		2	0 0 0	
3	1 5 0		3	1 1 2		3	1 5 0	
	2 0 1			8 2 6 9 3				
					F = 532			
						P1	1 2 5	
						1	0 0 0	
						2	0 0 0	
						3	1 5 0	
							F = 543	
							1 0 0 0	
							2 0 0 0	
							3 0 0 0	

### Восстановление системы

Существует два самых общих подхода.

- Прекращение процессов при этом завершаются процессы попавшие в тупик одни за другим. Тогда будут доступны ресурсы для других процессов попавших в тупик.
- Перехват ресурсов у процессов которые в тупик не попали. Отобрать ресурс можно только в результате отката. Процесс возвращается в состояние до запроса ресурса, но такой откат должен быть программно описан.

Критерии завершения процесса

- Приоритет
- Цена повторного запуска процесса
- Внешняя цена

## 4.2 Задача «Обедающие философы» – модели распределение ресурсов вычислительной системы. Множественные семафоры UNIX: системные вызовы, поддержка в системе, пример использования из лабораторной работы «производство-потребление».

### Множественные семафоры

Современные ОС поддерживают именно множественные семафоры, представляются как массивы считающих семафоров. И Windows, и Linux, и Unix поддерживают. Рассмотрим на примере задачи обедающие философы.

На круглом столе стоит 5 тарелок. Напротив каждой тарелки стоит философ. Жизнь философа проста: какое-то время он думает, какое-то ест. Есть философ может только если завладеет двумя приборами. Изначально у каждой тарелки один прибор. В связи с этим, различаются три способа действия философов:

1. Каждый пытается взять две вилки, если ему это удаётся, то он может есть.
2. Философ пытается взять правую вилку, после чего, если удалось взять правую, удерживая ее, пытается взять левую.
3. Философ пытается взять правую, если не может взять левую, то кладёт правую на место.

Это три негативные ситуации в системе:

1. Бесконечное откладывание. Сколько философов умрет голодной смертью? Голодание – бесконечное откладывание
2. Все философы одновременно взял правую вилку, и никто не может взять левую, они блокируют друг друга.
3. Захват и освобождение одних и тех же ресурсов.

Самое важное свойство семафоров: одной неделимой командой можно изменить все или часть семафоров набора. Если в программе используется большое число семафоров, которые освобождаются и объявляются в разных функциях, очень сложно отладить программу и найти когда она входит в тупик.

Win, Unix, современный Linux поддерживают наборы считающих семафоров, они обладают важнейшим свойством: одной неделимой операцией можно изменить все или часть семафоров набора.

Если какая-либо операция не может быть выполнена над одним семафором, то неудачной считается операция над всеми семафорами.

Семафоры поддерживаются в ядре таблицей семафоров. Каждая строка описывает 1 набор из всех созданных в системе наборов семафоров.

1. Имя – целое число, присваивается процессам создавшим набор семафоров. Другие процессы могут по имени открыть.
2. UID – идентификатор создателя набора семафора и его группы. Если UID процесса совпадает с UID создателя, то может удалять набор и изменять его управляющие параметры.
3. Права доступа
4. Количество семафоров в наборе
5. Время изменения одного или нескольких значений семафора последним процессом
6. Время изменения управляющих параметров
7. Указатель на массив семафоров

О каждом семафоре известно:

1. Значение семафора
2. ID процесса, который оперировал с семафором в последний раз
3. Число процессов заблокированных на семафоре

В отличие от семафора Дейкстры, на семафоре в Unix определено 3 операции:

- Захват, декремент
- Освобождение, инкремент
- Проверка семафора на 0. Процесс выполняющий такую проверку переходит в состояние ожидания освобождения ресурса.

В системе определены структуры для работы с семафорами: <sys/sem.h>

```
struct sembuf
{
    n_short sem_num; // индекс
    short sem_op; // операция
    short sem_flg; // флаги
}
```

3 операции:

- sem\_op < 0. Захват, декремент. Если не может то блокируется в операции.

- `sem_op > 0`. Освобождение, инкремент
- `sem_op = 0`. Проверка семафора на 0.

На семафорах определены флаги:

`IPC_NOWAIT` – информирует ядро о нежелании процесса переходить в состояние ожидания.

Это позволяет избежать очереди к семафору, в случае аварийного завершения или `kill`.

В силу того, что `kill` невозможно прекратить, убиваемый процесс не может осуществить освобождение семафора.

`SEM_UNDO` – указывает ядру, что необходимо отслеживать значение семафора, в результате завершения процесса, вызвавшего `sem_op`, ядро отменяет все изменения, чтобы процессы не были блокированы навсегда.

1. Взаимоисключений, организация монопольного доступа процесса к разделяемой переменной.

Чистое взаимоисключение реализуется в задаче читателя-писателя осуществляется монопольный доступ писателя.

2. Синхронизация, когда процесс заинтересован в действиях другого процесса.

Задача производства потребления, если потребитель работает быстрее производителя возникнет ситуация когда буфер пуст, потребитель будет ожидать когда производитель положит что-нибудь в буфер, это видно при передаче сообщений, сообщения несут информацию которая интересует процесс, для того чтобы продолжить свое выполнение.

```
#define P -1
#define V +1

#define SB 0
#define SE 1
#define SF 2

typedef struct sembuf sembuf;

sembuf take_semafor_producer[2] =
{
    { SE, P, SEM_UNDO },
    { SB, P, SEM_UNDO }
};

sembuf free_semafor_producer[2] =
{
    { SB, V, SEM_UNDO },
    { SF, V, SEM_UNDO },
};

sembuf take_semafor_consumer[2] =
{
    { SF, P, SEM_UNDO },
    { SB, P, SEM_UNDO }
};

sembuf free_semafor_consumer[2] =
{
    { SB, V, SEM_UNDO },
    { SE, V, SEM_UNDO }
};

void producer(int* shared_bufer, const int len, const int semid)
{
    while(1)
    {
        sleep(1);

        if (semop(semid, take_semafor_producer, 2) == -1)
            semop_error();
    }
}
```

```

*(shared_bufer + *shared_bufer) = *shared_bufer - 1;

printf("Produser [\%d] --> \%d\n", getpid(), *(shared_bufer + *
    shared_bufer));

(*shared_bufer)++;

if (semop(semid, free_semafor_producer, 2) == -1)
    semop_error();

if (*shared_bufer >= len - 1)
{
    *shared_bufer = 2;
}
}

exit(0);
}

void consumer(int* shared_bufer, const int len, const int semid)
{
    while(1)
    {
        sleep(1);

        if (semop(semid, take_semafor_consumer, 2) == -1)
            semop_error();

        printf("Consumer [\%d] <- \%d\n", getpid(), *(shared_bufer + *(

            shared_bufer + 1)));

        (*(shared_bufer + 1))++;

        if (semop(semid, free_semafor_consumer, 2) == -1)
            semop_error();

        if (*(shared_bufer + 1) >= len - 1)
        {
            *(shared_bufer + 1) = 2;
        }
    }

    exit(0);
}

#define COUNT 3
#define N 8
#define PERM S_IRWXU | S_IRWXG | S_IRWXO

int main(void)
{
    pid_t producers[COUNT];
    pid_t consumers[COUNT];
    int status, shmid, semid, ctrl_sb, ctrl_se, ctrl_sf, i;
    int* shared_bufer;

    shmid = shmget(IPC_PRIVATE, (N + 2) * sizeof(int), IPC_CREAT | PERM);

    if (shmid == -1)
        shget_error();

    shared_bufer = shmat(shmid, 0, 0);
}
```

```

if ((*((int*)shared_bufer) == -1)
    shmat_error();

semid = semget(IPC_PRIVATE, 3, IPC_CREAT | PERM);

if (semid == -1)
    semget_error();

ctrl_sb = semctl(semid, SB, SETVAL, 1);
ctrl_se = semctl(semid, SE, SETVAL, N);
ctrl_sf = semctl(semid, SF, SETVAL, 0);

if (ctrl_sb == -1 || ctrl_se == -1 || ctrl_sf == -1)
    semctl_error();

*shared_bufer = 2;
*(shared_bufer + 1) = 2;

for (i = 0; i < COUNT; ++i)
{
    producers[i] = fork();

    if (producers[i] == -1)
        fork_error();
    else if (producers[i] == 0)
        producer(shared_bufer, N, semid);

    consumers[i] = fork();

    if (consumers[i] == -1)
        fork_error();
    else if (consumers[i] == 0)
        consumer(shared_bufer, N, semid);

    sleep(2);
}

for (i = 0; i < COUNT; ++i)
{
    wait(&status);
    wait(&status);
}

if (shmctl(shmid, IPC_RMID, NULL) == -1)
    clean_error();

if (semctl(semid, 0, IPC_RMID, 0) == -1)
    clean_error();

return 0;
}

```

## 5 билет

5.1 Виртуальная память: распределение памяти страницами по запросам, схема с гиперстраницами, обоснование использования данной схемы. Управление памятью страницами по запросам в архитектурах x86 – расширенное преобразование (PAE) – схема преобразований. Анализ страничного поведения процессов: свойство локальности, рабочее множество.

**Виртуальная память** – память, размер которой превышает размер реального физического пространства. Используется адресное пространство диска как область swapинга или пейджинга, т.е. для временного хранения областей памяти.

Подходы к реализации управления виртуальной памятью:

1. управление памятью страницами по запросам
2. сегментами по запросам
3. сегментами, деленными на страницы по запросам

**Страница** – является единицей физического деления памяти. Её размер устанавливается системой. У страницы размер не обязательно 4Кб. Просто такой размер оптимальен по количеству страничных прерываний.

**Сегмент** – является единицей логического деления памяти. Её размер определяется объемом кода. Любой сегмент создается такого размера, который требуется программе.

**Запрос** – страничное прерывание, которое возникает при попытке доступа к незагруженной в память странице. При его обработке происходит пейджинг.

**Пейджинг** – загрузка с диска новых и замещение старых страниц.

### Двухуровневая страничная организация

При увеличении размера программы, увеличивается виртуальное адресное пространство, все это влечет увеличение размера таблиц страниц (находятся в адресном пространстве ядра системы). Таблиц страниц столько, сколько процессов в памяти. Чтобы сократить расходы на это хранение, в IBM/360 версии 67 и IBM 370 была предложена двухуровневая страничная организация.

В ней вводятся гиперстраницы, адресное пространство делится на гиперстраницы и страницы. Приводят к появлению виртуального адреса, состоящего из 3 частей.

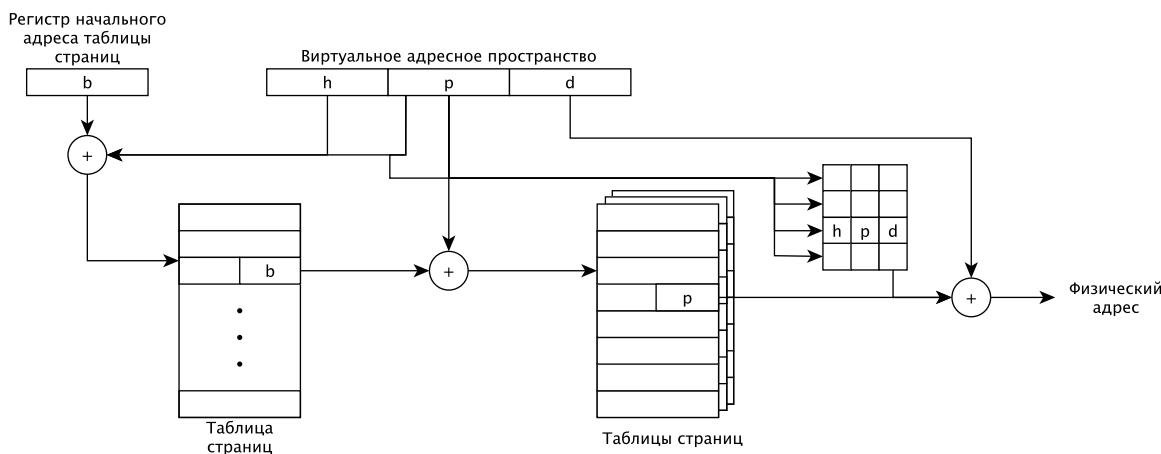


Таблица гиперстраниц на каждый процесс одна, но таблиц страниц столько же, сколько и страниц.

В итоге, сначала адрес физической страницы ищется в ассоциативном кэше, далее 2 обращения к физической памяти – затратное действие (так как программа не может обращаться к нескольким страницам).

В таблицу гиперстраниц загружаются только актуальные таблицы страниц. Время обращения увеличивается, то есть за эффективное хранение данных мы платим увеличением времени преобразования.

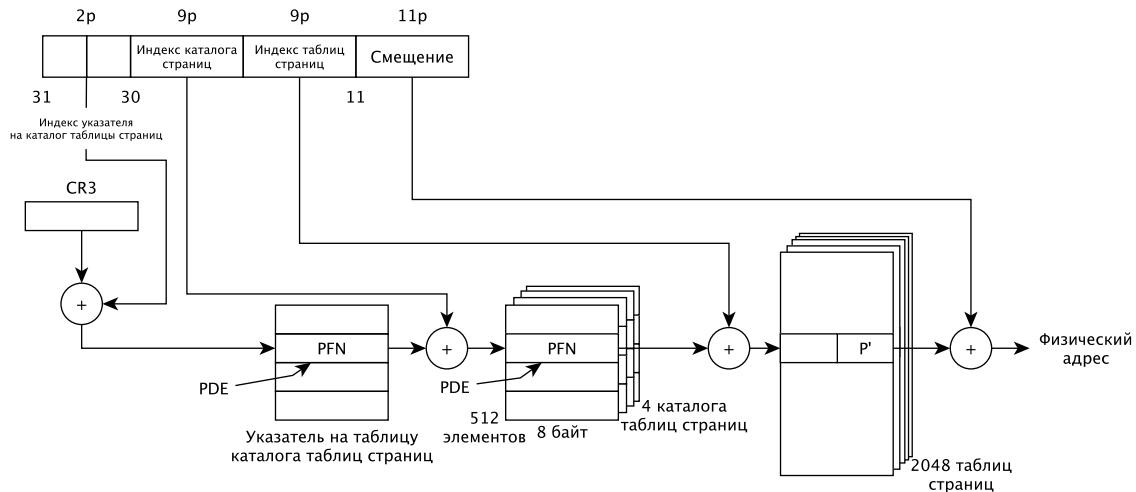
Начиная с Pentium Pro включен регистр CR4, где 5 бит это PAE (physical address extension) – расширение физического адреса.

В режиме PAE виртуальный адрес делится на 4 поля.

Если PAE = 1, то разрешено использование расширенной 36рр, вместо 32 pp физического адреса. При этом физический адрес остается 32 pp, все изменения касаются только работы страничного механизма.

PFN – Page Frame Number

PDE – Page Describe Entry



### Анализ страничного поведения процессов: свойство локальности, рабочее множество.

Свойство локальности:

Если обращение было к странице, то наиболее вероятно, что следующее обращение будет к этой же странице. (Если не слишком далеко заглядывать в будущее, то можно достаточно точно его прогнозировать исходя из прошлого.)

Страницное поведение программ – число страницных прерываний, происходящих в процессе.

Зависимость числа страницных прерываний от объема памяти является обобщенной мерой страницного поведения программы.

Программа одновременно не обращается ко всем своим страницам, позволяет использовать виртуальное пространство.

В разные моменты времени обращается к разным подмножествам множества страниц. В том числе в разные моменты времени программы обращаются к разному количеству страниц. Предсказать к каким страницам обратиться невозможно.

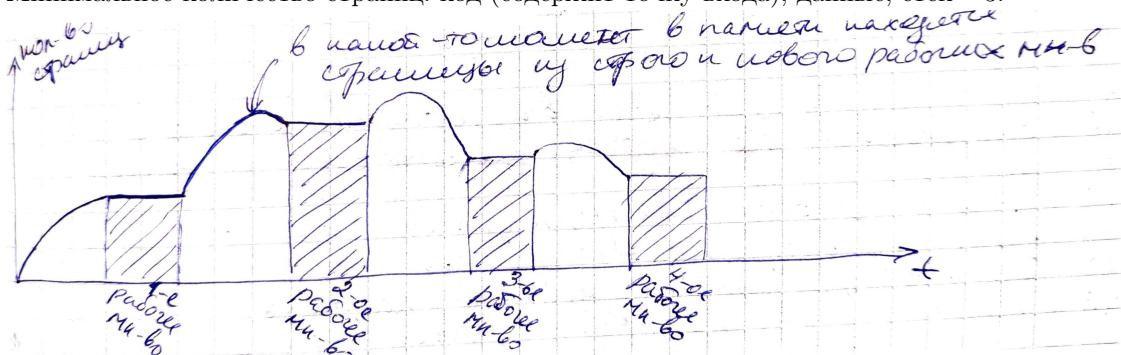
Деннинг в 1968 году предложил в качестве локальной меры производительности взять число страниц, к которым обращается программа за интервал времени  $\Delta t$ .



Размер рабочего множества является монотонной функцией от  $\Delta t$ , то есть при увеличении интервала  $\Delta t$ , число страниц будет стремиться к пределу  $L$  – которое определяет необходимое количество страниц для выполнения программы.

Нужно, чтобы рабочее множество находилось в оперативной памяти. Трэшинг – подкачка одних и тех же страниц.

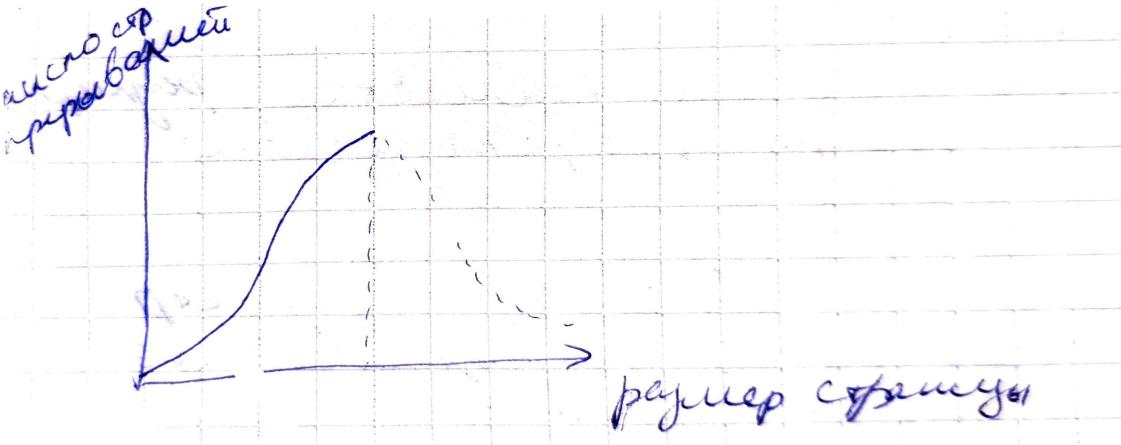
Минимальное количество страниц: код (содержит точку входа), данные, стек – 3.



Рабочее множество – набор страниц, используемых процессом в настоящее время.

Процесс удачных обращений – hit rate.

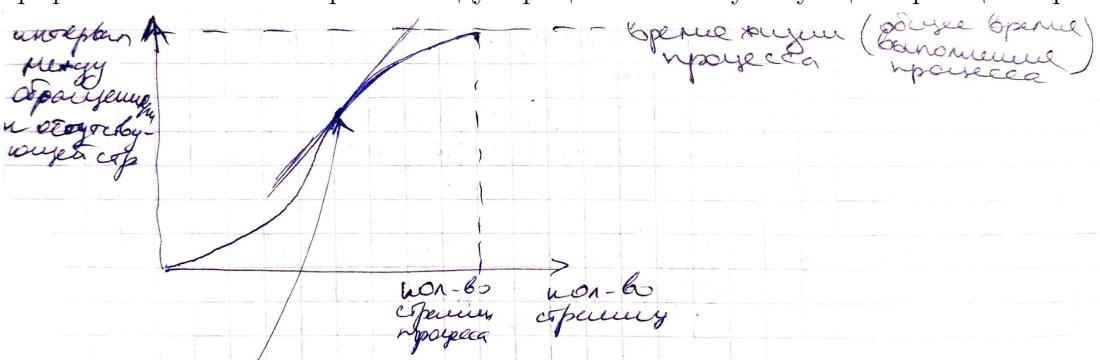
График. Влияние изменения размера страниц при сохранении объема оперативной памяти, на число прерываний.



С ростом размера страниц число страницных прерываний растет. Когда размер страницы становится соизмеримым с размером программы число страницных прерываний резко падает.

Чем меньше размер страницы тем больше страниц, но на маленьких страницах выполняется большое число команд.

График. Зависимость интервала между обращениями к отсутствующей странице за время жизни процесса.



Стрелка указывает на точку перегиба, в некоторый момент в оперативной памяти оказывается все рабочее множество процесса.

Система может контролировать количество страницных прерываний, если резко возросло, то означает что рабочее множество не может быть загружено в память.

Необходимо увеличить размер квоты на непродолжительный интервал времени.

Квота – каждому процессу может быть выделено определенное количество страниц, изначально.

## 5.2 Задача «Производство-потребление»: алгоритм Эд. Дейкстры, реализация на семафорах UNIX (код из лабораторной работы).

```
#define P -1
#define V +1

#define SB 0
#define SE 1
#define SF 2

typedef struct sembuf sembuf;

sembuf take_semafor_producer [2] =
{
    { SE, P, SEM_UNDO },
    { SB, P, SEM_UNDO }
};

sembuf free_semafor_producer [2] =
{
    { SB, V, SEM_UNDO },
    { SF, V, SEM_UNDO },
};

sembuf take_semafor_consumer [2] =
```

```

{
    { SF, P, SEM_UNDO },
    { SB, P, SEM_UNDO }
};

sembuf free_semafor_consumer[2] =
{
    { SB, V, SEM_UNDO },
    { SE, V, SEM_UNDO }
};

void producer(int* shared_bufer, const int len, const int semid)
{
    while(1)
    {
        sleep(1);

        if (semop(semid, take_semafor_producer, 2) == -1)
            semop_error();

        *(shared_bufer + *shared_bufer) = *shared_bufer - 1;

        printf("Produser [%d] --> %d\n", getpid(), *(shared_bufer + *
            shared_bufer));

        (*shared_bufer)++;

        if (semop(semid, free_semafor_producer, 2) == -1)
            semop_error();

        if (*shared_bufer >= len - 1)
        {
            *shared_bufer = 2;
        }
    }

    exit(0);
}

void consumer(int* shared_bufer, const int len, const int semid)
{
    while(1)
    {
        sleep(1);

        if (semop(semid, take_semafor_consumer, 2) == -1)
            semop_error();

        printf("Consumer [%d] <- %d\n", getpid(), *(shared_bufer + *((shared_bufer + 1))));

        (*(shared_bufer + 1))++;

        if (semop(semid, free_semafor_consumer, 2) == -1)
            semop_error();

        if (*(shared_bufer + 1) >= len - 1)
        {
            *(shared_bufer + 1) = 2;
        }
    }

    exit(0);
}

```

```

}

#define COUNT 3
#define N 8
#define PERM S_IRWXU | S_IRWXG | S_IRWXO

int main(void)
{
    pid_t producers[COUNT];
    pid_t consumers[COUNT];
    int status, shmid, semid, ctrl_sb, ctrl_se, ctrl_sf, i;
    int* shared_bufer;

    shmid = shmget(IPC_PRIVATE, (N + 2) * sizeof(int), IPC_CREAT | PERM);

    if (shmid == -1)
        shmat_error();

    shared_bufer = shmat(shmid, 0, 0);

    if (*((int*)shared_bufer) == -1)
        shmat_error();

    semid = semget(IPC_PRIVATE, 3, IPC_CREAT | PERM);

    if (semid == -1)
        semget_error();

    ctrl_sb = semctl(semid, SB, SETVAL, 1);
    ctrl_se = semctl(semid, SE, SETVAL, N);
    ctrl_sf = semctl(semid, SF, SETVAL, 0);

    if (ctrl_sb == -1 || ctrl_se == -1 || ctrl_sf == -1)
        semctl_error();

    *shared_bufer = 2;
    *(shared_bufer + 1) = 2;

    for (i = 0; i < COUNT; ++i)
    {
        producers[i] = fork();

        if (producers[i] == -1)
            fork_error();
        else if (producers[i] == 0)
            producer(shared_bufer, N, semid);

        consumers[i] = fork();

        if (consumers[i] == -1)
            fork_error();
        else if (consumers[i] == 0)
            consumer(shared_bufer, N, semid);

        sleep(2);
    }

    for (i = 0; i < COUNT; ++i)
    {
        wait(&status);
        wait(&status);
    }
}

```

```
if (shmctl(shmid, IPC_RMID, NULL) == -1)
    clean_error();

if (semctl(semid, 0, IPC_RMID, 0) == -1)
    clean_error();

return 0;
}
```

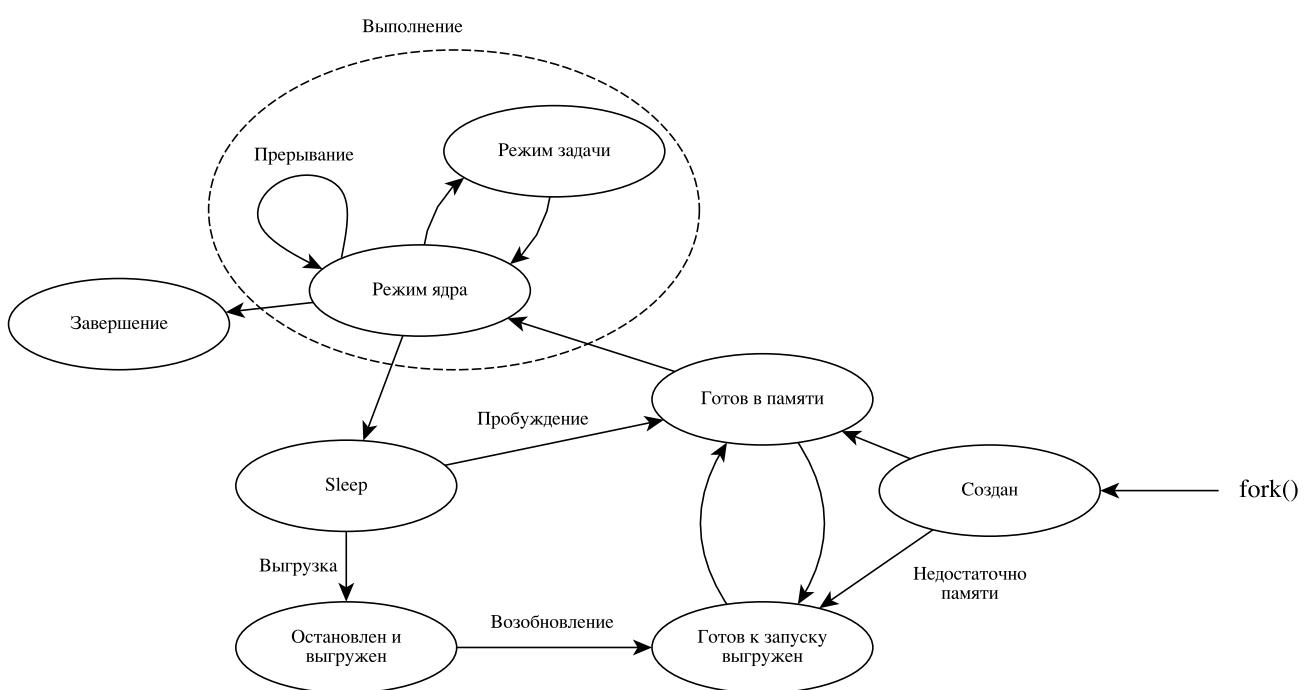
## 6 билет

6.1 Понятие процесса. Процесс как единица декомпозиции системы. Диаграмма состояний процесса с демонстрацией действий, выполняемых в режиме ядра. Планирование и диспетчеризация. Классификация алгоритмов планирования. Примеры алгоритмов планирования, соотнесенные с типами ОС. Процессы и потоки. Типы потоков.

Процесс – программа в стадии выполнения.

Процесс – главная абстракция системы. UNIX декларирует следующим образом: процесс – часть времени, выполняет собственный код, и тогда он выполняется в **режиме задачи**, а часть времени выполняет реинтегрируемый код операционной системы, и тогда он выполняется в **режиме ядра**.

### 6.1.1 Диаграмма состояний процесса



**Планирование scheduling** – постановка процесса в очередь на выполнение.

- без переключения / с переключением — процесс может выполняться от начала до конца, а может быть снят с выполнения (например при переключении кванта и возвращен в очередь готовых процессов)
- с приоритетами / без приоритетов

- с вытеснением / без вытеснением — процесс может быть вытеснен другим процессом с более высоким приоритетом, процесс снимается с выполнения и возвращается в очередь готовых процессов, а процессорное время передается процессу с более высоким приоритетом (вытеснение невозможно в системе без приоритетов)

### Приоритеты:

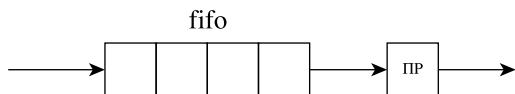
- статические - назначаются в начале и не меняются
- динамические - меняются в процессе выполнения

**Планирование в мультипрограммных системах пакетной обработки** (планирование на принципе распараллеливания функций. Программа выполняется до тех пор, пока не запросит дополнительный ресурс системы или пока не придет более высокоприоритетный процесс, в случае поддержки вытеснения, может вытеснить):

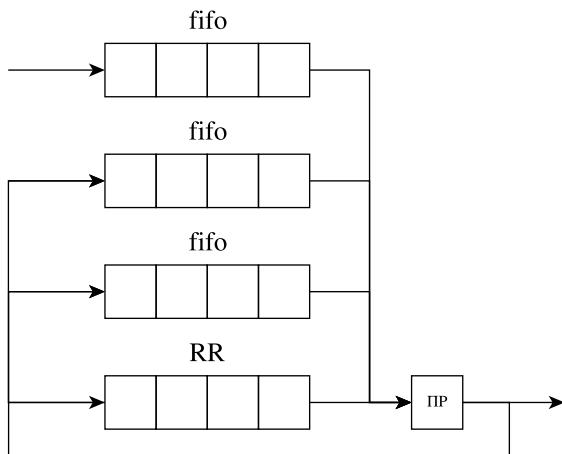
- FIFO - first in first out – без вытеснения, без приоритетов. После блокировки будет поставлен в конец очереди.
- SJF - shortest job first – кратчайшее задание первыми - меньшее количество процессорного времени. Приводило к тому, что задания с большим процессорным временем все время откладывались в конец очереди — бесконечное откладывание.
- SRT - shortest remaining time (наименьшее оставшееся время). В этом алгоритме выполняющийся процесс вытесняется, если поступит процесс с меньшим оценочным временем выполнения, чем оставшееся время процесса текущего.
- HRN - highest response rationext – наибольшее относительное время ответа. В этом алгоритме приоритет вычисляется по формуле:  $P = \frac{t_w - t_s}{t_s}$ , где  $t_s$  – запрошенное время обслуживания.  $t_w$  – время ожидания.

**Планирование в системах разделения времени** (каждому выделяется квант процессорного времени. Процесс выполняется до тех пор, пока не истек квант, не начался процесс ввода/вывода или не вытеснен другим высокоприоритетным процессом.): Время процессам выделяется квантами.

- RR - циклическое планирование — процессы выстраиваются в очередь, но по истечении кванта ставятся в конец очереди.



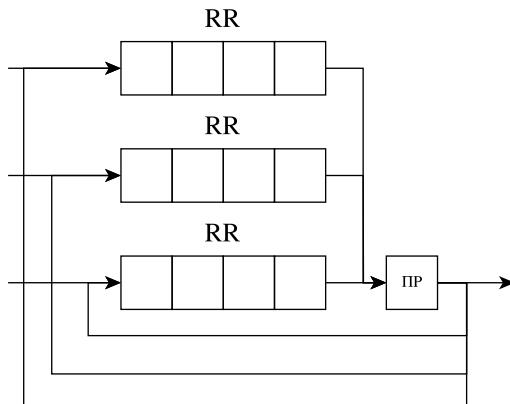
- Многоуровневые очереди или алгоритм адаптивного планирования – каждая очередь это fifo.



В первую очередь с наивысшим приоритетом поступают только что созданные процессы или завершившие блокировку в ожидании завершения ввода/вывода. Для 1 очереди квант процессорного времени выбирается таким образом, чтобы наибольшее количество процессов успело или завершится, или выполнить запрос на ввод/вывод. Если процесс не успел завершиться за выделенный квант процессорного времени, он поступает в следующую очередь с более низким приоритетом и так далее пока не окажется в очереди с самым низким приоритетом, которая будет работать по принципу RR. В этой очереди крутится так называемый холостой процесс, поскольку система не может ничего не делать.

Пересчитываться могут только пользовательские приоритеты (приложений). Стремится быть справедливым, чтобы у всех процессов было примерно одинаковое время, но также учитывается время простоя процесса. В старом UNIX используется формула.

- Интерактивные – процессы, запрашивающие ввод-вывод с клавиатуры, мыши. Самые высоко-приоритетные операции.



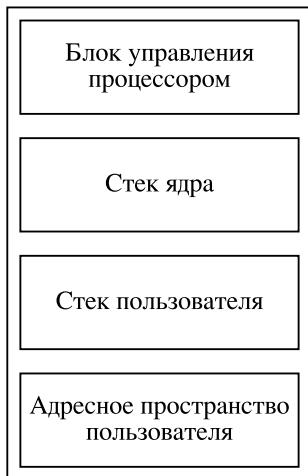
**Диспетчеризация** – непосредственно выделение процессу процессорного времени.

### 6.1.2 Потоки

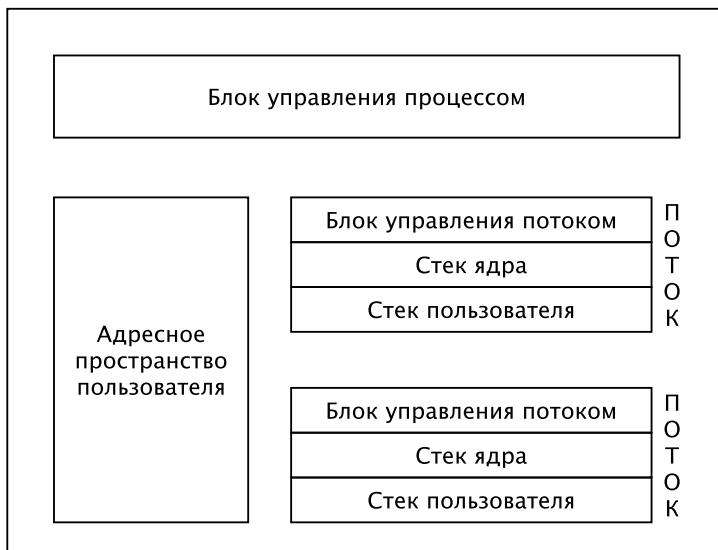
В качестве потока выделяются части кода процесса, которые могут выполняться параллельно с другими частями кода процессора. В любом случае владельцем ресурсов в процессе является процесс. Поток выполняется в адресном пространстве процесса (так как не имеет своего адресного пространства). Поток оказывается владельцем счетчика команд (аппаратного контекста).

Существуют многопоточные и однопоточные модели процесса.

#### Однопоточные модели



#### Многопоточные модели



Потоки разделяют одно адресное пространство. Виртуальное адресное пространство процесса делится на две части: пользовательское пространство и оперативная память.

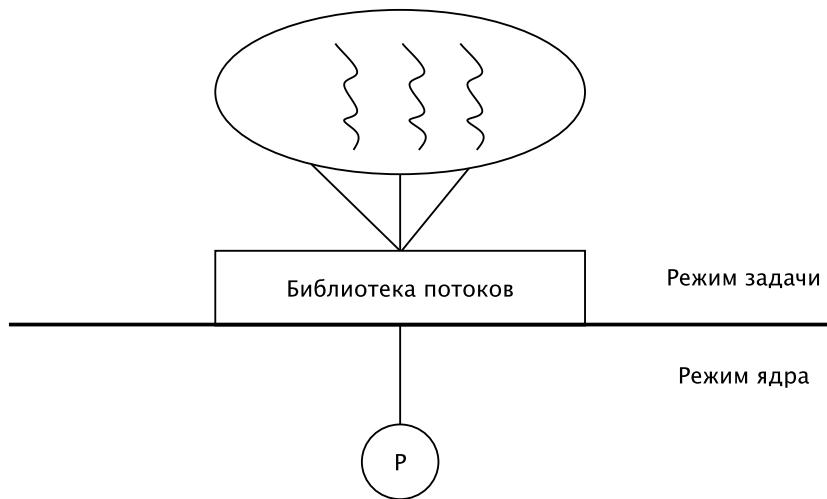
Единицей диспетчеризации становится поток, но приоритетом владеет процесс. А уровень приоритета потока определяется относительно приоритета процесса. Должна быть структура описывающая поток. Должны быть системные вызовы или функции создающие потоки. В нашей системе два стека стек ядра и стек пользователя. У нас нет отдельного стека для прерываний, но каждый процесс имеет два стека описанных выше. Стек ядра создаётся по умолчанию и используется когда процесс находится в режиме ядра. Невозможно информацию о режиме ядра писать в стек пользователя. Приложение может иметь несколько стеков и возникает неоднозначность. В многопоточной модели каждый поток должен иметь стек ядра и стек пользователя, так как переключаются уже потоки.

Чтобы сократить накладные расходы на переключение контекста (это очень затратно).

**Возникло два типа потоков:**

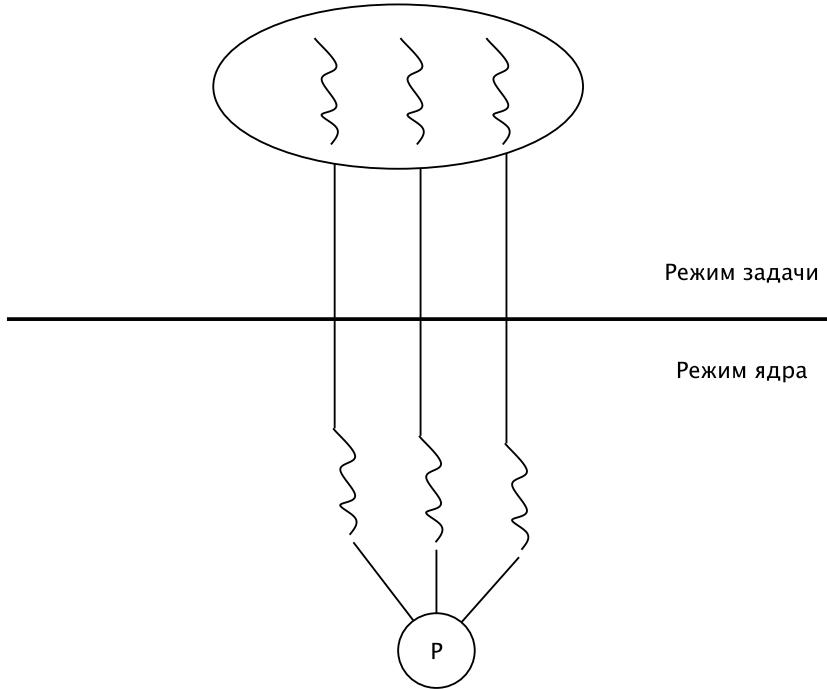
- Потоки на уровне пользователя

О потоках режима пользователя ядро ничего не знает. Должна быть специальная библиотека уровня пользователя. Она должна предоставлять функции создания и удаления, планирования потоков, диспетчеризации, сохранения контекста, обеспечивать возможность взаимодействия потоков. Все хорошо пока поток не запросит ввод/вывод (дополнительный системный ресурс). Он приведёт к тому что все потоки процесса будут заблокированы и процесс перейдёт в режим ядра.



- Потоки на уровне ядра

При переключении потоков (одного процесса) переключается только аппаратный контекст. При переключении потока другого процесса будет переключен полный контекст. В очередь к процессору стоят потоки, являются единицей диспетчеризации, разных процессов.



## 6.2 Обеспечение монопольного доступа к разделяемым данным в задаче «читатели-писатели» : реализация на базе Win32 API (пример кодов лабораторной работы «читатели-писатели» для ОС Windows).

```

#include <stdio.h>
#include <windows.h>
#include <iostream>
using namespace std;

#define OK 0
#define ERROR 1

#define WRITERS 3
#define READERS 5
#define ITERS 5

int val = 0;
bool activewriter = false;
int readercount = 0;

int waiting_writers = 0;
int waiting_readers = 0;

HANDLE writers[WRITERS];
HANDLE readers[READERS];
HANDLE can_write;
HANDLE can_read;

void start_read() {
    InterlockedIncrement(&waiting_readers);
    if (true == activewriter || waiting_writers > 0) {
        WaitForSingleObject(can_read, INFINITE);
    }
    InterlockedDecrement(&waiting_readers);
    InterlockedIncrement(&readercount);
    SetEvent(can_read);
}

```

```

void stop_read() {
    InterLockedDecrement(readercount);
    if (0 == readercount) {
        SetEvent(can_write);
    }
}

void start_write() {
    InterlockedIncrement(&waiting_writers);
    if (readercount > 0 || true == activewriter) {
        WaitForSingleObject(can_write, INFINITE);
    }
    InterlockedDecrement(&waiting_writers);
    activewriter = true;
}

void stop_write() {
    activewriter = false;
    ResetEvent(can_write);
    if (waiting_readers > 0) {
        SetEvent(can_read);
    } else {
        SetEvent(can_write);
    }
}

DWORD WINAPI writer(LPVOID) {
    for (int i = 0; i < ITERS; i++) {
        start_write();
        val++;
        cout << "Writer" << GetCurrentThreadId() << " write " << val << endl;
        stop_write();
        Sleep(100);
    }
    return OK;
}

DWORD WINAPI reader(LPVOID mutex) {
    for (int i = 0; i < ITERS + 7; i++) {
        start_read();
        WaitForSingleObject(mutex, INFINITE);
        cout << "Reader" << GetCurrentThreadId() << " read " << val << endl;
        ReleaseMutex(mutex);
        stop_read();
        Sleep(100);
    }
    return OK;
}

int create_mutex_threads() {
    HANDLE mutex = CreateMutex(NULL, FALSE, NULL);
    if (NULL == mutex) {
        cout << "Can't create mutex\n";
        return ERROR;
    }
    // создание писателей
    for (int i = 0; i < WRITERS; i++) {
        // данный аргумент определяет, может ли создаваемый поток быть унаследован
        // и дочерним процессом
        // размер стека в байтах. Если передать 0, то будет использоваться значение
        // по - умолчанию (1 мегабайт)
        // адрес функции, которая будет выполняться потоком
    }
}

```

```

// указатель на переменную, которая будет передана в поток
// флаги создания
// указатель на переменную, куда будет сохранён идентификатор потока

writers[i] = CreateThread(NULL, 0, &writer, NULL, 0, NULL);
if (NULL == writers[i]) {
    cout << "Can't create threads\n";
    return ERROR;
}
}

// создание читателей
for (int i = 0; i < READERS; i++) {
    readers[i] = CreateThread(NULL, 0, &reader, mutex, 0, NULL);
    if (NULL == readers[i]) {
        cout << "Can't create threads\n";
        return ERROR;
    }
}

return OK;
}

int create_events() {
// атрибут защиты
// тип сброса TRUE - ручной
// начальное состояние TRUE - сигнальное
// имя объекта

// с автосбросом
can_read = CreateEvent(NULL, FALSE, FALSE, TEXT("ReadEvent"));
if (can_read == NULL) {
    cout << "Can't create event\n";
    return ERROR;
}

// с ручным сбросом
can_write = CreateEvent(NULL, TRUE, FALSE, TEXT("WriteEvent"));
if (can_write == NULL) {
    cout << "Can't create event\n";
    return ERROR;
}
return OK;
}

int main() {

if (create_events() != OK) {
    return ERROR;
}
if (create_mutex_threads() != OK) {
    return ERROR;
}

WaitForMultipleObjects(WRITERS, writers, TRUE, INFINITE);
WaitForMultipleObjects(READERS, readers, TRUE, INFINITE);

return OK;
}

```

Семейство Interlocked-функций используются в случае, если разным потокам необходимо изменять одну и ту же переменную.

## 7 билет

- 7.1 Управление виртуальной памятью: распределение памяти сегментами по запросам: схема преобразования виртуального адреса, способы организации таблиц сегментов, стратегии выбора разделов памяти для загрузки сегментов, алгоритмы и особенности замещения сегментов.

**Виртуальная память** – память, размер которой превышает размер реального физического пространства. Используется адресное пространство диска как область свопинга или пейджинга, т.е. для временного хранения областей памяти.

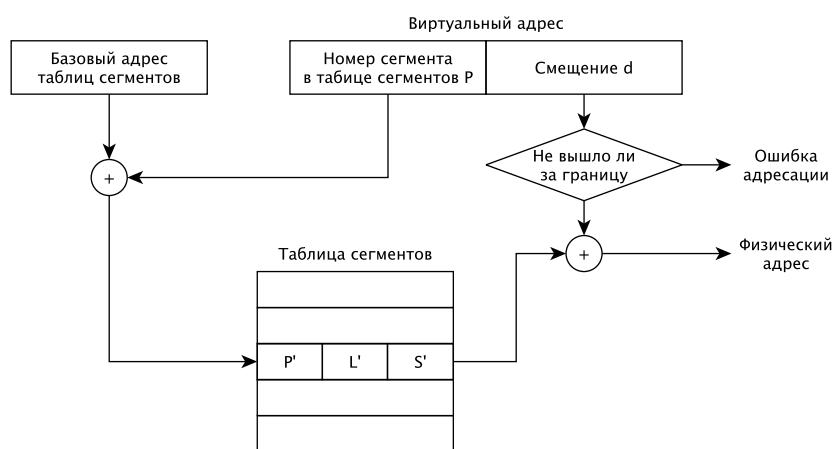
**Страница** – является единицей физического деления памяти. Её размер устанавливается системой. У страницы размер не обязательно 4Кб. Просто такой размер оптимальен по количеству страницных прерываний.

**Сегмент** – является единицей логического деления памяти. Её размер определяется объемом кода. Любой сегмент создается такого размера, который требуется программе.

Виртуальный адрес делится на 2 части – регистр, смещение. В процессоре должен существовать начальный адрес таблицы сегментов.

В дескрипторе сегмента присутствует поле, определяющее размер сегмента.

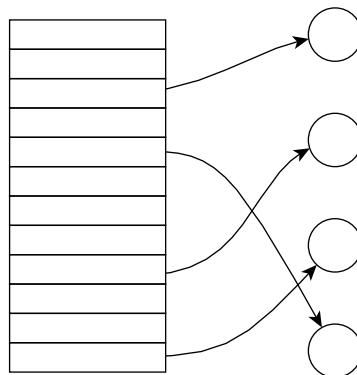
При сегментном преобразовании необходимо проверять, не вышел ли процесс за собственное адресное пространство.



### 7.1.1 Типы организаций таблиц сегментов

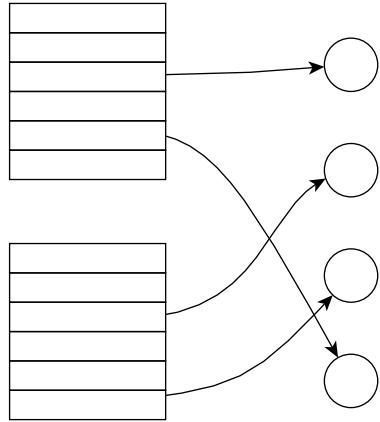
1. **Единая таблица** – единая таблица содержит все дескрипторы, единственное имя сегмента является индексом в таблице. Дескриптор содержит список прав доступа каждого пользователя сегмента.

Неудобный способ, особенно при совместном использовании, так как все используют единственное имя.



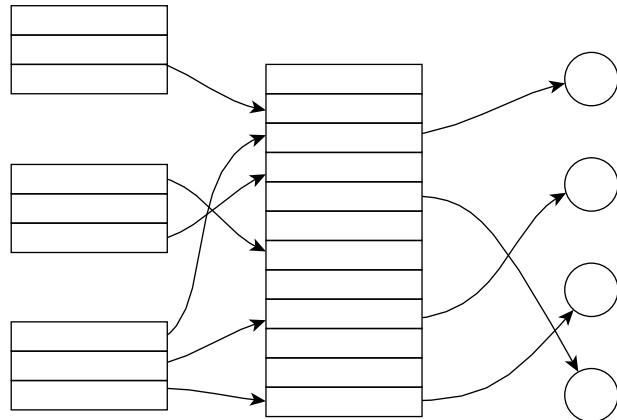
2. **Локальные таблицы** – каждая локальная таблица описывает адресное пространство (определяет среду) отдельных процессов.

В конкретном адресном пространстве имя сегмента является индексом в локальной таблице дескрипторов. Сегмент, доступный в нескольких адресных пространствах, будет иметь несколько имен и несколько дескрипторов.



### 3. Смешанная

Каждый сегмент имеет главный (центральный) дескриптор, который содержит все характеристики его физического размещения (размер, начальный адрес, флаги). Кроме главного дескриптора сегмент имеет локальные дескрипторы в тех виртуальных адресных пространствах, где он доступен. Такой локальный дескриптор содержит информацию, относящуюся к данному процессу (например, права доступа) и содержит указатель на глобальный дескриптор.



### Проблемы

- Сегмент – единица логического деления, то есть все сегменты имеют уникальные размеры, поэтому замещение сегмента на сегмент невозможно. Для замещение нужно использовать алгоритмы для страниц.
- Фрагментация

Сегмент хорошо обеспечивает коллективное использование.

#### 7.1.2 Алгоритмы, используемые при замещении сегментов:

Если какой-то сегмент нужно загрузить, значит один вытолкнуть.

1. Выталкивание случайного сегмента. Для замены выбирается любой случайный сегмент.
  - Малые накладные расходы
  - Может быть вытолкнут часто используемый или только что загруженный
2. Алгоритм FIFO: вытеснение сегмента, который дольше всего находится в памяти. Способы: временная метка (вытеснение с меньшей временной меткой) либо ведется связный список.  
Этот алгоритм исключает возможность выгрузки только что загруженного сегмента, но не исключает выгрузку часто использующегося.
3. Алгоритм LRU (Least Recently Used): замещаем наименее используемый сегмент.  
Строится на эвристическом предположении, правиле, если некоторые страницы имеют большое число обращений, то следующее обращение будет к ним.

Необходимо или изменять временную метку при каждом обращении, или при каждом обращении помещать этот сегмент в начало связного списка.

Наиболее затратно. (большие накладные расходы)

#### 4. Алгоритм NUR (Not Used Recently):

Колossalные накладные расходы LRU привели к использованию аппроксимирующих его алгоритмов.

Приписывание каждому сегменту бита обращения.

- При добавлении бит обращения 0.
- Периодически все биты обращений сбрасываются в 0.
- При обращении к сегменту устанавливается в 1.

Когда нужно заместить, ищется сегмент с нулевым битом обращения.

Кроме бита обращения вводится бит модификации.

Выгоднее вытеснять не модифицированный сегмент, потому что его копия лежит на диске и не надо выполнять точное копирование.

#### 5. Алгоритм LFU (Least Frequency Used): Наименее часто используемый сегмент.

При переполнении счетчик сбрасывается. Может быть вытеснен только что загруженный сегмент, не набравший число обращений.

**Глобальное вытеснение** – ищется среди всех процессов.

**Локальное вытеснение** – ищется среди множества сегментов процесса.

Вытеснение до страничного прерывания – обращение к несуществующей странице.

### 7.2 Управление памятью сегментами по запросам в архитектуре X86. Тип организации таблиц сегментов. Формат дескриптора сегмента в таблицах дескрипторов сегментов (GDT и LDT) (код и заполнение дескрипторов GDT из лабораторной работы по защищенному режиму).

В архитектуре x86 три вида системных таблиц:

- GDT (глобальная дескрипторная таблица)
- LDT (локальная)
- IDT (таблица векторов прерываний)

Для доступа к дескрипторам используются селекторы, записываются в сегментные регистры.

Сегментный регистр:

Index	2	1	0
15		0	

Индекс – смещение.

Первые 3 разряда в индексации не участвуют.

01 – RPL requested privilege level – запрошенный уровень привилегий.

2 – TI table indicator

Если равен 0 – глобальной, 1 – локальной

**Формат дескриптора** – размер 8 байт.

7	6	5	4	3	2	1	0
base_h	атрибуты	атрибуты	base_m	base_l	limit (размер сегмента)		

Опишем 6 и 5 байты (32 разряда):

7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0
G	D		lim		P	DPL	S	тип	A						

$2^{20}$  – размер шины данных в реальном времени. 1 Мб.

G – бит гранулярности, если он сброшен, то память в байтах (не больше 1 Мб), если установлен, то память меряется страницами (по 4 Кб).

A – access, бит доступа, устанавливается аппаратно при обращении к сегменту если было обращение – 1, нет – 0.

тип: 1 – для сегмента кода определяет если 0, то чтение из сегмента запрещено, это не касается выборки команд. Если 1 – то чтение из сегмента разрешено. Для сегмента данных если 0 – модификация данных в сегменте запрещена(запись), 1 – модификация разрешена.

тип 2: – для сегмента кода это бит подчинения, если 0 – то сегмент подчиненный, 1 – обычный. Для сегмента данных и стека определяет 0 – сегмент данных, 1 – сегмент стека.

тип 3: бит предназначения, если 0 – то это сегмент данных или стека, если 1 – то это сегмент кода.

бит 4 – S - system, если 0 то дескриптор описывает системный объект, если 1 то сегмент.

бит 5-6 – DPL - descriptor privilege level, уровень привилегий дескриптора (описывает segment).

бит 7 – бит присутствия (present), присутствует в памяти или нет.

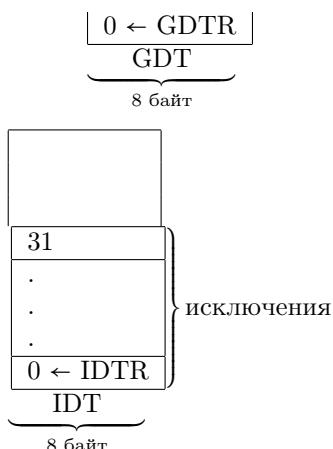
D - digit определяет разрядность операндов если 1, то операции 32х разрядные.

### 7.2.1 GDT и IDT

Глобальная дескрипторная таблица является общей для всех процессов. Её размер и расположение в физической памяти определяются регистром GDTR.

Особенностью GDT является то, что 0 дескриптор обязательно пустой, обращение запрещено.

Таблица прерываний IDT глобальна. Размещение в физической памяти определяется регистром IDTR.



### 7.2.2 Пример из лабораторной работы

```

descr struc ; Структура для описания дескриптора сегмента
    limit    dw 0      ; Граница (биты 0..15)
    base_l   dw 0      ; База, биты 0..15
    base_m   db 0      ; База, биты 16..23
    attr_1   db 0      ; Байт атрибутов 1
    arrt_2   db 0      ; Граница(биты 16..19) и атрибуты 2
    base_h   db 0      ; База, биты 24..31
descr ends

intr struc ; Структура для описания дескрипторов прерываний
    offs_l   dw 0      ; Смещение обработчика, нижняя часть (биты 0..15)
    sel      dw 0      ; Селектор сегмента команд
    rsrv    db 0      ; Зарезервировано
    attr    db 0      ; Атрибуты
    offs_h   dw 0      ; Смещение обработчика, верхняя часть (биты 16..31)
intr ends

GDT label byte ; Таблица глобальных дескрипторов
gdt_null descr <> ; Нулевой дескриптор
gdt_flatDS descr <0FFFFh,0,0,92h,0CFh,0>
                    ; переключение в 32-х битную модель памяти flat
                    ; с лимитом в 4 Гб и страницей адресацией
gdt_16bitCS descr <RM_seg_size-1,0,0,98h,0,0>
                    ; 16-битный 64-килобайтный сегмент кода с базой
                    ; RM_seg
gdt_32bitCS descr <PM_seg_size-1,0,0,98h,0CFh,0>
                    ; 32-битный 4-гигабайтный сегмент кода с базой PM_seg

```

```

gdt_32bitDS descr <PM_seg_size-1,0,0,92h,0CFh,0>
        ; 32-битный 4-гигабайтный сегмент данных с базой PM_seg
gdt_32bitSS descr <stack_l-1,0,0,92h,0CFh,0>
        ; 32-битный 4-гигабайтный сегмент данных с базой stack_seg
gdt_size = $-GDT ; размер нашей таблицы GDT+1байт (на саму метку)
; сегмент видеобуфера рассчитывается как смещение

gdtr dw gdt_size-1 ; Лимит GDT
dd ? ; Линейный адрес GDT

; Имена для селекторов
SEL_flatDS equ 8
SEL_16bitCS equ 16
SEL_32bitCS equ 24
SEL_32bitDS equ 32
SEL_32bitSS equ 40

IDT label byte ; Таблица дескрипторов прерываний IDT
trap1 intr 13 dup (<0, SEL_32bitCS,0, 8Fh, 0>)
        ; Первые 32 элемента таблицы (отведены под исключения)
trap13 intr <0, SEL_32bitCS,0, 8Fh, 0> ; Исключение общей защиты
trap2 intr 18 dup (<0, SEL_32bitCS,0, 8Fh, 0>)
        ; Первые 32 элемента таблицы (отведены под исключения)
int08 intr <0, SEL_32bitCS,0, 8Eh, 0> ; Дескриптор прерывания от таймера
int09 intr <0, SEL_32bitCS,0, 8Eh, 0> ; Дескриптор прерывания от клавиатуры
idt_size = $-IDT ; Размер IDT

idtr dw idt_size-1 ; Лимит IDT
dd ? ; Линейный адрес IDT

idtr_real dw 3FFh,0,0 ; содержимое регистра IDTR в реальном режиме

```

## 8 билет

### 8.1 Взаимоисключение и синхронизация процессов и потоков. Семафоры: определение, виды. Семафор, как средство синхронизации и передачи сообщений. Семафоры UNIX: примеры решения задач с помощью семафоров: «Производство-потребление» и «Читатели-писатели» в UNIX (пример реализации в лабораторной работе).

Чтобы не терять значение разделяемой переменной, необходимо обеспечить **монопольный доступ** процессов к **разделяемым переменным ресурсам** методами взаимоисключения. Только один процесс может получить доступ к данным в один момент. Текущие участки кода, в которых выполняется обращение к разделяемым данным называются **критическими**. Только один процесс может находиться в критическом участке.

Существуют следующие способы взаимоисключения:

1. Программный способ
2. Аппаратный способ
3. С помощью семафоров
4. С использованием мониторов

**Активное ожидание на процессоре** – ситуация, когда процесс занимает процессорное время, проверяя значение флага (занятости ресурса другим процессом). Активное ожидание на процессоре является неэффективным использованием процессорного времени.

Возможные варианты развития событий:

1. Возможно, что оба процесса пройдут цикл ожидания и попадут в свои критические секции.
2. Возможно бесконечное откладывание (зависание) – ситуация, когда разделённый ресурс снова захватывается тем же процессом.
3. Тупик (deadlock, взаимоблокировка) – ситуация, когда оба процесса установили флаги занятости и ждут. Т.е. каждый ожидает освобождения ресурса, занятого другим процессом.

#### 8.1.1 Семафоры

(ввел Дейкстра в 1965 г.)

Семафор – неотрицательная защищённая переменная S, над которой определено 2 неделимые операции: P (от датск. passeren - пропустить) и V (от датск. uitlegen - освободить). Переменная s является защищённой, значит изменять значение этой переменной могут только неделимые команды p и v.

**Основное свойство семафоров:** Одной неделимой операцией осуществляется выборка переменной s, инкремент и ее запоминание.

Если какая-либо операция не может быть выполнена над одним семафором, то неудачной считается операция над всеми семафорами.

Операция  $V(s) : s = s + 1$ , выполняется как одно неделимое действие

если  $s = 0$ , то операция V(s) может активизировать некоторый процесс блокированный на семафоре

операция  $P(s) : s = s - 1$ , процесс пытающийся войти в критическую секцию пытается декрементировать семафор

если  $s = 0$ , то декремент невозможен и процесс переводится в состояние ожидания (блокировки) до тех пор, пока другой процесс не освободит ресурс (не выйдет из своего критического участка).

Семафоры исключают бесконечное ожидание на процессоре, но платой за это является переход в режим ядра, т.е. команды определенные на семафоре являются системными вызовами, зато исключается активное ожидание на процессоре.

Процесс может создать семафор и изменять его. Удалить семафор может только процесс, создавший его, либо привилегированный процесс. Осуществляется переход в режим ядра при захвате и освобождении семафора, т.е. происходит переключение контекста.

- Если семафор может принимать 0 и 1 то называется бинарным.
- Если может принимать неотрицательные целые значения, то считающий.
- Множественные семафоры – массивы считающих семафоров (могут работать и как бинарные). Одной неделимой операцией можно изменить все или часть семафоров набора.

**Способы взаимодействий:**

## 1. Взаимоисключени, организация монопольного доступа процесса к разделяемой переменной.

Чистое взаимоисключение реализуется в задаче читателя-писателя осуществляется монопольный доступ писателя.

## 2. Синхронизация, когда процесс заинтересован в действиях другого процесса.

Задача производства потребления, если потребитель работает быстрее производителя возникнет ситуация когда буфер пуст, потребитель будет ожидать когда производитель положит что-нибудь в буфер, это видно при передаче сообщений, сообщения несут информацию которая интересует процесс, для того чтобы продолжить свое выполнение.

### Производство-потребление

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
#include <sys/stat.h>
#include <stdio.h>
#include <sys/shm.h>
#include <signal.h>

#define COUNT 3
#define PRODUCER 0
#define CONSUMER 1

#define PERMS S_IRWXU | S_IRWXG | S_IROTH

#define EMPTYCOUNT 0
#define FULLCOUNT 1
#define BIN 2

int semaphore;
int shared_memory;
char **addr_shared_memory;

// Массив структур
struct sembuf producer_grab[2] = { {EMPTYCOUNT, -1, SEM_UNDO}, {BIN, -1, SEM_UNDO} };
struct sembuf producer_free[2] = { {BIN, 1, SEM_UNDO}, {FULLCOUNT, 1, SEM_UNDO} };
struct sembuf consumer_grab[2] = { {FULLCOUNT, -1, SEM_UNDO}, {BIN, -1, SEM_UNDO} };
struct sembuf consumer_free[2] = { {BIN, 1, SEM_UNDO}, {EMPTYCOUNT, 1, SEM_UNDO} };

// Потребитель
void consumer(int semaphore, int value)
{
    while(1)
    {
        sleep(1);
        int sem_op_p = semop(semaphore, consumer_grab, 2);
        if (sem_op_p == -1)
        {
            perror("Can't semop \\n");
            exit(1);
        }

        if (((char*)(*addr_shared_memory + sizeof(int *))) == ((char*)(addr_shared_memory) + 2 * sizeof(int *) + 5 * sizeof(int)))
            *(addr_shared_memory + sizeof(int *)) = (char*)addr_shared_memory + 2 * sizeof(int *);

        printf("Consumer%d get %d\\n", value, *((addr_shared_memory + sizeof(int *))));
        *((addr_shared_memory + sizeof(int *)))++;
    }
}
```

```

    int sem_op_v = semop(semaphore, consumer_free, 2);
    if (sem_op_v == -1)
    {
        perror("Can't semop \n");
        exit(1);
    }
}

// Производитель
void producer(int semaphore, int value)
{
    while(1)
    {
        sleep(2);
        int sem_op_p = semop(semaphore, producer_grab, 2);
        if (sem_op_p == -1)
        {
            perror("Can't semop \\n");
            exit(1);
        }

        if ((*addr_shared_memory) == ((char*)(addr_shared_memory) + 2 * sizeof
            (int *) + 5 * sizeof(int)))
            (*addr_shared_memory) = (char*)addr_shared_memory + 2 * sizeof(int *);

        *((*addr_shared_memory) = ((char*)(*addr_shared_memory) - (char*)
            addr_shared_memory) - 16;
        printf("Producer%d put %d\n", value, *((*addr_shared_memory)));
        (*addr_shared_memory)++;

        int sem_op_v = semop(semaphore, producer_free, 2);
        if (sem_op_v == -1)
        {
            perror("Can't semop \n");
            exit(1);
        }
    }
}

int main()
{
    int process;
    int consumers[3];
    int producers[3];
    // Создание семафора
    semaphore = semget(IPC_PRIVATE, 3, IPC_CREAT | PERMS);
    int se = semctl(sem, EMPTYCOUNT, SETVAL, COUNT);
    int sf = semctl(sem, FULLCOUNT, SETVAL, 0);
    int sb = semctl(sem, BIN, SETVAL, 1);
    // Объявление разделяемого сегмента
    shared_memory = shmat(IPC_PRIVATE, 2 * sizeof(int *) + 5 * sizeof(int),
        IPC_CREAT | PERMS);
    addr_shared_memory = shmat(shared_memory, 0, 0);
    *((addr_shared_memory) = (char*)addr_shared_memory + 2 * sizeof(int *));
    *((addr_shared_memory + sizeof(int *)) = (char*)addr_shared_memory + 2 * sizeof(
        int *));

    // Создание процессов
    for (int i = 0; i < COUNT; i++) {
        if (-1 == (producers[i] = fork()))
        {

```

```

        return 1;
    }
    else if (0 == producers[i])
    {
        producer(semaphore, i);
        exit(0);
    }

    if (-1 == (consumers[i] = fork())))
    {
        return 1;
    }
    else if (0 == consumers[i])
    {
        consumer(semaphore, i);
        exit(0);
    }
}

signal(SIGINT, catch_sigp);
int status;
wait(&status);

// Очистка памяти
shmctl(shared_memory, IPC_RMID, NULL);
semctl(semaphore, 0, IPC_RMID, 0);
return 0;
}

```

### Читатели-писатели

```

#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
#include <sys/stat.h>
#include <stdio.h>
#include <sys/shm.h>
#include <signal.h>

#define WRITER 3
#define READER 5

#define PERMS S_IRWXU | S_IRWXG | S_IRWXO

// количество читателей
#define READERCOUNT 0
// активные писатели
#define ACTIVEWRITER 1
// очередь писатей
#define WRITERCOUNT 2
// очередь читателей
#define READERWAIT 3

int semaphore;
int shared_memory;
int *addr_shared_memory;

// Массив структур
struct sembuf start_read[] = { {READERWAIT, 1, SEM_UNDO}, {ACTIVEWRITER, 0,
    SEM_UNDO}, {WRITERCOUNT, 0, SEM_UNDO}, {READERCOUNT, 1, SEM_UNDO}, {READERWAIT
    , -1, SEM_UNDO} };
struct sembuf stop_read[] = { {READERCOUNT, -1, SEM_UNDO} };
struct sembuf start_write[] = { {WRITERCOUNT, 1, SEM_UNDO}, {READERCOUNT, 0,
    SEM_UNDO} };

```

```

    SEM_UNDO}, {ACTIVEWRITER, -1, SEM_UNDO}, {WRITERCOUNT, -1, SEM_UNDO}, };
struct sembuf stop_write[] = { {ACTIVEWRITER, 1, SEM_UNDO} };

// собственный обработчик сигнала ctrl-c
void catch_sigp(int sig_num)
{
    signal(sig_num, catch_sigp);
    shmctl(shared_memory, IPC_RMID, NULL);
    semctl(semaphore, 0, IPC_RMID, 0);
}

void StartRead()
{
    semop(semaphore, start_read, 3);
}

void StopRead()
{
    semop(semaphore, stop_read, 1);
}

void StartWrite()
{
    semop(semaphore, start_write, 4);
}

void StopWrite()
{
    semop(semaphore, stop_write, 1);
}

// Читатель
void Reader(int value)
{
    while (1)
    {
        StartRead();
        printf("Reader%d get = %d\n", value, *addr_shared_memory);
        StopRead();
        sleep(1);
    }
}

// Писатель
void Writer(int value)
{
    while (1)
    {
        StartWrite();
        (*addr_shared_memory)++;
        printf("Writer%d put = %d\n", value, *addr_shared_memory);
        StopWrite();
        sleep(2);
    }
}

int main()
{
    // Создание семафора
    semaphore = semget(IPC_PRIVATE, 4, IPC_CREAT | PERMS);
    int sr = semctl(sem, READERCOUNT, SETVAL, 0);
    int sa = semctl(sem, ACTIVEWRITER, SETVAL, 1);
    int sw = semctl(sem, WRITERCOUNT, SETVAL, 0);
}

```

```

// Объявление разделяемого сегмента
shared_memory = (int*)shmget(IPC_PRIVATE, sizeof(int), IPC_CREAT | PERMS);
addr_shared_memory = shmat(shared_memory, 0, 0);

// Создание процессов
int processes[READER + WRITER];
int parent = getpid();

for (int i = 0; i < WRITER + READER; i++)
{
    processes[i] = fork();
    if (getpid() != parent)
    {
        for (int j = 0; j < i; j++)
            processes[j] = -1;
        break;
    }
}

for (int i = 0; i < WRITER + READER; i++)
{
    if (processes[i] != 0)
        continue;

    if (i < WRITER)
    {
        Writer(i);
        return 0;
    }
    else
    {
        Reader(i - WRITER);
        return 0;
    }
}

signal(SIGINT, catch_sigp);
int status;
wait(&status);

// Очистка памяти
shmctl(shared_memory, IPC_RMID, NULL);
semctl(semaphore, 0, IPC_RMID, 0);
return 0;
}

```

## 8.2 Аппаратные прерывания: задачи обработчика прерываний от системного таймера в защищенном режиме.

Аппаратные прерывания – прерывания от устройств.

Т.к. некоторые из задач не требуют выполнения на каждом тике, то вводится понятие основного тика (равен n тикам), часть задач выполняется только при основном тике.

### 8.2.1 Функции обработчика прерываний от системного таймера в Windows

По тику

1. Обновление системного времени (инкремент);
2. Декремент показания счетчика, отслеживающего продолжительность работы текущего потока;
3. Декремент показаний счетчиков отложенных задач.

По главному тику

1. Инициализация диспетчера настройки баланса, который активизируется каждую секунду для возможной инициации событий, связанных с планированием и управлением памятью;

#### **По кванту**

1. Инициализация диспетчеризации потоков (добавление соответствующего объекта DPC в очередь).

### **8.2.2 Функции обработчика прерываний от системного таймера в Unix/Linux**

#### **По тику**

1. Обновление статистики использования процессора текущим процессом;
2. Обновление часов и других таймеров системы;
3. Обработка отложенных вызовов;
4. Ведение счета тиков аппаратного таймера по необходимости.

#### **По главному тику**

1. Выполнение функций, относящихся к работе планировщика, таких как пересчет приоритетов и действия, выполняющиеся по истечении выделенного кванта времени;
2. Пробуждение в нужные моменты системных процессов, такие как swapper и pagedaemon;
3. Обработка сигнала тревоги (будильники).

#### **По кванту**

1. Посылка текущему процессу сигнала SIGXCPU, если тот превысил выделенную ему квоту(квант) использования процессора (процессорного времени).



## 9.1 Виртуальная память: управление памятью страницами по запросу – три схемы преобразования; реализация страничного преобразования в компьютерах на базе процессоров Intel (x86): стандартное преобразование и РАЕ в защищном режиме – схемы, размеры таблиц и их количество на каждом этапе преобразования.

**Виртуальная память** – память, размер которой превышает размер реального физического пространства. Используется адресное пространство диска как область свопинга или пейджинга, т.е. для временного хранения областей памяти.

Подходы к реализации управления виртуальной памятью:

1. управление памятью страницами по запросам
2. сегментами по запросам
3. сегментами, деленными на страницы по запросам

**Страница** – является единицей физического деления памяти. Её размер устанавливается системой. У страницы размер не обязательно 4Кб. Просто такой размер оптимальен по количеству страничных прерываний.

**Сегмент** – является единицей логического деления памяти. Её размер определяется объемом кода. Любой сегмент создается такого размера, который требуется программе.

**Запрос** – страничное прерывание, которое возникает при попытке доступа к незагруженной в память странице. При его обработке происходит пэйджинг.

**Пэйджинг** – загрузка с диска новых и замещение старых страниц.

**Методы преобразования адресов:**

1. Прямое отображение таблиц страниц.

Каждый процесс должен иметь собственную таблицу страниц.

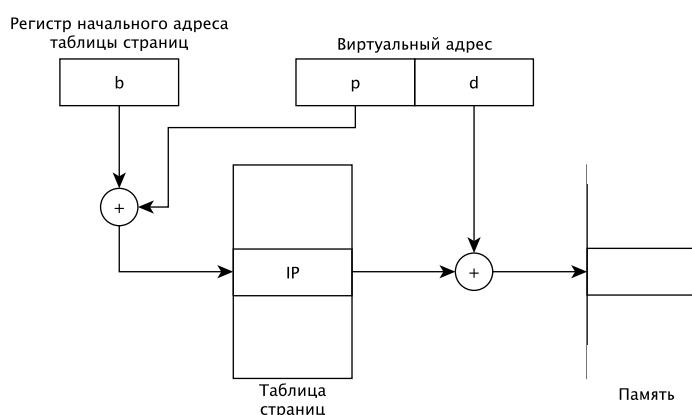
Главная таблица – таблица (современные ОС оперируют связными списками) процессов.

Таблица состоит из дескрипторов процессов.

В прямом отображении таблиц страниц – системные таблицы хранятся в оперативной памяти в системной области.

В процессоре должен быть регистр начального адреса таблицы страниц.

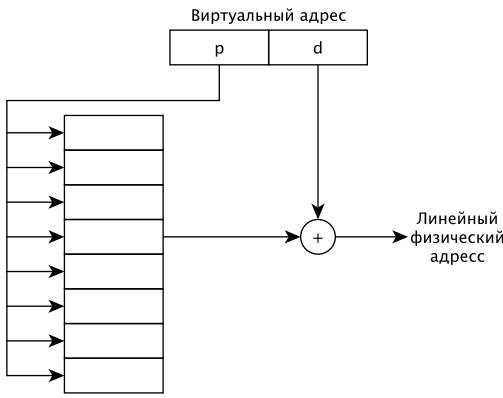
Если страница загружена в физическую память, то она имеет физический адрес.



Преобразование поддерживается аппаратно.

2. Ассоциативное отображение (для сокращения обращений к памяти).

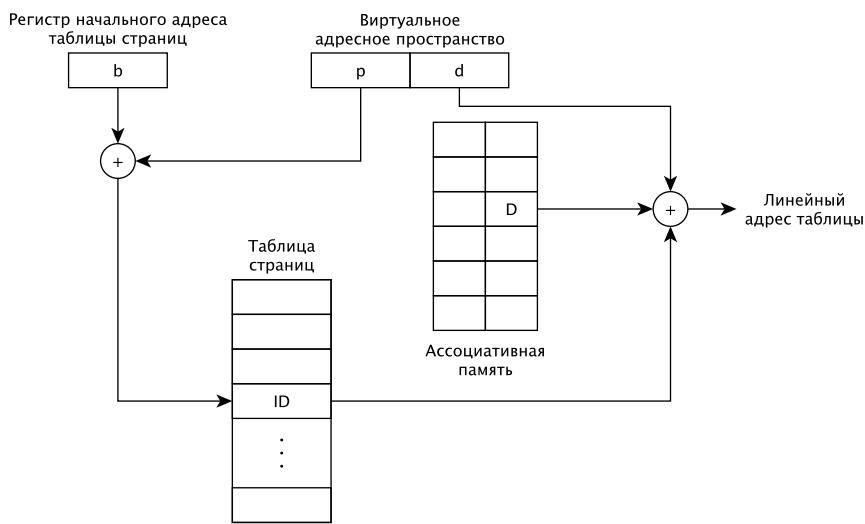
Использование ассоциативной памяти – память, которая обеспечивает выборку по ключу за 1 такт.



Дорогая память, всегда регистровая количества схем удваивается, много соединений.

Практически не используется.

### 3. Ассоциативно прямое отображение.



Ассоциативный кэш (последние обращения) – хранятся физические адреса страниц, используемых недавно.

Существует небольшая по объему ассоциативная память. (в современных ОС)

Эвристическое предположение: Если было обращение к странице, то следующее будет к той же странице.

Системы обеспечивают скоростные показатели 90% и более в отличие от полностью с прямым отображением.

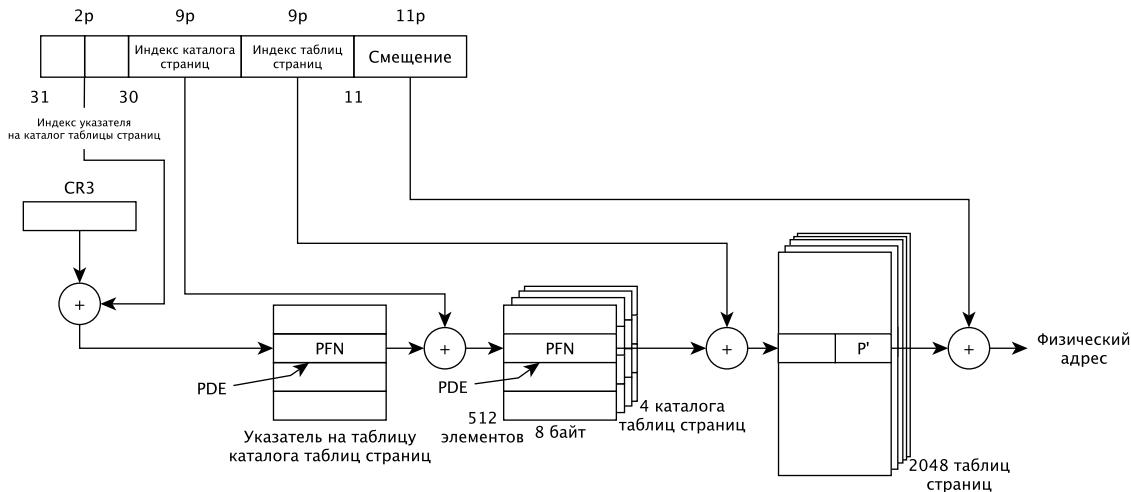
Начиная с Pentium Pro включен регистр CR4, где 5 бит это PAE (physical address extension) – расширение физического адреса.

В режиме PAE виртуальный адрес делится на 4 поля.

Если PAE = 1, то разрешено использование расширенной 36рр, вместо 32 pp физического адреса. При этом физический адрес остается 32 pp, все изменения касаются только работы страничного механизма.

PFN – Page Frame Number

PDE – Page Describe Entry



## 9.2 Unix: концепция процессов; иерархия процессов, процессы «сироты», процессы «зомби», демоны; примеры из лабораторной работы (5 программ).

Процесс в Unix является базовый. Каждый процесс имеет собственное виртуальное пространство, в котором находятся все сегменты процесса, код, данные, стек. При этом с точки зрения Unix процесс часть времени выполняет собственный код и тогда он выполняется в режиме задачи (user mode), а часть времени код выполняет реентерабельный код ОС и тогда это режим ядра (kernel mode).

В Unix любой процесс создаётся системным вызовом fork. Любой процесс может создать любое кол-во процессов (вызовов fork).

В результате создаётся иерархия процессов в отношении предок-потомок (parent-child).

В UNIX процесс, все его дочерние процессы и более отдаленные потомки образуют группу процессов. Когда пользователь отправляет сигнал с клавиатуры, тот достигает всех участников этой группы процессов, связанных на тот момент времени с клавиатурой (обычно это все действующие процессы, которые были созданы в текущем окне). Каждый процесс по отдельности может захватить сигнал, игнорировать его или совершить действие по умолчанию, которое должно быть уничтожено сигналом.

В качестве другого примера, поясняющего ту роль, которую играет иерархия процессов, давайте рассмотрим, как UNIX инициализирует саму себя при запуске. В загрузочном образе присутствует специальный процесс, называемый init. В начале своей работы он считывает файл, сообщающий о количестве терминалов. Затем он разветвляется, порождая по одному процессу на каждый терминал. Эти процессы ждут, пока кто-нибудь не зарегистрируется в системе. Если регистрация проходит успешно, процесс регистрации порождает оболочку для приема команд. Эти команды могут породить другие процессы, и т. д. Таким образом, все процессы во всей системе принадлежат единому дереву, в корне которого находится процесс init.

**Процессы демоны** – процессы которые не имеют родителей, они существуют сами и не входят ни в какие группы. Демон – процесс, выполняющий какую-то фоновую задачу, не имеющий управляющего терминала и, как следствие, обычно не интерактивный по отношению к пользователю.

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

#define ANSI_COLOR_GREEN "\x1b[32m"
#define ANSI_COLOR_RESET "\x1b[0m"

int main()
{
    int status;
    pid_t childpid1, childpid2;

    childpid1 = fork();

    if (childpid1 == -1)
    {
        perror("Can't fork.\n");
    }
}
```

```

        return 1;
    }
    else if (childpid1 == 0)
    {
        if (execl("/bin/ls", "ls", "-lah", 0) == -1)
        {
            perror("Can't exec.\n");
            exit(1);
        }
    }

    childpid2 = fork();

    if (childpid2 == -1)
    {
        perror("Can't fork.\n");
        return 1;
    }
    else if (childpid2 == 0)
    {
        if (execl("/bin/cat", "cat", "makefile", 0) == -1)
        {
            perror("Can't exec.\n");
            exit(1);
        }
    }
    else
    {
        wait(&status);

        if (WIFEXITED(status)) printf(ANSI_COLOR_GREEN "child process exit
            success\n" ANSI_COLOR_RESET);
    }
}

return 0;
}

```

**Процесс-сирота** – процесс, у которого завершился предок.

Предок завершился, если не предпринять действий иерархия будет нарушена. Процесс сирота усыновляется терминальным процессом.

При завершении любого процесса система проверяет не осталось ли у него незавершенных потомков (по дескриптору процесса, где есть указатели на потомков). Если такие остались, то выполняется процесс усыновления, фактически изменения указатели: процесс-потомок получает указатель на нового процесса-предка.

```

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

int main()
{
    pid_t childpid1, childpid2;

    childpid1 = fork();

    if (childpid1 == -1)
    {
        perror("Can't fork.\n");
        return 1;
    }
    else if (childpid1 == 0)
    {
        printf("CHILD1\npid: %d; ppid: %d; gid: %d\n\n", getpid(), getppid(),
            getgid());
        getchar();
        printf("CHILD1\npid: %d; ppid: %d; gid: %d\n\n", getpid(), getppid(),

```

```
        getgid());
    return 0;
}

childpid2 = fork();

if (childpid2 == -1)
{
    perror("Can't fork.\n");
    return 1;
}
else if (childpid2 == 0)
{
    printf("CHILD2\npid: %d; ppid: %d; gid: %d\n\n", getpid(), getppid(),
           getgid());
    getchar();
    printf("CHILD2\npid: %d; ppid: %d; gid: %d\n\n", getpid(), getppid(),
           getgid());
    return 0;
}
else
{
    printf("PARENT\npid: %d; ppid: %d; gid: %d\n\n", getpid(), getppid(),
           getgid());
    getchar();
}

return 0;
}
```

10 билет

file answers.pdf

## 11 билет

**11.1 Параллельные процессы: взаимодействие, обоснование необходимости монопольного доступа к разделяемым переменным, способы взаимоисключения.**  
**Мониторы: определение; примеры – простой монитор и монитор кольцевой буфер.**

Чтобы не терять значение разделяемой переменной, необходимо обеспечить **монопольный доступ** процессов к **разделяемым переменным ресурсам** методами взаимоисключения. Только один процесс может получить доступ к данным в один момент. Те участки кода, в которых выполняется обращение к разделяемым данным называются **критическими**. Только один процесс может находиться в критическом участке.

Существуют следующие способы взаимоисключения:

1. Программный способ
2. Аппаратный способ
3. С помощью семафоров
4. С использованием мониторов

**Активное ожидание на процессоре** – ситуация, когда процесс занимает процессорное время, проверяя значение флага (занятости ресурса другим процессом). Активное ожидание на процессоре является неэффективным использованием процессорного времени.

Возможные варианты развития событий:

1. Возможно, что оба процесса пройдут цикл ожидания и попадут в свои критические секции.
2. Возможно бесконечное откладывание (зависание) – ситуация, когда разделённый ресурс снова захватывается тем же процессом.
3. Тупик (deadlock, взаимоблокировка) – ситуация, когда оба процесса установили флаги занятости и ждут. Т.е. каждый ожидает освобождения ресурса, занятого другим процессом.

### Мониторы

Проблема связанная с семафорами и аналогичными средствами которые дают ОС состоит в том, что они не структурные, программа может утратить блокировку и синхронизацию.

Чтобы устранить эту проблему была придумана концепция мониторов, которая была реализована на нескольких языках программирования.

Мониторы придуманы для структурирования средств взаимоисключения. В результате синхронизация и связь выполняется в мониторе.

Потенциальные ошибки программирования связаны с программированием самого монитора. Их легче выявлять.

Отрицательные стороны: на обращение к функциям монитора тратится больше времени.

Идея монитора заключается в создании механизма, который унифицировал бы создание параллельных процессов по данным и функциям, которые обрабатывают эти данные.

Монитор – это языковая конструкция состоящая из структуры данных и набора подпрограмм, которые могут обращаться к полям этой структуры.

Мониторы обозначаются ключевым словом monitor.

Монитор защищает свои переменные, так как доступ к полям мониторов может быть осуществлён только с помощью подпрограммы (функции) монитора.

На мониторах чаще всего определяется переменная типа условия. Используются две функции wait и signal.

### Простой монитор

Для выделение единственного ресурса n-параллельным процессам или предназначен для обеспечения взаимоисключающего доступа к одной переменной.

Сам монитор является ресурсом. Процессам, которым удалось вызвать функцию монитора, говорят что находятся в мониторе, остальные выстраиваются в очередь к монитору.

```
resource: monitor;
var
    busy: logical;
    x: conditional;

procedure require
begin
    if busy then
        wait(X);
```

```

    busy:=true;
end;

procedure release
begin
    busy:=false;
    signal(x);
end;

begin
    busy:=false;
end.

Кольцевой буфер

Решает задачу производства-потребления. Характерно наличие процессов производителей и процессов потребителей. Производитель помещает данные в массив, а потребитель берет и они перестают существовать.

resource: monitor;
var
    bcircle:array[0...n-1] of <type>;
    pos:0...n;
    j:0...n-1;
    k:0...n-1;
    bufferfull, bufferempty:conditional;

procedure producer(data:<type>)
begin
    if pos = n then
        wait(bufferempty);
    bcircle[j] := data;
    pos:=pos+1;
    j:=(j+1)mod n;
    signal(bufferfull);
end;

procedure consumer(var data <type>)
begin
    if pos = 0 then
        wait(bufferfull);
    data := bcircle[j];
    pos:=pos-1;
    k:=(k+1)mod n;
    signal(bufferempty);
end;

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

```

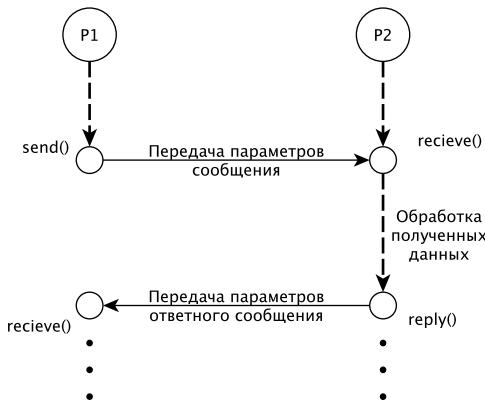
## 11.2 Средства межпроцессорного взаимодействия (IPC) операционной системы UNIX System V: очереди сообщений и программные каналы – сравнение, примеры (для программных каналов пример из лабораторной работы с сигналами).

Набор средств взаимодействия параллельных процессов в Unix System V (IPC):

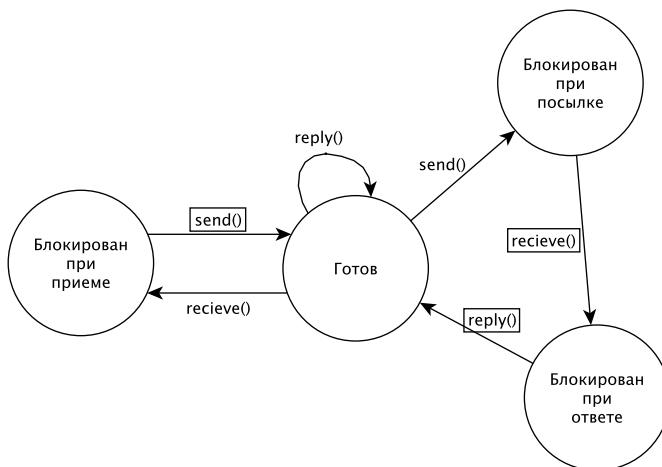
- Семафоры
- Сигналы
- Программные каналы (именованные и неименованные)
- Разделяемая память

## Очереди сообщений

Системные вызовы типа send и receive. Сообщения это специальный вид информации. Они обычно пакуются и это называется пакетом. Сам системный вызов send означает, что мы посылаем сообщение, receive означает получение сообщения. При этом возникает три состояния блокировки при передаче сообщений. Рассмотрим временную диаграмму при передаче сообщений.



## Три состояния блокировки



Система поддерживает очереди сообщений с помощью системных таблиц.

Каждый элемент такой таблицы описывает очередь сообщений.

```
struct msg_buf
{
    long msgtype; // тип
    char msgtext[MSGMAX]; // текст и размер
```

На очередях сообщений определены следующие системные вызовы или API:

```
msgget();
msgctl();
msgsnd();
msgrcv();
```

Когда процесс передает сообщение в очередь, ядро создает для него новую запись и помещает ее в конец связного списка записей, указанной очереди, идентификатор которой известен. В каждой такой записи указывается: тип сообщения, размер в байтах и указатель на область данных ядра, где фактически находится сообщение.

Ядро копирует сообщение из пространства пользователя в адресное пространство ядра, чтобы процесс отправивший сообщений мог завершиться. Сообщение остается доступным другим процессам.

Когда какой-то процесс выбирает сообщение из очереди ядро копирует это сообщение из своего адресного пространства в адресное пространство процесса-получателя (двойное копирование).

После того, как сообщение получено, оно перестает существовать (потребляемый ресурс).

Процесс может выбрать сообщение из очереди следующими способами:

1. Взять самое старое сообщение, независимо от его типа.

2. Если идентификатор сообщения совпадает с идентификатором, который создал процесс. Если существует несколько с одинаковым идентификатором, то выбирается самое старое.
3. Выбрать сообщение, числовое значение типа которого меньшее или равное значению типа, указанному процессом.

Таким образом, мы видим что данное средство позволяет процессам не блокироваться при передаче посылки, а также при приеме.

Процесс, отправивший сообщение, если незаинтересован в ответе, может выполняться дальше или быть завершен.

Тип сообщения позволяет разделять сообщения от клиента к серверу и от сервера к клиенту.

### **Программные каналы**

Именованные программные каналы создаются командой mknode.

Программный канал - это специальный файл, в который можно «писать» информацию и из которого эту информацию можно «читать».

Причем порядок записи информации и последующего чтения – FIFO (очередь).

Именованный канал имеет имя, которое указывается при вызове команды mknode [опции] <имя> p.

Использовать именованный канал может любой процесс, «знающий» имя канала.

Системный вызов pipe() создает неименованный программный канал. Неименованные программные каналы могут использоваться для обмена сообщениями между процессами родственниками. В отличие от именованных программных каналов неименованные не имеют идентификатора, но имеют дескриптор.

Процесс-потомок наследует все дескрипторы открытых файлов процесса-предка, в том числе и неименованных программных каналов.

Программные каналы имеют встроенные средства взаимоисключения: из канала нельзя читать, если в него пишут, и в канал нельзя писать, если из него читают.

Программный канал – буфер в системной области памяти. (последовательность адресов)

Программный канал небольше страницы.

Создаются в системной области памяти, так как процесс не может обращаться за собственное адресное пространство.

Pipe буфферизируется на 3 уровнях:

Трубы буфферизуются в системную область памяти, в системе может существовать определенное кол-во труб, если при попытке создать трубу число будет превышать, то буфера имеющие наибольшее время существования будут переписаны на диск.

При этом будут использоваться стандартные функции работы с файлами.

Если процесс пытается записать в программный канал больше 4096 байт, то труба буфферизуется во времени, до тех пор пока данные не будут из нее прочитаны.

Другими словами подсистема ввода-вывод обеспечивает приостановку процесса, если канал заполнен.

Команда write будет успешно выполнена, если есть доступное место, в противном случае ожидание.

Команда read процесс будет блокирован если канал пуст.

Ограничение размера объясняется стремлением повысить эффективность обмена засчет размещения канала в оперативной памяти, если канал большой (превышает размер страницы), то он частично лежит на диске, операции чтения и записи выполняются значительно медленнее.

#### **11.2.1 Пример из лабораторной работы**

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

#define ANSI_COLOR_GREEN "\x1b[32m"
#define ANSI_COLOR_RESET "\x1b[0m"

#define N 20

void call_signal_int()
{
    printf("Child exit\n");
    exit(0);
}
```



```
    close(channel1[1]);
    read(channel1[0], buffer, sizeof(buffer));
    printf("message1 = \%\s\n", buffer);

    close(channel2[1]);
    read(channel2[0], buffer, sizeof(buffer));
    printf("message2 = \%\s\n", buffer);
    return 0;
}

return 0;
}
```

## 12 билет

**12.1 ОС с монолитным ядром. Переключение в режим ядра. Диаграмма состояний процесса и переход из одного состояния в другое – причины каждого перехода. Диаграмма состояний процесса в UNIX. Переключение контекста. Система прерываний.**

**ОС с монолитным ядром** – это программа имеющая модульную структуру, то есть состоящая из подпрограмм. Системы Windows, Unix, Linux имеют монолитные ядра. Unix имеет минимизированное ядро (вынесен графический интерфейс).

Единственный способ изменить его конфигурацию, например, добавить или удалить компоненты в ядро, можно только путём его перекомпиляции.

Взаимодействие приложений с ядром выполняется с помощью системных вызовов. Все функции в иерархической машине и структуре Unix BSD выполняются монолитным ядром, то есть монолитное ядро включает в себя все.

Определяется состав системы прерываний.

Системные вызовы (API) функции ОС, которые предоставляются приложениям, для того чтобы приложения могли запрашивать функции системы. ОС переходит в режим ядра.

Исключения – ошибки, делятся на исправимые (страничное прерывание (обращение к странице адрес, которой отсутствует в физической памяти) и неисправимые (деление на 0, компьютер не может обрабатывать бесконечно большие и бесконечно малые числа)).

### Переключение в режим ядра.

Процесс – главная абстракция системы. UNIX декларирует следующим образом: процесс часть времени выполняет собственный код, и тогда он выполняется в режиме задачи, а часть времени выполняет реинтегрируемый код операционной системы, и тогда он выполняется в режиме ядра.

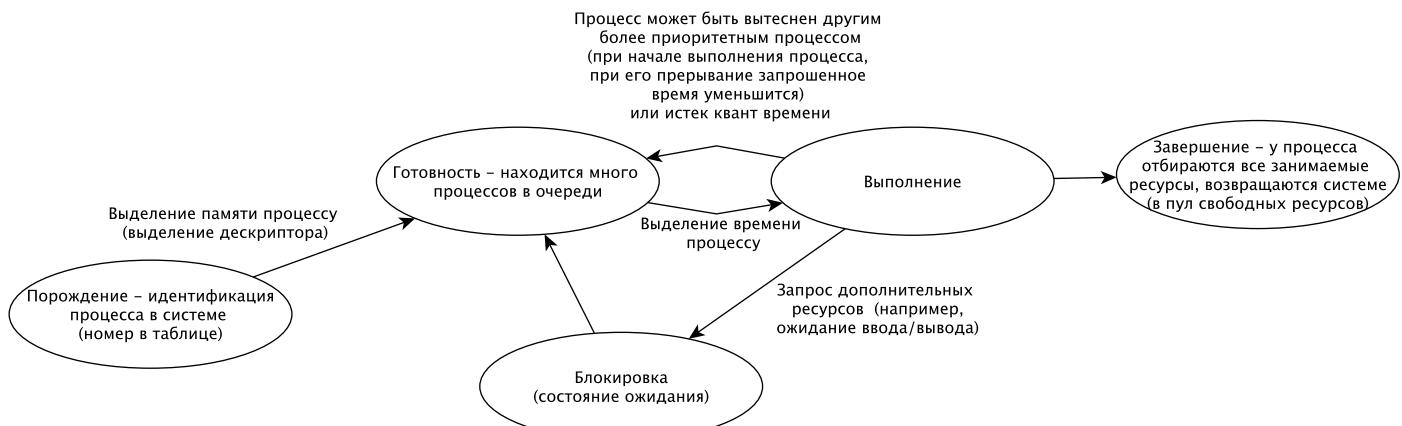
Системные вызовы – запрос процесса на системные ресурсы (ввод-вывод). Ни одна операционная система не позволяет процессу напрямую обратиться к устройствам ввода-вывода. Если бы обращались напрямую, то такая система была бы незащищенной. Для обслуживания системного вызова процесс должен перейти в режим ядра, в режиме ядра выполняется реинтегрируемый код операционной системы. Реинтегрируемый код – код чистой процедуры (чистая процедура не модифицирует саму себя, то есть вынесены все данные). Коды ядра работают с системными таблицами (коды ядра реинтегрируемы). Процедуры сами являются ресурсами, которые могут запрашивать процессы, разные процессы могут находиться в разных точках одной и той же процедуры. При выполнении системного вызова процесс переключается в режим ядра и выполняет реинтегрируемый код операционной системы. После того, как системный вызов обработан, процесс опять переключается в режим задачи и выполняет собственный код.

В режиме ядра процесс переносится в очередь готовых процессов, выполняют действия для конкретного процесса. Затем когда процесс получает квант, он начинает выполнять собственный код. Если процесс запросил дополнительный ресурс, то процесс переключится в режим ядра и будет блокирован, будет находиться в режиме блокировки пока процесс не получит нужный ему ресурс, после чего система поместит процесс в очередь готовых процессов и после получения кванта процесс может продолжить выполнение своего кода до конца.

### Три события, переводящих систему в режим ядра

1. Системные вызовы
2. Исключения
3. Аппаратные прерывания

### Диаграмма состояний процесса и переход из одного состояния в другое



## Переключение контекста

Аппаратный – это содержимое регистров процессора (РОНы, счетчики команд, адресные регистры). Переключение аппаратных контекстов поддерживается аппаратно: есть команда PUSHА. Ни одна система не позволяет оперировать процессу с устройствами напрямую, так как это сводит на нет любую защиту. На самом деле, read/write – системные вызовы. Они начинают выполняться в режиме пользователя (код программы), потом происходит переключение в режим ядра (реэнтериабельный код ОС) ==> произойдет переключение аппаратного контекста.

Полный – аппаратный + информация о выделенных процессу ресурсах: например, выделенная ему память. Память процесса – память, которая выделена процессу – описывается соответствующими таблицами. Очевидно, что когда происходит переключение процессов: он исчерпал свой квант или происходит системный вызов, то кроме аппаратного контекста необходимо иметь инфу о тех ресурсах, которые были выделены процессу, т. е происходит переключение полного контекста. Каждый процесс имеет свою таблицу памяти. Любой процесс имеет свои таблицы страниц. Переключение полного контекста – затратная операция, так как аппаратно она не поддерживается. Теряется т.н актуальность кешей. В связи с затратностью по времени возник интерес к потокам.

UNIX BSD → struct proc

UNIX → struct task\_struct

Вся эта информация нужна системе для того, чтобы иметь возможность для выполнения процесса.

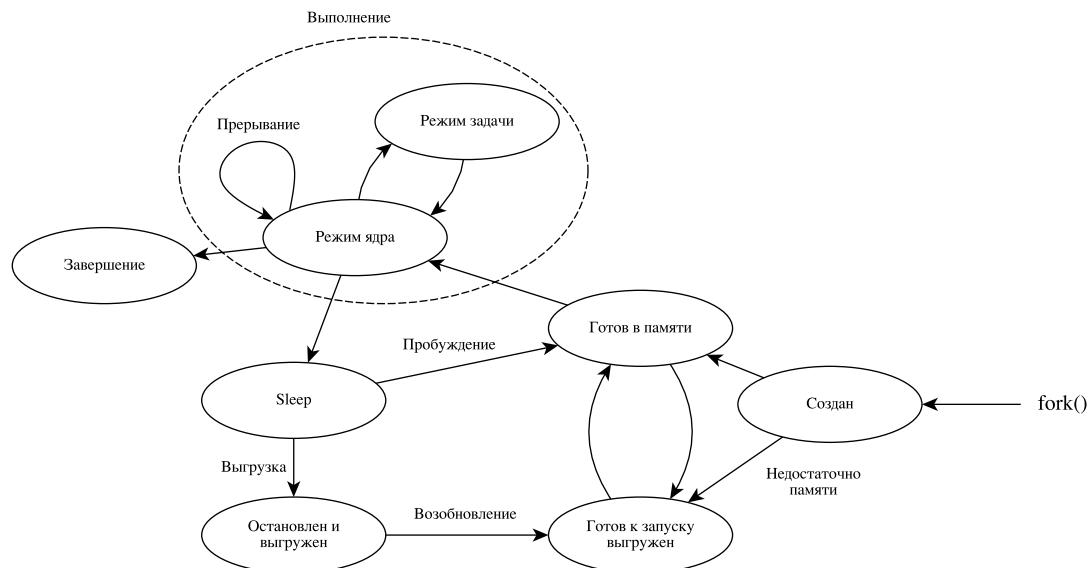
Понятие контекста является важнейшим в системе.

Полный контекст переключается тогда, когда процесс возвращается в готовность.

Когда переходит из режима ядра в режим задачи, переключается только аппаратный контекст.

Любой процесс вызывается системным вызовом Fork. Он создает новый процесс.

**Диаграмма состояний процесса в UNIX.**



## Система прерываний

### 12.1.1 3 типа прерываний (составляют систему прерываний):

#### 1. Системные вызовы (программные прерывания)

- вызов, когда требуется сервис системы (ввод/вывод, обращение к внешнему устройству)
- клавиатура, мышь, вторичная память (флешки, диски)
- ни одна система не позволяет напрямую процессом обращение к устройствам ввода-вывода, если разрешить такие обращения, то систему защитить невозможно.
- Система предоставляет соответствующие функции (примитивы, т. к. низкоуровневые (ядро) действия) (API)
- синхронные события.

#### 2. Исключительные ситуации

- прерывание выполняемой программы при возникновении исключения
- (исправимые – страничное прерывание; неисправимое – деление на 0)
- синхронные события.

#### 3. Аппаратные прерывания

- Бывают нескольких типов:
  - Прерывания системного таймера (важнейшие функции)
  - Прерывание от внешних устройств (возникают по завершению операции ввода/вывода)
  - Действия оператора (ctrl-alt-del)
- Асинхронные события, возникают независимо от каких-либо действий, которые выполняются в системе.

## 12.2 Задача: читатели-писатели – монитор Хоара, решение с использованием семафоров Unix и разделяемой памяти, пример реализации из лабораторной работы.

**Монитор читатели-писатели.**

2 типа процессов:

- Читатели – читают, поскольку они информацию не меняют, то они не мешают друг другу и могут читать параллельно.
- Писатель (только 1) – изменяет данные, в режиме взаимоисключения (читать и писать нельзя)

**Классический монитор Хоара.**

В нем описывается 4 функции: start\_read, stop\_read, start\_write, stop\_write.

В результате всех проверок исключается бесконечное откладывание читателей и писателей.

**Читатели-писатели**

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
#include <sys/stat.h>
#include <stdio.h>
#include <sys/shm.h>
#include <signal.h>

#define WRITER 3
#define READER 5

#define PERMS S_IRWXU | S_IRWXG | S_IRWXO

// количество читателей
#define READERCOUNT 0
// активные писатели
#define ACTIVEWRITER 1
// очередь писателей
#define WRITERCOUNT 2
// очередь читателей
#define READERWAIT 3

int semaphore;
int shared_memory;
int *addr_shared_memory;

// Массив структур
struct sembuf start_read[] = { {READERWAIT, 1, SEM_UNDO},
                               {ACTIVEWRITER, 0, SEM_UNDO},
                               {WRITERCOUNT, 0, SEM_UNDO},
                               {READERCOUNT, 1, SEM_UNDO},
                               {READERWAIT, -1, SEM_UNDO} };
struct sembuf stop_read[] = { {READERCOUNT, -1, SEM_UNDO} };
struct sembuf start_write[] = { {WRITERCOUNT, 1, SEM_UNDO},
                               {READERCOUNT, 0, SEM_UNDO},
                               {ACTIVEWRITER, -1, SEM_UNDO},
                               {WRITERCOUNT, -1, SEM_UNDO}, };
struct sembuf stop_write[] = { {ACTIVEWRITER, 1, SEM_UNDO} };
```

```

// собственный обработчик сигнала ctrl-c
void catch_sigp(int sig_numb)
{
    signal(sig_numb, catch_sigp);
    shmctl(shared_memory, IPC_RMID, NULL);
    semctl(semaphore, 0, IPC_RMID, 0);
}

void StartRead()
{
    semop(semaphore, start_read, 3);
}

void StopRead()
{
    semop(semaphore, stop_read, 1);
}

void StartWrite()
{
    semop(semaphore, start_write, 4);
}

void StopWrite()
{
    semop(semaphore, stop_write, 1);
}

// Читатель
void Reader(int value)
{
    while (1)
    {
        StartRead();
        printf("Reader%d get = %d\n", value, *addr_shared_memory);
        StopRead();
        sleep(1);
    }
}

// Писатель
void Writer(int value)
{
    while (1)
    {
        StartWrite();
        (*addr_shared_memory)++;
        printf("Writer%d put = %d\n", value, *addr_shared_memory);
        StopWrite();
        sleep(2);
    }
}

int main()
{
    // Создание семафора
    semaphore = semget(IPC_PRIVATE, 4, IPC_CREAT | PERMS);
    int sr = semctl(sem, READERCOUNT, SETVAL, 0);
    int sa = semctl(sem, ACTIVEWRITER, SETVAL, 1);
    int sw = semctl(sem, WRITERCOUNT, SETVAL, 0);
    // Объявление разделяемого сегмента
    shared_memory = (int*)shmget(IPC_PRIVATE, sizeof(int), IPC_CREAT | PERMS);
    addr_shared_memory = shmat(shared_memory, 0, 0);
}

```

```

// Создание процессов
int processes[READER + WRITER];
int parent = getpid();

for (int i = 0; i < WRITER + READER; i++)
{
    processes[i] = fork();
    if (getpid() != parent)
    {
        for (int j = 0; j < i; j++)
            processes[j] = -1;
        break;
    }
}

for (int i = 0; i < WRITER + READER; i++)
{
    if (processes[i] != 0)
        continue;

    if (i < WRITER)
    {
        Writer(i);
        return 0;
    }
    else
    {
        Reader(i - WRITER);
        return 0;
    }
}

signal(SIGINT, catch_sigp);
int status;
wait(&status);

// Очистка памяти
shmctl(shared_memory, IPC_RMID, NULL);
semctl(semaphore, 0, IPC_RMID, 0);
return 0;
}

```

Читатель получает доступ к общему ресурсу только в том случае, если в настоящий момент нет работающего писателя и нет ожидающего писателя. Первое условие должно обеспечиваться взаимоисключением событий, определяющих входы процессов писателей и читателей к общему ресурсу. Второе условие предотвращает бесконечное откладывание реализации процессов писателей из-за наплыва читателей. После окончания работы читателя счетчик читателей уменьшается на единицу, и если он будет пустой, то работа читателей может быть продолжена до тех пор, пока счетчик читателей не будет пустым.

Писатель получает доступ к общему ресурсу только в том случае, если в настоящий момент времени нет работающего читателя и счетчик читателей пуст или когда независимо от состояния счетчика читателей нет как ожидающего, так и работающего читателя. Когда писатель заканчивает работу, предпочтение отдается ожидающим читателям, а не ожидающим писателям, что предотвращает бесконечное откладывание процессов читателей из-за наплыва писателей.

## 13 билет

13.1 Виртуальная память: управление памятью страницами по запросу – три схемы. Алгоритмы вытеснения страниц: демонстрация особенностей на модели траектории страниц. Рабочее множество – определение, глобальное и локальное замещение. Флаги в дескрипторах страниц, предназначенные для реализации замещения страниц.

**Виртуальная память** – память, размер которой превышает размер реального физического пространства. Используется адресное пространство диска как область свопинга или пейджинга, т.е. для временного хранения областей памяти.

Подходы к реализации управления виртуальной памятью:

1. управление памятью страницами по запросам
2. сегментами по запросам
3. сегментами, деленными на страницы по запросам

**Страница** – является единицей физического деления памяти. Её размер устанавливается системой. У страницы размер не обязательно 4Кб. Просто такой размер оптимальен по количеству страничных прерываний.

**Сегмент** – является единицей логического деления памяти. Её размер определяется объемом кода. Любой сегмент создается такого размера, который требуется программе.

**Запрос** – страничное прерывание, которое возникает при попытке доступа к незагруженной в память странице. При его обработке происходит пэйджинг.

**Пэйджинг** – загрузка с диска новых и замещение старых страниц.

**Методы преобразования адресов:**

1. Прямое отображение таблиц страниц.

Каждый процесс должен иметь собственную таблицу страниц.

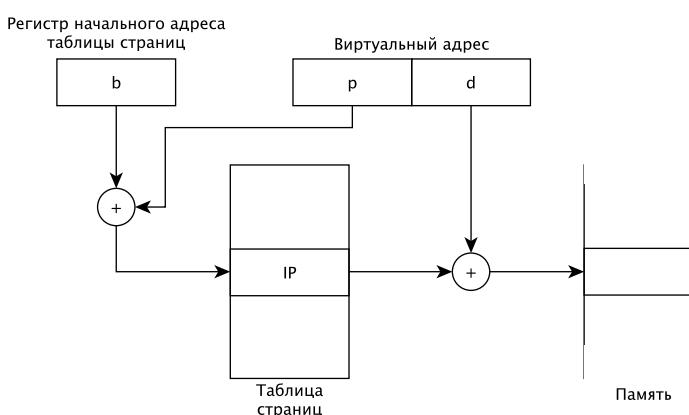
Главная таблица – таблица (современные ОС оперируют связными списками) процессов.

Таблица состоит из дескрипторов процессов.

В прямом отображении таблиц страниц – системные таблицы хранятся в оперативной памяти в системной области.

В процессоре должен быть регистр начального адреса таблицы страниц.

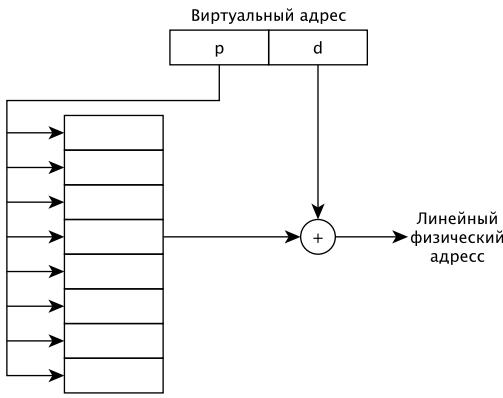
Если страница загружена в физическую память, то она имеет физический адрес.



Преобразование поддерживается аппаратно.

2. Ассоциативное отображение (для сокращения обращений к памяти).

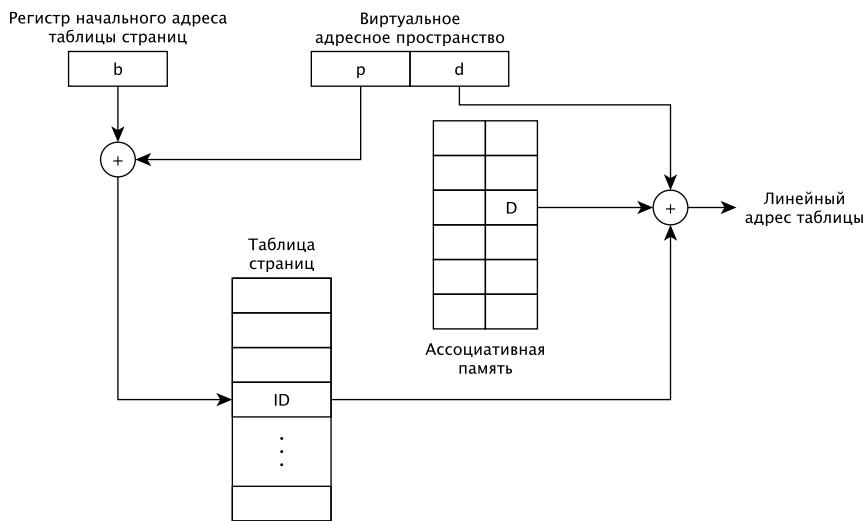
Использование ассоциативной памяти – память, которая обеспечивает выборку по ключу за 1 такт.



Дорогая память, всегда регистровая количества схем удваивается, много соединений.

Практически не используется.

### 3. Ассоциативно прямое отображение.



Ассоциативный кэш (последние обращения) – хранятся физические адреса страниц, используемых недавно.

Существует небольшая по объему ассоциативная память. (в современных ОС)

Эвристическое предположение: Если было обращение к странице, то следующее будет к той же странице.

Системы обеспечивают скоростные показатели 90% и более в отличие от полностью с прямым отображением.

## Алгоритмы замещения

Если какую-то страницу нужно загрузить, значит одну надо вытолкнуть.

### 1. Выталкивание случайной страницы.

- Малые накладные расходы
- Может быть вытолкнута часто используемая или только что загруженная

### 2. Алгоритм FIFO: вытеснение страницы, которая дольше всего находится в памяти. Способы: временная метка (вытеснение с меньшей временной меткой) либо ведется связный список.

Этот алгоритм исключает возможность выгрузки только что загруженной страницы, но не исключает выгрузку часто используемой.

### 3. Алгоритм LRU (Least Recently Used): замещаем наименее используемую страницу.

Строится на эвристическом предположении, правиле, если некоторые страницы имеют большое число обращений, то следующее обращение будет к ним.

Необходимо или изменять временную метку при каждом обращении, или при каждом обращении помещать этот сегмент в начало связного списка.

Наиболее затратно. (большие накладные расходы)

#### 4. Алгоритм NUR (Not Used Recently):

Колossalные накладные расходы LRU привели к использованию аппроксимирующих его алгоритмов.

Приписывание каждой странице бита обращения.

- При добавлении бит обращения 0.
- Периодически все биты обращений сбрасываются в 0.
- При обращении к сегменту устанавливается в 1.

Когда нужно заместить, ищется страница с нулевым битом обращения.

Кроме бита обращения вводится бит модификации.

Выгоднее вытеснять не модифицированную страницу, потому что ее копия лежит на диске и не надо выполнять точное копирование.

#### 5. Алгоритм LFU (Least Frequency Used): Наименее часто используемая страница.

При переполнении счетчик сбрасывается. Может быть вытеснена только что загруженная страница, не набравшая число обращений.

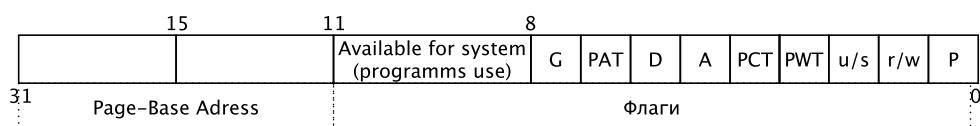
#### Рабочее множество

Рабочее множество – набор страниц, используемых процессом в настоящее время.

#### Вытеснение

Глобальное вытеснение – вытеснение любой страницы любого процесса. Локальное вытеснение – страница для вытеснения выбирается из пула загруженных страниц данного процесса.

#### Флаги в дескрипторах страниц



- P (Present): указывает, находится ли страница (или таблица страниц) в физической памяти.
- R/W (Read/Write): определяет привилегии чтения/записи для страницы или группы страниц (в случае, когда элемент каталога страниц указывает на таблицу страниц).
- U/S (User/Supervisor): определяет привилегии пользователя/супервизора для страницы или группы страниц.
- PWT (Page Write Throw): контролирует кэширование страницы (write-through и write-back).
- PCD (Page Cache Disable): контролирует кэширование страницы.
- A (Accessed): показывает, было ли произведено обращение к странице с момента загрузки ее в память.
- D (Dirty): показывает была ли произведена запись в страницу.
- PAT (Page throw Attribute Index)
- G (Global page): указывает, что описываемая страница является ГЛОБАЛЬНОЙ, если установлен.
- Available for system

#### Флаги для замещения

Present, Access, Dirty.

### 13.2 Синхронизация и взаимоисключение параллельных процессов в распределенных системах: централизованный и распределенный алгоритмы, алгоритмы Token-ring; сравнение алгоритмов. Транзакции: определение, особенности, двухфазный протокол фиксации.

Чтобы не терять значение разделяемой переменной, необходимо обеспечить монопольный доступ процессов к разделяемым переменным ресурсам методами взаимоисключения. Только один процесс может получить доступ к данным в один момент. Те участки кода, в которых выполняется обращение к разделяемым данным называются **критическими**. Только один процесс может находиться в критическом участке.

Существуют следующие способы взаимоисключения:

1. Программный способ
2. Аппаратный способ
3. С помощью семафоров
4. С использованием мониторов

**Активное ожидание на процессоре** – ситуация, когда процесс занимает процессорное время, проверяя значение флага (занятости ресурса другим процессом). Активное ожидание на процессоре является неэффективным использованием процессорного времени.

Возможные варианты развития событий:

1. Возможно, что оба процесса пройдут цикл ожидания и попадут в свои критические секции.
2. Возможно бесконечное откладывание (зависание) – ситуация, когда разделённый ресурс снова захватывается тем же процессом.
3. Тупик (deadlock, взаимоблокировка) – ситуация, когда оба процесса установили флаги занятости и ждут. Т.е. каждый ожидает освобождения ресурса, занятого другим процессом.

### **Централизованный алгоритм.**

Наиболее очевидный путь реализации с использованием тех же подходов, как и для отдельной машины.

- Выбирается процесс-координатор (процесс, выполняющийся на машине с наибольшим сетевым номером).
- Когда какой-нибудь процесс хочет войти в критический участок  $\Rightarrow$  посыпает координатору запрос (критическая секция, куда хочет войти) и ждет сообщение-ответ, которое разрешит ему войти в данный критический участок.
- Если какой-то процесс уже находится в данном критическом участке, то координатор ставит запрос в очередь и когда процесс выходит из критического участка, то 1-ому процессу из очереди посыпается ответ.
- Если 2 процесса хотят одновременно войти в критический участок: решается с помощью алгоритма Лампорта: чье сообщение пришло раньше, логические часы синхронизированы.
- Процесс-координатор может прекратить существование. Если какой-то процесс обнаружит отсутствие координатора, то существует подход для выбора нового, он инициирует выборы. Процесс посыпает сообщение с предложением выбора нового координатора (указав собственный номер и рассыпает всем процессам, если процесс обнаружит что его номер больше, то посыпает назад подтверждение приема такого сообщения, а сам инициирует новые выборы).

В результате все процессы кроме одного прекратят выборы и процесс, который не получит ни одного ответа является «победителем» и становится координатором.

На каждой машине должно быть прописано ПО для решения задачи + для решения работы координаторы.

### **Распределенный алгоритм.**

Когда процесс хочет войти в критический участок, то создает сообщение с ID критического участка, своим номером и временем отправки.

После создания рассыпает n-1 сообщение всем процессам.

Передача сообщения является надежной, подтверждается получение.

Получив сообщение - запрос, процесс может выполнить следующие действия, в зависимости от того в каких отношениях находится с критическим участком:

- Процесс-получатель не находится и не собирается, отправляется сообщение-разрешение.
- Процесс-получатель находится в критическом участке, ответ не отправляется, ставит запрос в очередь.
- Процесс-получатель определил, что сам хочет войти в эту секцию, он сравнивает временную отметку поступившего сообщения со своей собственной. Если его временная отметка меньше, то он входит в критический участок и ставит в очередь запрос. Если его временная отметка больше, посыпает сообщение-разрешение.

В результате процесс может войти в критический участок, если получит n-1 разрешение. Если хотя бы 1 не получит, войти не может.

### **Token Ring**

В этом алгоритме процессы создают логическое кольцо (направленное).

Каждый процесс знает номер своей позиции и номер ближайшего к нему процесса.

По этому кольцу циркулирует Token в определенном направлении.

Такой Token отмечает 1 критическую секцию. Когда процесс получает Token, он анализирует, не требуется ли ему войти в критическую секцию. Если надо – входит (удерживая Token), выходя отправляет дальше. Если нет – посыпает Token дальше.

Если желающих нет, Token циркулирует по кольцу с большой скоростью.

В системе столько Token сколько критических секций.

### Транзакции

Транзакция – само по себе неделимое действие.

Транзакция – средство взаимодействия процессов, высокоуровневое.

Транзакция – последовательность операций над одним/несколькими объектами базы данных (файлами, записями и т. д.), которые переводят систему из одного целостного состояния в другое целостное состояние.

Модель неделимой транзакции пришла из бизнеса.

Один процесс объявляет, что хочет начать транзакцию с одним или более объектами. Происходит изменения объектов какое-то время.

Инициатор транзакции объявляет, что он хочет завершить транзакцию. Если все процессы с ним соглашаются, то результат фиксируется. Если один/более процессов отказываются/потерпели крах, тогда все изменения возвращаются к исходному состоянию(откат).

Системные вызовы: `begintransaction`, `aborttransaction`, `endtransaction`, `read`, `write`.

Транзакции обладают свойствами

1. упорядоченности. Гарантирует, что если 2 или более транзакция выполняются параллельно, то конечный результат будет выглядеть будто транзакции выполнялись в порядке.
2. неделимости. Когда транзакция выполняется никакой другой процесс не может увидеть промежуточные результаты.
3. постоянством. После фиксирования транзакции никакой сбой не может отменить результатов ее выполнения.

### Механизмы транзакции

1. Предполагает, что все процессы выполняющие общую транзакцию, выполняются в индивидуальных рабочих пространствах, где содержатся все необходимые копии объектов, используемых во время транзакции.

Изменяются копии, если транзакция завершена, изменения вносятся в исходные объекты.

Недостаток: большие накладные расходы.

2. Список намерений.

Заключается в том, что модифицируются сами исходные объекты, файлы, структуры. Но перед их изменением производится запись в специальный файл, журнал регистрации, в котором отмечается какая транзакция делает изменения, какой файл или структура меняется: и новая, и старая.

Только после того, как успешно выполнена, производится запись в журнал и изменение исходных объектов.

Если транзакция фиксируется, об этом делается запись в журнале, но старые значения сохраняются.

Если транзакция прерывается, то информация записывается в журнал, используемый для отката.

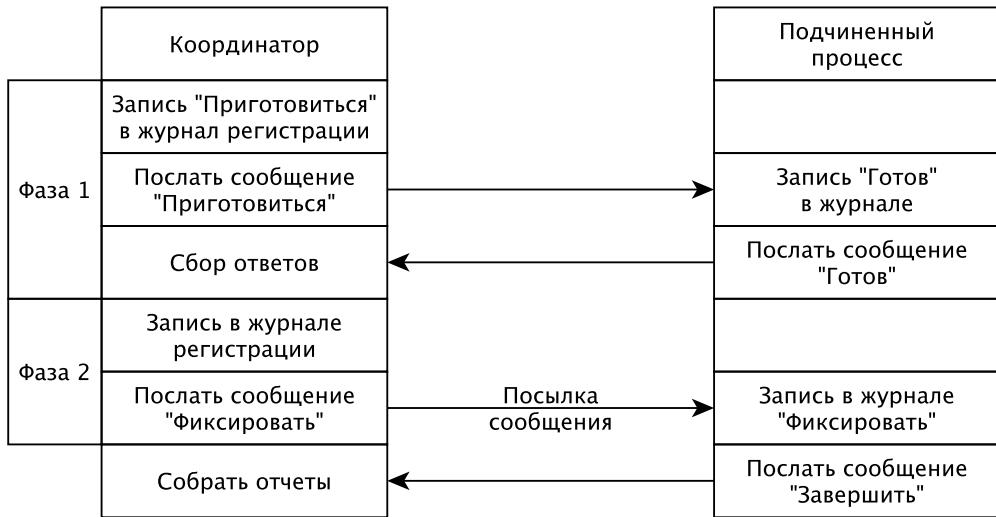
Транзакции очень важны в распределенных системах, когда процессы выполняются на разных машинах (свои локальные файлы и данные).

Для достижения свойства неделимости в распределенных системах используется протокол **2-ух фазной фиксации** транзакций.

Суть протокола: один из процессов выполняет функции координатора.

Координатор начинает транзакцию и делает запись в своем локальном журнале. После этого он посыпает подчиненным процессам, участвующим в транзакции, сообщение о том, что началась транзакция. Процессы приступают к выполнению задач.

Через некоторое время координатор приходит к мнению о завершении транзакции, пишет в журнале сообщений приготовиться к фиксированию, рассыпает подчиненным процессам. Процесс анализирует состояние, если он готов, запись в журнал Готов, посылка сообщения Готов координатору.



Если координатор получил сообщение Готов от всех процессов, он делает запись в журнале фиксировать. Посыпает сообщение всем процессам.

Получив такое сообщение, процесс проверяет готовность и делает запись в журнал Фиксировать и посыпает сообщение Завершить.

## 14 билет

### 14.1 Процессы: взаимодействие процессов в распределенных системах; централизованный и распределенный алгоритмы, синхронизация часов (алгоритм Лампорта); RPC – механизм

В распределенных системах, называющим распределениями потому что ресурсы рассредоточены, процессы не имеют общей памяти (например, сети). Могут синхронизироваться только сообщениями. Взаимодействие по системе «клиент-сервер».

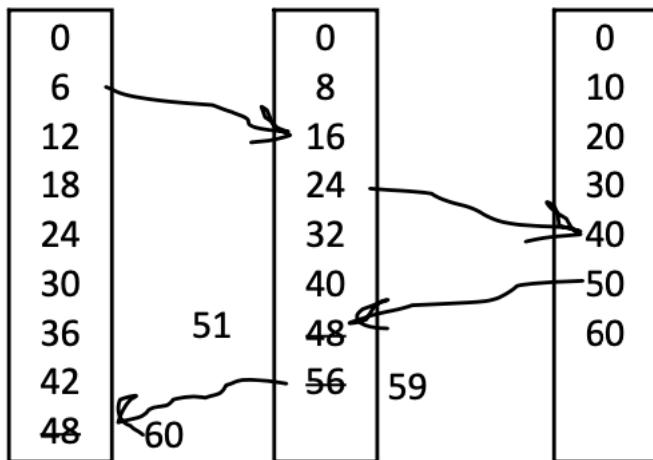
Процессам часто нужно взаимодействовать друг с другом, например, 1 процесс может передавать данные другому процессу, или несколько процессов могут обрабатывать данные из общего файла. В этих случаях возникает проблема синхронизации процессов. Она связана с потерей доступа к параметрам из-за их некорректного разделения.

Необходимо обеспечить монопольный доступ процесса к критическому ресурсу до тех пор пока процесс его не освободит. Т.е. чтобы не могли одновременно войти в критическую секцию.

#### Алгоритм синхронизации логических часов (алг. Лампорта)

Проблема: локальные часы компьютера, имеющие ограниченную точность. Сообщение получено раньше, чем было отправлено – НЕ МОЖЕТ!

1. Процесс, отправивший сообщение, отправляет и время отправки по локальным часам.
2. Получивший процесс приравнивает свое время и время отправки + 1, при условии, что оно больше времени получения.



#### Централизованный алгоритм.

Наиболее очевидный путь реализации с использованием тех же подходов, как и для отдельной машины.

- Выбирается процесс-координатор (процесс, выполняющийся на машине с наибольшим сетевым номером).
- Когда какой-нибудь процесс хочет войти в критический участок  $\Rightarrow$  посылает координатору запрос (критическая секция, куда хочет войти) и ждет сообщение-ответ, которое разрешит ему войти в данный критический участок.
- Если какой-то процесс уже находится в данном критическом участке, то координатор ставит запрос в очередь и когда процесс выходит из критического участка, то 1-ому процессу из очереди посыпается ответ.
- Если 2 процесса хотят одновременно войти в критический участок: решается с помощью алгоритма Лампорта: чье сообщение пришло раньше, логические часы синхронизированы.
- Процесс-координатор может прекратить существование. Если какой-то процесс обнаружит отсутствие координатора, то существует подход для выбора нового, он инициирует выборы. Процесс посылает сообщение с предложением выбора нового координатора (указав собственный номер и рассыпает всем процессам, если процесс обнаружит что его номер больше, то посыпает назад подтверждение приема такого сообщения, а сам инициирует новые выборы.

В результате все процессы кроме одного прекратят выборы и процесс, который не получит ни одного ответа является «победителем» и становится координатором.

На каждой машине должно быть прописано ПО для решения задачи + для решения работы координаторы.

## Распределенный алгоритм.

Когда процесс хочет войти в критический участок, то создает сообщение с ID критического участка, своим номером и временем отправки.

После создания рассыпает n-1 сообщение всем процессам.

Передача сообщения является надежной, подтверждается получение.

Получив сообщение - запрос, процесс может выполнить следующие действия, в зависимости от того в каких отношениях находится с критическим участком:

- Процесс-получатель не находится и не собирается, отправляется сообщение-разрешение.
- Процесс-получатель находится в критическом участке, ответ не отправляется, ставит запрос в очередь.
- Процесс-получатель определил, что сам хочет войти в эту секцию, он сравнивает временную отметку поступившего сообщения со своей собственной. Если его временная отметка меньше, то он входит в критический участок и ставит в очередь запрос. Если его временная отметка больше, посылает сообщение-разрешение.

В результате процесс может войти в критический участок, если получит n-1 разрешение. Если хотя бы 1 не получит, войти не может.

## RPC (вызов удаленных процедур).

Механизм реализованный первоначально в Unix, с помощью которого один процесс активизирует другой процесс на удаленной или той же самой машине для запуска/выполнения функции от своего имени.

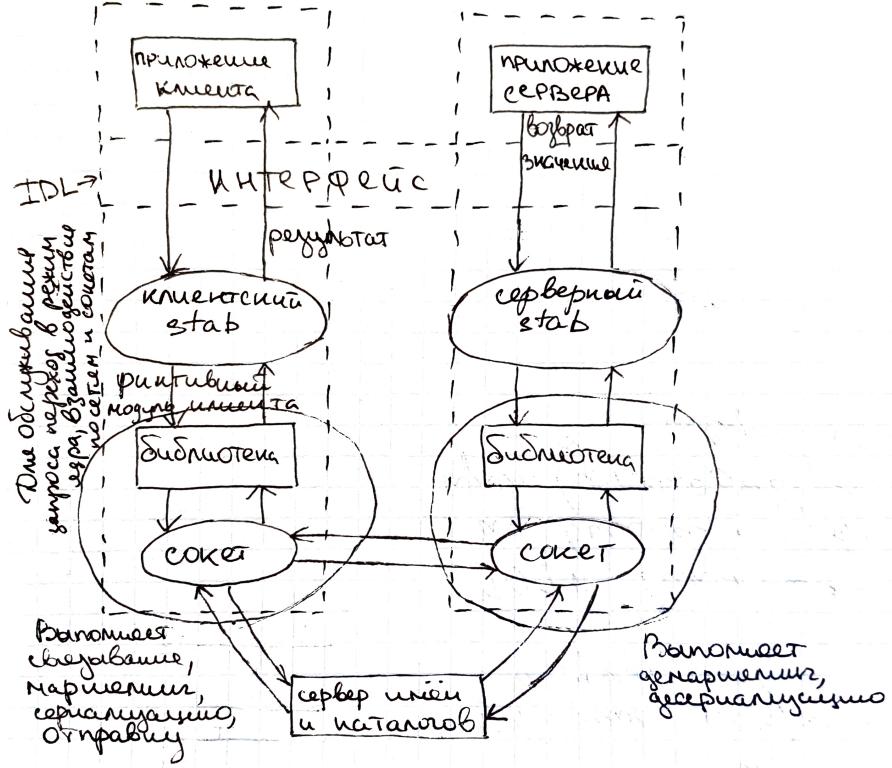
Вызов локальной процедуры – процесс вызывает функцию и передает ей данные (фактические параметры) и собственно ждет, когда будет возвращен результат выполнения этой функции.

Спецификация RPC – эту функцию выполняет другой процесс.

Такое взаимодействие выполняется по **схеме клиент-сервер**.

Процесс,зывающий RPC, является клиентским, а процесс, который выполняет RPC функцию – является серверным.

RPC является низкоуровневым средством взаимодействия, от этого не становится менее интересным. У RPC есть особенности, главное из которых является то, что механизм RPC скрывает от пользователя этот механизм удаленного доступа. Пользователь RPC может не заморачиваться на то, что он обращается к удаленной машине. RPC появились в 80-е.



Клиент и сервер связаны с стаб на этапе компоновки.

Стаб (фиктивный модуль) – генерируется компилятором из определения интерфейса, который используется клиентом.

В результате действий система создает пакет RPC, а также создает одно или несколько обращений к серверу. Выполнение интерфейсов с помощью специального языка – IDL.

Клиентский сервер подключается через RPC.

Передаются форматированные данные (маршинг и сериализация).

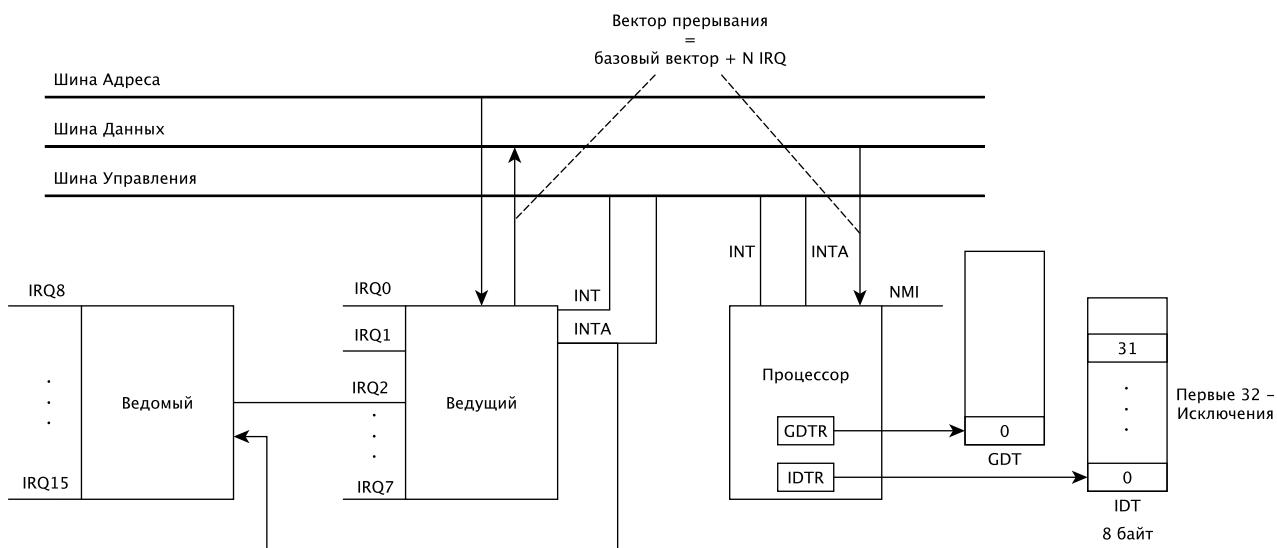
Маршинг – перекомпоновка и упаковка данных в связи с требованиями системы.

Сериализация – преобразование сообщения в серию байтов.

## ~~14.2 Аппаратные прерывания: типы аппаратных прерываний; особенности. Прерывания от устройств ввода-вывода: назначение и аппаратная реализация. Прерывание от системного таймера в защищенном режиме. Пример кода обработчика прерывания от системного таймера из лабораторной работы по защищенному режиму.~~

Типы аппаратных прерываний:

- Асинхронные события в системе. Также имеют различный характер в системе. Самое массовое – прерывания от внешних устройств.
- Второй тип – прерывание от системного таймера, которое выполняется по тику(выделенный импульс) 18,3 раз в секунду.
- Третий тип – прерывание от действий оператора (ctrl+alt+del , ctrl+C = завершение процесса в unix)



IRQ1 – сигнал от клавиатуры.

Когда завершается операция ввода/вывода, в буфер клавиатуры записывается код, контроллер отправляет его на ножку IRQ1. Формируется запрос INT и по шине управление поступает в процессор.

В конце цикла выполнение каждой команды процессор проверяет наличие сигнала прерывания и переходит в обработчик, прежде посыпая через INT A сигнал.

Получив INT A контроллер формирует вектор прерывания, который по шине данных поступает в процессор. Вектор используется для адресации обработчика.

У контроллера прерываний есть порт, значит он адресуется.

### 14.2.1 Функции обработчика прерывания системного таймера (int 8h)

#### Защищенный режима

У первых компьютеров были только дисководы без винчестеров. Чтобы что-то прочитать нужно было разворачивать дискету. А это временные затраты! Было решено возложить функцию отключения моторчика на таймер 2с. На каждом тике он декрементируется. Если 0, то в порт посыпается команда выключения.

INC и DEC поддерживаются аппаратно. Поэтому выполняются быстро.

Вызов пользовательского прерывания INT 1CH - чистая заглушка, чтобы программисты не вставляли большой код в прерывание.

#### Windows

#### По тику

- Инкремент счётчика системного времени
- Декремент счетчиков отложенных задач
- Декремент остатка кванта текущего потока.
- Активизация обработчика ловушки профилирования ядра

## По главному тику

- Инициализация диспетчера настройки баланса (освобождение объекта «событие» каждую секунду)

## По кванту

- Инициализация диспетчеризации потоков (посредством добавления соответствующего объекта DPC в очередь)

## Unix/Linux

## По тику

- Счет тиков аппаратного таймера
- Декремент кванта текущего потока.
- Инкремент времени использования процессора
- Наблюдение за списком отложенных вызовов.

## По основному тику

- Добавление в очередь на выполнение функций, относящихся к работе планировщика-диспетчера (напр: пересчет приоритетов)
- Пробуждение системных процессов, таких, как swapper и pagedaemon (процедура wakeup перемещает дескрипторы процессов из очереди «спящих» в очередь «готовых к выполнению»)
- Обработка сигналов тревоги; декремент будильников и измерение времени работы процесса

## По кванту

- Посылка текущему процессу сигнала SIGXCPU, если израсходован выделенный ему квант процессорного времени.

## Пример из лабораторной

```
; Обработчик прерывания таймера
int08_handler:
    push eax ; Это аппаратное прерывание, сохраняем регистры
    push ebp
    push ecx
    push dx
    mov eax,time_08

    mov ebp,190 ; Ещё 30h - для слова timer и поскольку число печатается справа-нал
                 ево
    mov ecx,8 ; Число выводимых символов
    add ebp,0B8000h ; Начальное смещение на экране
                     ; 0B8000h - смещение видеобуфера относительно начала сегмента.

    cycle1:
        mov dl,al ; Кладём в DL текущее значение AL (самый младший байт EAX)
        and dl,0Fh ; Оставляем от него одно 16ричное число (последняя цифра)
        cmp dl,10
        jl number1
        sub dl, 10 ; Превращаем число в букву
        add dl,'A'
        jmp print1

    number1:
        add dl,'0' ; Превращаем это цифру в символ

    print1:
        mov es:[ebp],dl ; Выводим в видеобуфер
        ror eax,4 ; Циклически двигаем биты в EAX - таким образом, после всех
                  ; перестановок, EAX окажется тем же что и в начале, нет необходимости на
                  ; PUSH; POP
        sub ebp,2 ; Смещаем позицию вывода
```

```
    loop cycle1      ; Цикл
; Иниремент
inc eax
mov time_08, eax

; Посыпаем сигнал EOI контроллеру прерываний и восстанавливаем регистры
mov al, 20h
out 20h, al
pop dx
pop ecx
pop ebp
pop eax
; Выходим из прерывания
iretd
```

## 15 билет

### 15.1 Межпроцессорное взаимодействие в Unix System V (IPC): сигналы, программные каналы, семафоры и разделяемая память; примеры использования из лабораторных работ.

Набор средств взаимодействия параллельных процессов в Unix System V (IPC):

- Семафоры
- Сигналы
- Программные каналы (именованные и неименованные)
- Разделяемая память

#### Сигналы

Сигнал – способ информирования процессов ядром о произшествии какого-то события.

Процессы одной группы получают одинаковые сигналы.

Если возникает несколько однотипных событий, процессу будет подан только один сигнал. Сигнал означает, что произошло событие, но ядро не сообщает сколько таких событий произошло.

Средство передачи сигнала – kill(), приема сигнала – signal().

Реакция процесса на сигнал – по умолчанию. Но процесс можно определить на реакцию с помощью собственного обработчика.

SIG\_IGN – флаг игнорирования

SIG\_DFL – реакция по умолчанию

signal(SIGINT, SIG\_IGN);

Методом посылки и приема сигналов служат 2 системных вызова: kill и signal.

Переносимое ПО нельзя писать с сигналами.

```
#include <signal.h>
void (*signal(int sig, void(*handler)(int)))(int);
```

Системный вызов signal возвращает указатель на предыдущий обработчик данного сигнала и его можно использовать для восстановления обработчика сигнала.

Сигналы можно использовать для изменения хода процесса.

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

int signal_flag = 0;

/* собственный обработчик сигнала ctrl-c */
void catch_sigp(int sig_num)
{
    signal(sig_num, catch_sigp);
    signal_flag = 1;
    printf("\n Parent catch sig %d\n", sig_num);
}

int main(void)
{
    pid_t childpid1;
    pid_t childpid2;
    /* дескрипторы программных каналов */
    int my_pipe1[2];
    int my_pipe2[2];
    /* [0] - выход для чтения, [1] - выход для записи */

    /* потомок унаследует открытый программный канал предка */
    if (pipe(my_pipe1) == -1)
    {
        perror("Can't pipe");
        exit(1);
    }
    if (pipe(my_pipe2) == -1)
```

```

{
    perror("Can't pipe");
    exit(1);
}

if ((childpid1 = fork()) == -1) /* если fork завершился успешно, pid > 0 в роди-
    тельском процессе */
{
    perror("Can't fork"); /* fork потерпел неудачу (например, память или какая-ли-
        бо */                                /* таблица заполнена) */
    exit(1);
}

if (childpid1 == 0)
{ /* здесь располагается дочерний код */
    printf("Child1 forked \n");
    printf("Child1 id = %d, parent id = %d, group id = %d \n", getpid(), getppid()
        (), getpgrp());
}

char buf[40];
close(my_pipe1[1]); /* потомок ничего не запишет в канал */
read(my_pipe1[0], buf, sizeof(buf));
printf("Child1 catch message = %s \n", buf);

return 0;
}

if ((childpid2 = fork()) == -1) /* если fork завершился успешно, pid > 0 в роди-
    тельском процессе */
{
    perror("Can't fork"); /* fork потерпел неудачу (например, память или какая-ли-
        бо */                                /* таблица заполнена) */
    exit(1);
}

if (childpid2 == 0)
{ /* здесь располагается дочерний код */
    printf("Child2 forked \n");
    printf("Child2 id = %d, parent id = %d, group id = %d \n", getpid(), getppid()
        (), getpgrp());
}

char buf[40];
close(my_pipe2[1]); /* потомок ничего не запишет в канал */
read(my_pipe2[0], buf, sizeof(buf));
printf("Child2 catch message = %s \n", buf);

}

if (childpid1 != 0 && childpid2 != 0)
{ /* здесь располагается родительский код */
    printf("Parent id = %d, child1 id = %d, child2 id = %d, group id = %d \n",
        getpid(), childpid1, childpid2, getpgrp());

/* ожидание сигнала */
signal(SIGINT, catch_sigp);
sleep(2);

if (!signal_flag)
{
    char msg1[] = "Hello, child1!";
    close(my_pipe1[0]); /* предок ничего не считает из канала */
    write(my_pipe1[1], msg1, sizeof(msg1));

    char msg2[] = "Hello, child2!";
    close(my_pipe2[0]); /* предок ничего не считает из канала */
    write(my_pipe2[1], msg2, sizeof(msg2));
}

```

```

}

int status1;
childpid1 = wait(&status1);
printf("Child has finished: PID = %d\n", childpid1);

int status2;
childpid2 = wait(&status2);
printf("Child has finished: PID = %d\n", childpid2);

if (WIFEXITED(status1) && WIFEXITED(status2))
    printf("Children exited with code %d and %d\n", WEXITSTATUS(status1),
           WEXITSTATUS(status2));
else
    printf("Child terminated abnormally\n");
}
return 0;
}

```

## Программные каналы

Именованные программные каналы создаются командой mknode.

Программный канал - это специальный файл, в который можно «писать» информацию и из которого эту информацию можно «читать».

Причем порядок записи информации и последующего чтения – FIFO (очередь).

Именованный канал имеет имя, которое указывается при вызове команды mknode [опции] <имя> p.

Использовать именованный канал может любой процесс, «знающий» имя канала.

Системный вызов pipe() создает неименованный программный канал. Неименованные программные каналы могут использоваться для обмена сообщениями между процессами родственниками. В отличие от именованных программных каналов неименованные не имеют идентификатора, но имеют дескриптор.

Процесс-потомок наследует все дескрипторы открытых файлов процесса-предка, в том числе и неименованных программных каналов.

Программные каналы имеют встроенные средства взаимоисключения: из канала нельзя читать, если в него пишут, и в канал нельзя писать, если из него читают.

Программный канал – буфер в системной области памяти. (последовательность адресов)

Программный канал небольше страницы.

Создаются в системной области памяти, так как процесс не может обращаться за собственное адресное пространство.

Pipe буфферизуется на 3 уровнях:

Трубы буфферизуются в системную область памяти, в системе может существовать определенное кол-во труб, если при попытке создать трубу число будет превышать, то буфера имеющие наибольшее время существования будут переписаны на диск.

При этом будут использоваться стандартные функции работы с файлами.

Если процесс пытается записать в программный канал больше 4096 байт, то труба буфферизуется во времени, до тех пор пока данные не будут из нее прочитаны.

Другими словами подсистема ввода-вывод обеспечивает приостановку процесса, если канал заполнен.

Команда write будет успешно выполнена, если есть доступное место, в противном случае ожидание.

Команда read процесс будет блокирован если канал пуст.

Ограничение размера объясняется стремлением повысить эффективность обмена засчет размещения канала в оперативной памяти, если канал большой (превышает размер страницы), то он частично лежит на диске, операции чтения и записи выполняются значительно медленнее.

```

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

#define ANSI_COLOR_GREEN "\x1b[32m"
#define ANSI_COLOR_RESET "\x1b[0m"

#define N 20

int main()
{

```

```

int status;
pid_t childpid1, childpid2;
char buffer[N];
char message1[N] = "hello";
char message2[N] = "goodbye";
int channel1[2], channel2[2];

if (pipe(channel1) == -1)
{
    perror("Can not pipe.\n");
    return 1;
}

childpid1 = fork();

if (childpid1 == -1)
{
    perror("Can not fork.\n");
    return 1;
}
else if (childpid1 == 0)
{
    close(channel1[0]);
    write(channel1[1], message1, sizeof(message1));
    exit(0);
}

if (pipe(channel2) == -1)
{
    perror("Can not pipe.\n");
    return 1;
}

childpid2 = fork();

if (childpid2 == -1)
{
    perror("Can not fork.\n");
    return 1;
}
else if (childpid2 == 0)
{
    close(channel2[0]);
    write(channel2[1], message2, sizeof(message2));
    exit(0);
}
else
{
    wait(&status);
    if (WIFEXITED(status))
    {
        printf(ANSI_COLOR_GREEN "child process exit success\n"
               ANSI_COLOR_RESET);

        close(channel1[1]);
        read(channel1[0], buffer, sizeof(buffer));
        printf("message = %s\n", buffer);

        close(channel2[1]);
        read(channel2[0], buffer, sizeof(buffer));
        printf("message = %s\n", buffer);
    }
}

```

```
    return 0;  
}
```

## Семафоры

Win, Unix, современный Linux поддерживают наборы считающих семафоров, они обладают важнейшим свойством: одной неделимой операцией можно изменить все или часть семафоров набора.

Если какая-либо операция не может быть выполнена над одним семафором, то неудачной считается операция над всеми семафорами.

Семафоры поддерживаются в ядре таблицей семафоров. Каждая строка описывает 1 набор из всех созданных в системе наборов семафоров.

1. Имя – целое число, присваивается процессам создавшим набор семафоров. Другие процессы могут по имени открыть.
2. UID – идентификатор создателя набора семафора и его группы. Если UID процесса совпадает с UID создателя, то может удалять набор и изменять его управляющие параметры.
3. Права доступа
4. Количество семафоров в наборе
5. Время изменения одного или нескольких значений семафора последним процессом
6. Время изменения управляющих параметров
7. Указатель на массив семафоров

О каждом семафоре известно:

1. Значение семафора
2. ID процесса, который оперировал с семафором в последний раз
3. Число процессов заблокированных на семафоре

В отличие от семафора Дейкстры, на семафоре в Unix определено 3 операции:

- Захват, декремент
- Освобождение, инкремент
- Проверка семафора на 0. Процесс выполняющий такую проверку переходит в состояние ожидания освобождения ресурса.

В система определены структуры для работы с семафорами: <sys/sem.h>

```
struct sembuf  
{  
    n_short sem_num; // индекс  
    short sem_op; // операция  
    short sem_flg; // флаги  
}
```

3 операции:

- sem\_op < 0. Захват, декремент. Если не может то блокируется в операции.
- sem\_op > 0. Освобождение, инкремент
- sem\_op = 0. Проверка семафора на 0.

На семафорах определены флаги:

IPC\_NOWAIT – информирует ядро о нежелании процесса переходить в состояние ожидания.

Это позволяет избежать очереди к семафору, в случае аварийного завершения или kill.

В силу того, что kill невозможно прекратить, убиваемый процесс не может осуществить освобождение семафора.

SEM\_UNDO – указывает ядру, что необходимо отслеживать значение семафора, в результате завершения процесса, вызвавшего sem\_op, ядро отменяет все изменения, чтобы процессы не были блокированы навсегда.

На семафоре определены следующие системные вызовы:

- semget() – идентификатор, количество семафоров в наборе, флаги | права доступа

- semcontrol()
- semop() – fd, массив структур, значения для семафора набора

Достоинство: Исключают активное ожидание на процессоре (за счет того, что процесс который не может получить, процесс блокируется. Заблокировать может только ядро. Работает в режиме ядра, переключение контекста.)

#### **Разделяемая память**

Средство передачи сообщений от одного процесса к другому.

Разделяемые сегменты разработаны для сокращения работы.

Могут создаваться в области ядра, так как адресное пространство процессов защищено.

Разделяемая память не имеет средств взаимоисключения, поэтому часто используется вместе с семафорами.

Поскольку разделяемая память создана для увеличения производительности, то никакого копирования из адресного пространства в разделяемые сегменты не происходит.

Разделяемый сегмент подключается к адресному пространству.

Процесс присоединяется для чтения и записи, путем получения указателя.

В ядре есть таблица разделяемых сегментов (памяти). <sys/shm.h>

Каждый элемент таблицы описывает разделяемый сегмент.

В дескрипторе есть поле-указатель на разделяемый сегмент.

Определены следующие системные вызовы:

- shmget() – создание разделяемого сегмента
- shmcontrol() – изменение управляемых параметров
- shmat() – подключение к адресному пространству, передается дескриптор
- shmdt() – отсоединение от адресного пространства

После создания разделяемого сегмента любой процесс может подключить его к своему адресному пространству и работать с ним, как с обычным сегментом памяти.

По завершении процесса, если никаких действий не предпринято, разделяемый сегмент сохраняется в памяти. На разделяемый сегменты определены несколько системных ограничений.

- SHMMNI – максимальное число разделяемых сегментов, которые могут существовать одновременно (блокируется если превышено число)
- SHMMIN – минимальный размер разделяемого сегмента (создание будет отклонено)
- SHMMAX – максимальный размер разделяемого сегмента (создание будет отклонено)

Семафоры и разделяемая память используется в задачах читатели-писатели, производство-потребление.

## **15.2 Синхронизация и взаимоисключение параллельных процессов в распределенных системах: централизованный и распределенные алгоритмы – сравнение.**

Чтобы не терять значение разделяемой переменной, необходимо обеспечить **монопольный доступ** процессов к **разделяемым переменным ресурсам** методами взаимоисключения. Только один процесс может получить доступ к данным в один момент. Те участки кода, в которых выполняется обращение к разделяемым данным называются **критическими**. Только один процесс может находиться в критическом участке.

Существуют следующие способы взаимоисключения:

1. Программный способ
2. Аппаратный способ
3. С помощью семафоров
4. С использованием мониторов

**Активное ожидание на процессоре** – ситуация, когда процесс занимает процессорное время, проверяя значение флага (занятости ресурса другим процессом). Активное ожидание на процессоре является неэффективным использованием процессорного времени.

Возможные варианты развития событий:

1. Возможно, что оба процесса пройдут цикл ожидания и попадут в свои критические секции.

2. Возможно бесконечное откладывание (зависание) – ситуация, когда разделённый ресурс снова захватывается тем же процессом.
3. Тупик (deadlock, взаимоблокировка) – ситуация, когда оба процесса установили флаги занятости и ждут. Т.е. каждый ожидает освобождения ресурса, занятого другим процессом.

**В распределенных системах**, называющим распределениями потому что ресурсы рассредоточены, процессы не имеют общей памяти (например, сети). Могут синхронизироваться только сообщениями. Взаимодействие по системе «клиент-сервер».

Процессам часто нужно взаимодействовать друг с другом, например, 1 процесс может передавать данные другому процессу, или несколько процессов могут обрабатывать данные из общего файла. В этих случаях возникает проблема синхронизации процессов. Она связана с потерей доступа к параметрам из-за их некорректного разделения.

Необходимо обеспечить монопольный доступ процесса к критическому ресурсу до тех пор пока процесс его не освободит. Т.е. чтобы не могли одновременно войти в критическую секцию.

#### **Централизованный алгоритм.**

Наиболее очевидный путь реализации с использованием тех же подходов, как и для отдельной машины.

- Выбирается процесс-координатор (процесс, выполняющийся на машине с наибольшим сетевым номером).
- Когда какой-нибудь процесс хочет войти в критический участок  $\Rightarrow$  посылает координатору запрос (критическая секция, куда хочет войти) и ждет сообщение-ответ, которое разрешит ему войти в данный критический участок.
- Если какой-то процесс уже находится в данном критическом участке, то координатор ставит запрос в очередь и когда процесс выходит из критического участка, то 1-ому процессу из очереди посыпается ответ.
- Если 2 процесса хотят одновременно войти в критический участок: решается с помощью алгоритма Лампорта: чье сообщение пришло раньше, логические часы синхронизированы.
- Процесс-координатор может прекратить существование. Если какой-то процесс обнаружит отсутствие координатора, то существует подход для выбора нового, он инициирует выборы. Процесс посылает сообщение с предложением выбора нового координатора (указав собственный номер и рассыпает всем процессам, если процесс обнаружит что его номер больше, то посыпает назад подтверждение приема такого сообщения, а сам инициирует новые выборы.

В результате все процессы кроме одного прекратят выборы и процесс, который не получит ни одного ответа является «победителем» и становится координатором.

На каждой машине должно быть прописано ПО для решения задачи + для решения работы координаторы.

#### **Распределенный алгоритм.**

Когда процесс хочет войти в критический участок, то создает сообщение с ID критического участка, своим номером и временем отправки.

После создания рассыпает n-1 сообщение всем процессам.

Передача сообщения является надежной, подтверждается получение.

Получив сообщение - запрос, процесс может выполнить следующие действия, в зависимости от того в каких отношениях находится с критическим участком:

- Процесс-получатель не находится и не собирается, отправляется сообщение-разрешение.
- Процесс-получатель находится в критическом участке, ответ не отправляется, ставит запрос в очередь.
- Процесс-получатель определил, что сам хочет войти в эту секцию, он сравнивает временную отметку поступившего сообщения со своей собственной. Если его временная отметка меньше, то он входит в критический участок и ставит в очередь запрос. Если его временная отметка больше, посыпает сообщение-разрешение.

В результате процесс может войти в критический участок, если получит n-1 разрешение. Если хотя бы 1 не получит, войти не может.

## 16 билет

16.1 Взаимодействие параллельных процессов: проблемы; монопольный доступ и взаимоисключение; взаимодействие параллельных процессов в распределенных системах – особенности; централизованный алгоритм, распределенный алгоритм; синхронизация логических часов (алгоритм Лампорта).

Чтобы не терять значение разделяемой переменной, необходимо обеспечить **монопольный доступ** процессов к **разделяемым переменным ресурсам** методами взаимоисключения. Только один процесс может получить доступ к данным в один момент. Те участки кода, в которых выполняется обращение к разделяемым данным называются **критическими**. Только один процесс может находиться в критическом участке.

Существуют следующие способы взаимоисключения:

1. Программный способ
2. Аппаратный способ
3. С помощью семафоров
4. С использованием мониторов

**Активное ожидание на процессоре** – ситуация, когда процесс занимает процессорное время, проверяя значение флага (занятости ресурса другим процессом). Активное ожидание на процессоре является неэффективным использованием процессорного времени.

Возможные варианты развития событий:

1. Возможно, что оба процесса пройдут цикл ожидания и попадут в свои критические секции.
2. Возможно бесконечное откладывание (зависание) – ситуация, когда разделённый ресурс снова захватывается тем же процессом.
3. Тупик (deadlock, взаимоблокировка) – ситуация, когда оба процесса установили флаги занятости и ждут. Т.е. каждый ожидает освобождения ресурса, занятого другим процессом.

В **распределенных системах**, называющим распределениями потому что ресурсы рассредоточены, процессы не имеют общей памяти (например, сети). Могут синхронизироваться только сообщениями. Взаимодействие по системе «клиент-сервер».

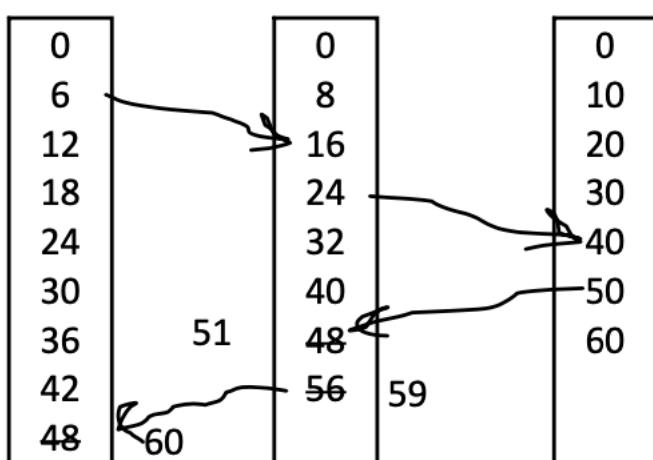
Процессам часто нужно взаимодействовать друг с другом, например, 1 процесс может передавать данные другому процессу, или несколько процессов могут обрабатывать данные из общего файла. В этих случаях возникает проблема синхронизации процессов. Она связана с потерей доступа к параметрам из-за их некорректного разделения.

Необходимо обеспечить монопольный доступ процесса к критическому ресурсу до тех пор пока процесс его не освободит. Т.е. чтобы не могли одновременно войти в критическую секцию.

**Алгоритм синхронизации логических часов (алг. Лампорта)**

Проблема: локальные часы компьютера, имеющие ограниченную точность. Сообщение получено раньше, чем было отправлено – НЕ МОЖЕТ!

1. Процесс, отправивший сообщение, отправляет и время отправки по локальным часам.
2. Получивший процесс приравнивает свое время и время отправки + 1, при условии, что оно больше времени получения.



### **Централизованный алгоритм.**

Наиболее очевидный путь реализации с использованием тех же подходов, как и для отдельной машины.

- Выбирается процесс-координатор (процесс, выполняющийся на машине с наибольшим сетевым номером).
- Когда какой-нибудь процесс хочет войти в критический участок  $\Rightarrow$  посыпает координатору запрос (критическая секция, куда хочет войти) и ждет сообщение-ответ, которое разрешит ему войти в данный критический участок.
- Если какой-то процесс уже находится в данном критическом участке, то координатор ставит запрос в очередь и когда процесс выходит из критического участка, то 1-ому процессу из очереди посыпается ответ.
- Если 2 процесса хотят одновременно войти в критический участок: решается с помощью алгоритма Лампорта: чье сообщение пришло раньше, логические часы синхронизированы.
- Процесс-координатор может прекратить существование. Если какой-то процесс обнаружит отсутствие координатора, то существует подход для выбора нового, он инициирует выборы. Процесс посыпает сообщение с предложением выбора нового координатора (указав собственный номер и рассыпает всем процессам, если процесс обнаружит что его номер больше, то посыпает назад подтверждение приема такого сообщения, а сам инициирует новые выборы).

В результате все процессы кроме одного прекратят выборы и процесс, который не получит ни одного ответа является «победителем» и становится координатором.

На каждой машине должно быть прописано ПО для решения задачи + для решения работы координаторы.

### **Распределенный алгоритм.**

Когда процесс хочет войти в критический участок, то создает сообщение с ID критического участка, своим номером и временем отправки.

После создания рассыпает n-1 сообщение всем процессам.

Передача сообщения является надежной, подтверждается получение.

Получив сообщение - запрос, процесс может выполнить следующие действия, в зависимости от того в каких отношениях находится с критическим участком:

- Процесс-получатель не находится и не собирается, отправляется сообщение-разрешение.
- Процесс-получатель находится в критическом участке, ответ не отправляется, ставит запрос в очередь.
- Процесс-получатель определил, что сам хочет войти в эту секцию, он сравнивает временную отметку поступившего сообщения со своей собственной. Если его временная отметка меньше, то он входит в критический участок и ставит в очередь запрос. Если его временная отметка больше, посыпает сообщение-разрешение.

В результате процесс может войти в критический участок, если получит n-1 разрешение. Если хотя бы 1 не получит, войти не может.

## **16.2 Процессы в UNIX: системные вызовы fork(), exec(), wait(), signal() – примеры из лабораторных работ.**

Любой процесс создается системным вызовом fork(), который определяется в системе как потомок (child). В Unix используется иерархия процессов, которая строится в отношении предок-потомок.

В результате вызова fork создается процесс-потомок. В Unix все рассматривается как файл (файлы, директории, устройства).

В современных системах применяется, так называемая, оптимизация fork(): для процесса потомка создаются собственные карты трансляции адресов (таблицы страниц), но они ссылаются на адресное пространство процесса-предка (на страницы предка). При этом для страниц адресного пространства предка права доступа меняются на only-read и устанавливается флаг – copy-on-write. Если или предок или потомок попытаются изменить страницу, возникнет исключение по правам доступа. Выполняя это исключение супервизор обнаружит флаг copy-on-write и создаст копию страницы в адресном пространстве того процесса, который пытался ее изменить. Таким образом код процесса-предка не копируется полностью, а создаются только копии страниц, которые редактируются.

fork() возвращает 0 – для потомка, -1 – если ветвление невозможно, для родителя возвращается натуральное число (ID потомка). Любой процесс имеет предка, кроме демонов.

Для Unix очень важно понятие group. Процесс предок создает группу. Основная группа – терминальная, предок всех процессов в терминальной группе процесс id=1.

Процесс-сирота – процесс, у которого завершился предок.

Предок завершился, если не предпринять действий иерархия будет нарушена. Процесс сирота усыновляется терминальным процессом.

При завершении любого процесса система проверяет не осталось ли у него незавершенных потомков (по дескриптору процесса, где есть указатели на потомков). Если такие остались, то выполняется процесс усыновления, фактически изменения указатели: процесс-потомок получает указатель на нового процесса-предка.

```
int main(void)
{
    pid_t childpid1;
    pid_t childpid2;
    if ((childpid1 = fork()) == -1) /* если fork завершился успешно, pid > 0 в родительском процессе */
    {
        perror("Can't fork"); /* fork потерпел неудачу (например, память или какая-либо
                               ошибка) */
        exit(1); /* таблица заполнена */
    }
    if (childpid1 == 0)
    { /* здесь располагается дочерний код */
        printf("Child1 forked \n");
        printf("Child1 id = %d, parent id = %d, group id = %d \n", getpid(), getppid()
              (), getpgrp());
        getchar();
        printf("Child1 id = %d, parent id = %d, group id = %d \n", getpid(), getppid()
              (), getpgrp());
        return 0;
    }

    if ((childpid2 = fork()) == -1) /* если fork завершился успешно, pid > 0 в родительском процессе */
    {
        perror("Can't fork"); /* fork потерпел неудачу (например, память или какая-либо
                               ошибка) */
        exit(1); /* таблица заполнена */
    }
    if (childpid2 == 0)
    { /* здесь располагается дочерний код */
        printf("Child2 forked \n");
        printf("Child2 id = %d, parent id = %d, group id = %d \n", getpid(), getppid()
              (), getpgrp());
        getchar();
        printf("Child2 id = %d, parent id = %d, group id = %d \n", getpid(), getppid()
              (), getpgrp());
    }

    if (childpid1 != 0 && childpid2 != 0)
    { /* здесь располагается родительский код */
        printf("Parent id = %d, child1 id = %d, child2 id = %d, group id = %d \n",
               getpid(), childpid1, childpid2, getpgrp());
        getchar();
        printf("Parent exited \n");
    }

    return 0;
}
```

Системный вызов `wait(&status)` вызывается в теле предка. Системный вызов `wait()` блокирует родительский процесс до момента завершения дочернего. При их завершении предок получает статус завершения потомка.

```
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
    pid_t childpid1;
    pid_t childpid2;
    if ((childpid1 = fork()) == -1) /* если fork завершился успешно, pid > 0 в роди
```

```

тельском процессе */
{
    perror("Can't fork"); /* fork потерпел неудачу (например, память или какая-ли-
        бо */                                         /* таблица заполнена) */
    exit(1);
}
if (childpid1 == 0)
{ /* здесь располагается дочерний код */
    printf("Child1 forked \n");
    printf("Child1 id = %d, parent id = %d, group id = %d \n", getpid(), getppid
        (), getpgrp());
    getchar();
    printf("Child1 id = %d, parent id = %d, group id = %d \n", getpid(), getppid
        (), getpgrp());
    return 0;
}

if ((childpid2 = fork()) == -1) /* если fork завершился успешно, pid > 0 в роди-
тельском процессе */
{
    perror("Can't fork"); /* fork потерпел неудачу (например, память или какая-ли-
        бо */                                         /* таблица заполнена) */
    exit(1);
}
if (childpid2 == 0)
{ /* здесь располагается дочерний код */
    printf("Child2 forked \n");
    printf("Child2 id = %d, parent id = %d, group id = %d \n", getpid(), getppid
        (), getpgrp());
    getchar();
    printf("Child2 id = %d, parent id = %d, group id = %d \n", getpid(), getppid
        (), getpgrp());
}

if (childpid1 != 0 && childpid2 != 0)
{ /* здесь располагается родительский код */
    printf("Parent id = %d, child1 id = %d, child2 id = %d, group id = %d \n",
        getpid(), childpid1, childpid2, getpgrp());
    int status1;
    childpid1 = wait(&status1);
    printf("Child has finished: PID = %d\n", childpid1);

    int status2;
    childpid2 = wait(&status2);
    printf("Child has finished: PID = %d\n", childpid2);

    if (WIFEXITED(status1) && WIFEXITED(status2))
        printf("Children exited with code %d and %d\n", WEXITSTATUS(status1),
            WEXITSTATUS(status2));
    else
        printf("Child terminated abnormally\n");
}
return 0;
}

```

Чаще всего нет смысла в выполнении двух одинаковых процессов и потомок сразу выполняет системный вызов exec(), параметрами которого является имя исполняемого файла и, если нужно, параметры, которые будут переданы этой программе. Говорят, что системный вызов exec() создает низкоуровневый процесс: создаются таблицы страниц для адресного пространства программы, указанной в exec(), но программа на выполнение не запускается, так как это не полноценный процесс, имеющий идентификатор и дескриптор. Системный вызов exec() создает таблицу страниц для адресного пространства программы, переданной ему в качестве параметра, а затем заменяет старый адрес новой таблицы страниц.

Бывает шести видов: execlp, execvp, execl, execv, execle, execve.

В результате системного вызова exec() адресное пространство процесса будет заменено на адресное про-

странство новой программы, а сам процесс будет возвращен в режим задачи с установкой указателя команд на первую выполняемую инструкцию этой программы.

Возникает процесс-зомби – это процесс, у которого отобраны все ресурсы, кроме последнего – строки в таблице процессов. Если потомок завершился(аварийно завершился exec) до вызовы wait у предка, то для того чтобы предок не завис, возникает зомби. Зомби существует пока не будет выполнен wait у родителя.

```

#include <stdio.h>
#include <stdlib.h>

int main(void)
{
    pid_t childpid1;
    pid_t childpid2;
    if ((childpid1 = fork()) == -1) /* если fork завершился успешно, pid > 0 в роди-
        тельском процессе */
    {
        perror("Can't fork"); /* fork потерпел неудачу (например, память или какая-ли-
            бо */                                /* таблица заполнена) */
        exit(1);
    }
    if (childpid1 == 0)
    { /* здесь располагается дочерний код */
        printf("Child1 forked \n");
        printf("Child1 id = %d, parent id = %d, group id = %d \n", getpid(), getppid()
            (), getpgrp());
        char msg[] = "exec() called from child1";

        /* обращаемся к программе echo, параметрами передаём её имя и текст для печат-
            и */
        /* NULL - завершает список параметров */
        if (execlp("echo", "echo", msg, NULL) == -1) /* при успешном вызове exec
            () заменяет адресное пространство процесса */
        {
            perror("Can't exec"); /* - эти вызовы произойдут только в том случае,
                если exec() не выполнится*/
            exit(1);
        }
        return 0;
    }

    if ((childpid2 = fork()) == -1) /* если fork завершился успешно, pid > 0 в роди-
        тельском процессе */
    {
        perror("Can't fork"); /* fork потерпел неудачу (например, память или какая-ли-
            бо */                                /* таблица заполнена) */
        exit(1);
    }
    if (childpid2 == 0)
    { /* здесь располагается дочерний код */
        printf("Child2 forked \n");
        printf("Child2 id = %d, parent id = %d, group id = %d \n", getpid(), getppid()
            (), getpgrp());

        char msg[40] = "exec() called from child2";

        /* обращаемся к программе echo, параметрами передаём её имя и текст для печат-
            и */
        /* NULL - завершает список параметров */
        if (execlp("echo", "echo", msg, NULL) == -1) /* при успешном вызове exec() за-
            меняет адресное пространство процесса */
        {
            perror("Can't exec"); /* - эти вызовы произойдут только в том случае,
                если exec() не выполнится*/
        }
    }
}

```

```

        exit(1);
    }

if (childpid1 != 0 && childpid2 != 0)
{ /* здесь располагается родительский код */
    printf("Parent id = %d, child1 id = %d, child2 id = %d, group id = %d \n",
           getpid(), childpid1, childpid2, getpgrp());
    int status1;
    childpid1 = wait(&status1);
    printf("Child has finished: PID = %d\n", childpid1);

    int status2;
    childpid2 = wait(&status2);
    printf("Child has finished: PID = %d\n", childpid2);

    if (WIFEXITED(status1) && WIFEXITED(status2))
        printf("Children exited with code %d and %d\n", WEXITSTATUS(status1),
               WEXITSTATUS(status2));
    else
        printf("Child terminated abnormally\n");
}
return 0;
}

```

Сигнал - способ информирования процесса ядром о произшествии какого-то события. Если возникает несколько однотипных событий, процессу будет подан только один сигнал. Сигнал означает, что произошло событие, но ядро не сообщает сколько таких событий произошло.

Средство передачи сигнала – kill(), приема сигнала – signal().

```

#include <stdio.h>
#include <stdlib.h>
#include <signal.h>

int signal_flag = 0;

/* собственный обработчик сигнала ctrl-c */
void catch_sigp(int sig_num)
{
    signal(sig_num, catch_sigp);
    signal_flag = 1;
    printf("\n Parent catch sig %d\n", sig_num);
}

int main(void)
{
    pid_t childpid1;
    pid_t childpid2;
    /* дескрипторы программных каналов */
    int my_pipe1[2];
    int my_pipe2[2];
    /* [0] - выход для чтения, [1] - выход для записи */

    /* потомок унаследует открытый программный канал предка */
    if (pipe(my_pipe1) == -1)
    {
        perror("Can't pipe");
        exit(1);
    }
    if (pipe(my_pipe2) == -1)
    {
        perror("Can't pipe");
        exit(1);
    }
}

```

```

if ((childpid1 = fork()) == -1) /* если fork завершился успешно, pid > 0 в роди
тельском процессе */
{
    perror("Can't fork"); /* fork потерпел неудачу (например, память или какая-ли
бо */                                /* таблица заполнена) */
    exit(1);
}
if (childpid1 == 0)
{ /* здесь располагается дочерний код */
    printf("Child1 forked \n");
    printf("Child1 id = %d, parent id = %d, group id = %d \n", getpid(), getppid
(), getpgrp());
}

char buf[40];
close(my_pipe1[1]); /* потомок ничего не запишет в канал */
read(my_pipe1[0], buf, sizeof(buf));
printf("Child1 catch message = %s \n", buf);

return 0;
}

if ((childpid2 = fork()) == -1) /* если fork завершился успешно, pid > 0 в роди
тельском процессе */
{
    perror("Can't fork"); /* fork потерпел неудачу (например, память или какая-ли
бо */                                /* таблица заполнена) */
    exit(1);
}
if (childpid2 == 0)
{ /* здесь располагается дочерний код */
    printf("Child2 forked \n");
    printf("Child2 id = %d, parent id = %d, group id = %d \n", getpid(), getppid
(), getpgrp());
}

char buf[40];
close(my_pipe2[1]); /* потомок ничего не запишет в канал */
read(my_pipe2[0], buf, sizeof(buf));
printf("Child2 catch message = %s \n", buf);
}

if (childpid1 != 0 && childpid2 != 0)
{ /* здесь располагается родительский код */
    printf("Parent id = %d, child1 id = %d, child2 id = %d, group id = %d \n",
        getpid(), childpid1, childpid2, getpgrp());

/* ожидание сигнала */
signal(SIGINT, catch_sigp);
sleep(2);

if (!signal_flag)
{
    char msg1[] = "Hello, child1!";
    close(my_pipe1[0]); /* предок ничего не считает из канала */
    write(my_pipe1[1], msg1, sizeof(msg1));

    char msg2[] = "Hello, child2!";
    close(my_pipe2[0]); /* предок ничего не считает из канала */
    write(my_pipe2[1], msg2, sizeof(msg2));
}

int status1;
childpid1 = wait(&status1);

```

```
printf("Child has finished: PID = %d\n", childpid1);

int status2;
childpid2 = wait(&status2);
printf("Child has finished: PID = %d\n", childpid2);

if (WIFEXITED(status1) && WIFEXITED(status2))
    printf("Children exited with code %d and %d\n", WEXITSTATUS(status1),
           WEXITSTATUS(status2));
else
    printf("Child terminated abnormally\n");
}
return 0;
}
```

## 17 билет

**17.1 Виртуальная память: распределение памяти страницами по запросам, свойство локальности, рабочее множество, анализ страничного поведения процессов. Схема страничного преобразования в процессорах Intel (X86) PAE – размеры таблиц дескрипторов.**

**Виртуальная память** – память, размер которой превышает размер реального физического пространства. Используется адресное пространство диска как область свопинга или пейджинга, т.е. для временного хранения областей памяти.

Подходы к реализации управления виртуальной памятью:

1. управление памятью страницами по запросам
2. сегментами по запросам
3. сегментами, деленными на страницы по запросам

**Страница** – является единицей физического деления памяти. Её размер устанавливается системой. У страницы размер не обязательно 4Кб. Просто такой размер оптимальен по количеству страничных прерываний.

**Сегмент** – является единицей логического деления памяти. Её размер определяется объемом кода. Любой сегмент создается такого размера, который требуется программе.

**Запрос** – страничное прерывание, которое возникает при попытке доступа к незагруженной в память странице. При его обработке происходит пэйджинг.

**Пэйджинг** – загрузка с диска новых и замещение старых страниц.

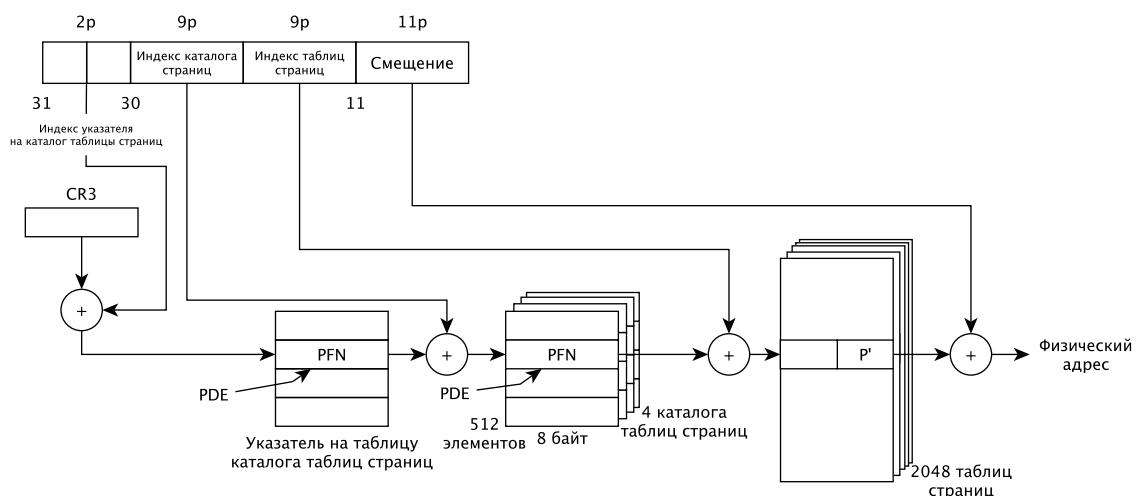
Начиная с Pentium Pro включен регистр CR4, где 5 бит это PAE (physical address extension) – расширение физического адреса.

В режиме PAE виртуальный адрес делится на 4 поля.

Если PAE = 1, то разрешено использование расширенной 36пп, вместо 32 pp физического адреса. При этом физический адрес остается 32 pp, все изменения касаются только работы страничного механизма.

PFN – Page Frame Number

PDE – Page Describe Entry



**Анализ страничного поведения процессов: свойство локальности, рабочее множество.**

**Свойство локальности:**

Если обращение было к странице, то наиболее вероятно, что следующее обращение будет к этой же странице. (Если не слишком далеко заглядывать в будущее, то можно достаточно точно его прогнозировать исходя из прошлого.)

Страницное поведение программ – число страничных прерываний, происходящих в процессе.

Зависимость числа страничных прерывания от объема памяти является обобщенной мерой страничного поведения программы.

Программа одновременно не обращается ко всем своим страницам, позволяет использовать виртуальное пространство.

В разные моменты времени обращается к разным подмножествам множества страниц. В том числе в разные моменты времени программы обращаются к разному количеству страниц. Предсказать к каким страницам обратиться невозможно.

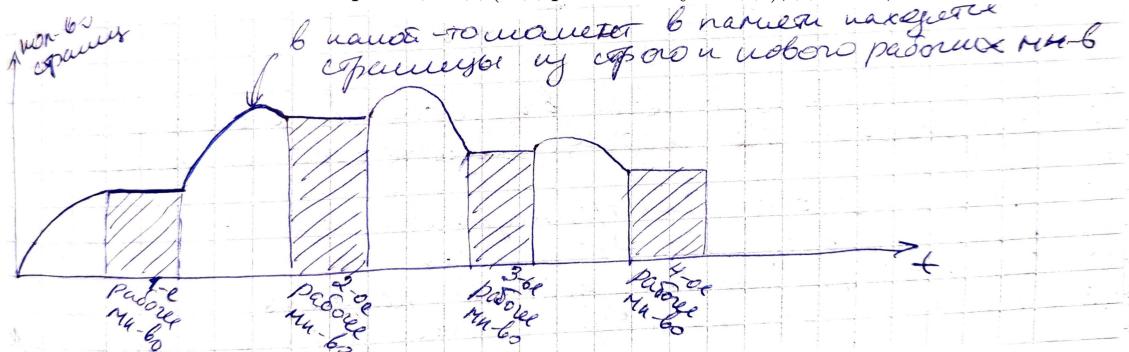
Деннинг в 1968 году предложил в качестве локальной меры производительности взять число страниц, к которым обращается программа за интервал времени  $\Delta t$ .

$$\frac{W(t, \Delta t)}{\Delta t} \rightarrow \text{число страниц}$$

Размер рабочего множества является монотонной функцией от  $\Delta t$ , то есть при увеличении интервала  $\Delta t$ , число страниц будет стремиться к пределу  $L$  – которое определяет необходимое количество страниц для выполнения программы.

Нужно, чтобы рабочее множество находилось в оперативной памяти. Трэшинг – подкачка одних и тех же страниц.

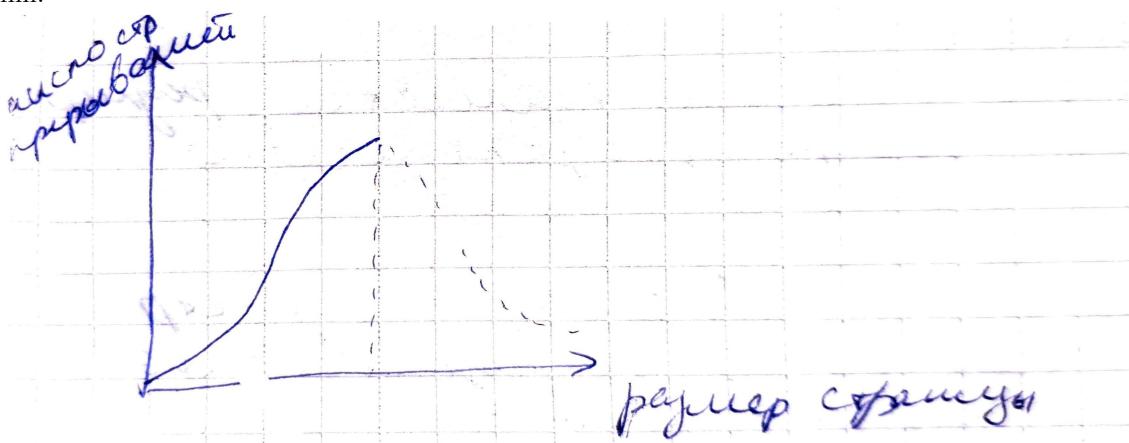
Минимальное количество страниц: код (содержит точку входа), данные, стек – 3.



Рабочее множество – набор страниц, используемых процессом в настоящее время.

Процесс удачных обращений – hit rate.

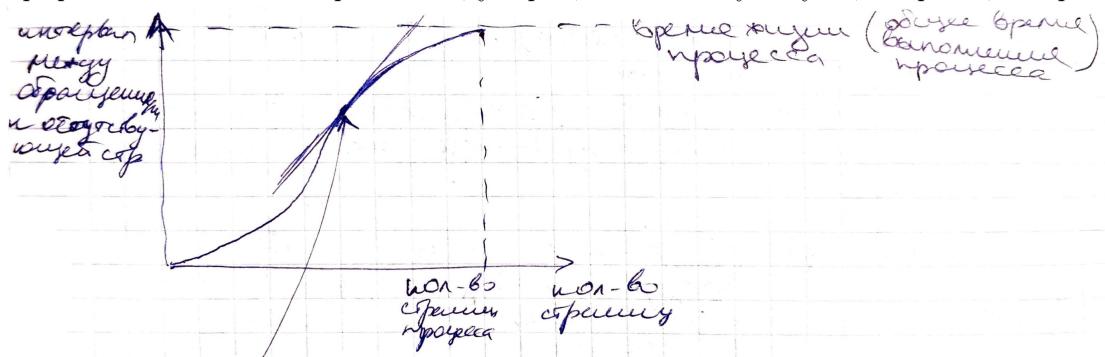
График. Влияние изменения размера страниц при сохранении объема оперативной памяти, на число прерываний.



С ростом размера страниц число страничных прерываний растет. Когда размер страницы становится соизмеримым с размером программы число страничных прерываний резко падает.

Чем меньше размер страницы тем больше страниц, но на маленьких страницах выполняется большое число команд.

График. Зависимость интервала между обращениями к отсутствующей странице за время жизни процесса.



Стрелка указывает на точку перегиба, в некоторый момент в оперативной памяти оказывается все рабочее множество процесса.

Система может контролировать количество страничных прерываний, если резко возросло, то означает что рабочее множество не может быть загружено в память.

Необходимо увеличить размер квоты на непродолжительный интервал времени.

Квота – каждому процессу может быть выделено определенное количество страниц, изначально.

## **17.2 Прерывания от системного таймера в защищенном режиме: функции (по материалам лабораторной работы).**

Т.к. некоторые из задач не требуют выполнения на каждом тике, то вводится понятие основного тика (равен n тикам), часть задач выполняется только при основном тике.

### **По тику**

1. Обновление системного времени (инкремент);
2. Декремент показания счетчика, отслеживающего продолжительность работы текущего потока;
3. Декремент показаний счетчиков отложенных задач.

### **По главному тику**

1. Инициализация диспетчера настройки баланса, который активизируется каждую секунду для возможной инициации событий, связанных с планированием и управлением памятью;

### **По кванту**

1. Инициализация диспетчеризации потоков (добавление соответствующего объекта DPC в очередь).

### **17.2.1 Функции обработчика прерываний от системного таймера в Unix/Linux**

#### **По тику**

1. Обновление статистики использования процессора текущим процессом;
2. Обновление часов и других таймеров системы;
3. Обработка отложенных вызовов;
4. Ведение счета тиков аппаратного таймера по необходимости.

#### **По главному тику**

1. Выполнение функций, относящихся к работе планировщика, таких как пересчет приоритетов и действия, выполняющиеся по истечении выделенного кванта времени;
2. Пробуждение в нужные моменты системных процессов, такие как swapper и pagedaemon;
3. Обработка сигнала тревоги (будильники).

#### **По кванту**

1. Посылка текущему процессу сигнала SIGXCPU, если тот превысил выделенную ему квоту(квант) использования процессора (процессорного времени).

## 18 билет

### 18.1 Процессы: взаимодействие параллельных процессов – монопольный доступ и взаимоисключение; программная реализация взаимоисключения – флаги, алгоритм Деккера, алгоритм Лампорта.

Чтобы не терять значение разделяемой переменной, необходимо обеспечить **монопольный доступ** процессов к **разделяемым переменным ресурсам** методами взаимоисключения. Только один процесс может получить доступ к данным в один момент. Те участки кода, в которых выполняется обращение к разделяемым данным называются **критическими**. Только один процесс может находиться в критическом участке.

Существуют следующие способы взаимоисключения:

1. Программный способ
2. Аппаратный способ
3. С помощью семафоров
4. С использованием мониторов

**Активное ожидание на процессоре** – ситуация, когда процесс занимает процессорное время, проверяя значение флага (занятости ресурса другим процессом). Активное ожидание на процессоре является неэффективным использованием процессорного времени.

#### Флаги

```
flagp1, flagp2 : logical
P1:
while(1)
{
    while (flagp2);
    flagp1 = 1;
    CR1;
    flagp1 = 0;
}
PR1;

P2:
while(1)
{
    while (flagp1);
    flagp2 = 1;
    CR2;
    flagp2 = 0;
}
PR2;

// начальные значения
flagp1 = 0;
flagp2 = 0;
parbegin;
    P1; P2;
parend;
```

Способ не работает, также теряются единицы.

Пусть процесс устанавливает флаг до проверки.

```
P1:
while(1)
{
    flagp1 = 1;
    while (flagp2);
    CR1; // критическая секция
    flagp1 = 0;
}
PR1;
```

```

P2:
while(1)
{
    flagp2 = 1;
    while (flagp1);
    CR2; // критическая секция
    flagp2 = 0;
}
PR2;

// начальные значения
flagp1 = 0;
flagp2 = 0;
parbegin;
    P1; P2;
parend;

```

Не могут выполняться, оба ждут освобождения (туниковая ситуация).

Проблемы:

- Для проверки флага тратится процессорное время (активное ожидание на процессоре) непроизводительный расход процессорного времени
- Можно попасть в туниковую ситуацию, каждый процесс ждёт освобождения флага другого процесса и ни один из них не может продолжить свою работу

Эту проблему удалось решить голландскому математику Деккеру, алгоритм был назван **алгоритм Деккера**. Но решил только для двух параллельных процессов. Он ввёл флаги и дополнительную переменную, которую назвал «чья очередь».

```

flagp1, flagp2: logical;
que:int;

p1:
while(1)
{
    flagp1=1;
    while(flagp2)
    {
        if (que == 2)
        {
            flagp1=0;
            while(que == 2);
            flagp1=1;
        }
    }
    CR1;
    flagp1=0;
    que=2;
}

...
p2:
while(1)
{
    flagp2=1;
    while(flagp1)
    {
        if (que == 1)
        {
            flagp2=0;
            while(que == 1);
            flagp2=1;
        }
    }
    CR2;
}

```

```

    flagp2=0;
    que=1;
}
...

// начальные значения
flagp1 = 0;
flagp2 = 0;
que=1;
parbegin;
    P1; P2;
parend;

```

### Алгоритм bakery(булочная) – Лампорта

Каждому клиенту выдается листок с номером, новому – с большим номером. Когда продавец освобождается, то обслуживает клиента с меньшим номером.

Бывает, что одновременно приходят 2 клиента, в этом случае им выдаются одинаковые номера, но при обслуживании, обслуживает клиента с меньшим номером паспорта, к примеру.

```

1 var choosing: shared array [0..n-1] of boolean;
2         number: shared array [0..n-1] of integer;
3 repeat

4 choosing[i]:=true;
5 number[i]:=max(number[0],...,number[n-1])+1;
6 choosing[i]:=false;

7 for j:=0 to n-1 do
     begin
8         while choosing[j] do /*nothing*/;
9         while(number[j] <> 0) and
10            (number[j],j)<(number[i],i) do
11            /*nothing*/;
12     end;
13 /*critical section*/
14 number[i] :=0;
15 /*remainder section*/

16 until false;

```

1 и 2 определяет массивы флагов и целых чисел.

choosing позволяет нам определять критическую секцию choosing[i]=true, если процесс p[i] выбирает номер. Номер, которые использует p[i] для входа в критический участок, это number[i].

number[i]=0, если p[i] не пытается войти в свой критический участок.

4,5,6 строки показывают, что процесс выбирает номер и устанавливает свой флаг choosing[i] в true. После этого он пытается получить уникальный номер, максимальный среди выданных +1, если это возможно. Получив номер, он сбрасывает свой флаг (строчка 6).

7-12 определяет какой процесс входит в критический участок. Процесс p[i] ждет пока имеется процесс, у которого меньший номер, для того чтобы войти в критический участок. Если 2 процесса имеют одинаковые номера, то выбирается процесс с меньшим идентификатором.

Запись 10 – лексикографический порядок ((a,b) < (c,d), если (a<c) или (a=c, b<d) )

Также необходимо обратить внимание на то, когда 2 процесса выбирают номер, то 2 процесс будет ждать, пока 1 не сделает (строчка 8).

14 строка – процесс p[i] больше не заинтересован во входе в критический участок.

## 18.2 Защищенный режим: перевод компьютера в защищенный режим – реализация – пример кода из лабораторной работы.

```

.386p ; Разрешение трансляции всех, в том
        ; числе привилегированных команд МП 386
        ; и 486

descr struc ; Структура для описания дескриптора сегмента
    limit      dw 0      ; Граница (биты 0..15)

```

```

base_l    dw 0      ; База, биты 0..15
base_m    db 0      ; База, биты 16..23
attr_1    db 0      ; Байт атрибутов 1
arrt_2    db 0      ; Граница(биты 16..19) и атрибуты 2
base_h    db 0      ; База, биты 24..31
descr ends

data segment ; Начало сегмента данных
; Таблица глобальных дескрипторов GDT
gdt_null descr <0,0,0,0,0,0> ; Нулевой дескриптор
gdt_data descr <data_size-1,0,0,92h,0,0> ; Сегмент данных, селектор 8
gdt_code descr <code_size-1,0,0,98h,0,0> ; Сегмент команд, селектор 16
gdt_stack descr <255,0,0,92h,0,0> ; Сегмент стека, селектор 24
gdt_screen descr <4095,8000h,0Bh,92h,0,0> ; Видеобуфер, селектор 32
gdt_size=$-gdt_null ; Размер GDT
; Поля данных программы
pdescr dq 0 ; Псевдодескриптор
mes db "Real mode$" ; Сообщение для вывода в режиме реальном
mes1 db "Protected mode$" ; Сообщение для вывода в режиме защищенном
data_size=$-gdt_null ; Размер сегмента данных
data ends ; Конец сегмента данных

text segment 'code' use16 ; 16-разрядный режим
; в первой части нам его достаточно
assume CS:text, DS:data

main proc
    xor EAX,EAX ; Очистим EAX
    mov AX,data ; Загрузим в DS адрес сегмента данных
    mov DS,AX ; Адрес сегмента данных
; Вычислим 32-битовый линейный адрес сегмента данных и загрузим
; в дескриптор сегмента данных в GDT
    shl EAX,4 ; В EAX линейный базовый адрес
    mov EBP,EAX ; Сохраним его в EBP
    mov EBX,offset gdt_data ; В EBX адрес дескриптора
    mov [BX].base_l,AX ; Загрузим младшую часть базы
    rol EAX,16 ; Обмен старшей и младшей половин в EAX
    mov [BX].base_m,AL ; Загрузим среднюю часть базы
; Аналогично для сегмента команд
    xor EAX,EAX
    mov AX,CS
    shl EAX,4
    mov EBX,offset gdt_code
    mov [BX].base_l,AX
    rol EAX,16
    mov [BX].base_m,AL
; Аналогично для сегмента стека
    xor EAX,EAX
    mov AX, SS
    shl EAX,4
    mov EBX,offset gdt_stack
    mov [BX].base_l,AX
    rol EAX,16
    mov [BX].base_m,AL
; Подготовим псевдодескриптор pdescr и загрузим регистр GDTR
    mov dword ptr pdescr+2,EBP ; База GDT, биты 0..31
    mov word ptr pdescr,gdt_size-1 ; Граница GDT
    lgdt pdescr ; Загрузим регистр GDTR
; Подготовимся к переходу в защищенный режим
    cli ; Запрет аппаратных прерывания (маскируемых)
    mov AL,80h ; Запрет NMI, немаскируемых прерываний
    out 70h,AL
; Переходим в защищенный режим

```

```

        mov EAX,CRO ; Получим содержимое CRO
        or EAX,1 ; Установим бит PE
        mov CRO,EAX ; Запишем назад
; Теперь процессор работает в защищенном режиме
; Загружаем в CS:IP селектор:смещение точки continue
; и заодно очищаем очередь команд
        db 0EAh ; Код команды far jmp
        dw offset continue ; смещение
        dw 16 ; селектор сегмента команд

continue:
; Делаем адресуемыми данные
        mov AX,8 ; Селектор сегмента данных
        mov DS,AX
; Делаем адресуемым стек
        mov AX,24 ; Селектор сегмента стека
        mov SS,AX
; Делаем адресуемыми видеобуфер и выводим сообщение о преходе
        mov AX,32 ; Селектор сегмента видеобуфера
        mov ES,AX
        mov BX,800 ; Начальное смещение на экране
        mov CX,15 ; Число выводимых символов
        mov SI,0 ; Итератор (смещение от начала)

screen:
        mov EAX,word ptr mes1[SI] ; Символ для вывода со смещением SI
        mov ES:[BX],EAX ; Вывод в видеобуфер
        add BX,2 ; Смещаемся в видеобуфере
        inc SI ; Следующий символ строки
        loop screen ; Цикл вывода на экран
; Подготовим переход в реальный режим
; Сформируем и загрузим дескриптор для реального режима
; В нашем варианте не обязательно загружать FFFF, так как 16 битная система
        mov gdt_data.limit,0FFFFh ; Граница сегмента данных
        mov gdt_code.limit,0FFFFh ; Граница сегмента кода
        mov gdt_stack.limit,0FFFFh ; Граница сегмента стека
        mov gdt_screen.limit,0FFFFh ; Граница сегмента видеобуфера

        mov AX,8 ; Загрузим теневой регистр
        mov DS,AX ; сегмента данных
        mov AX,24 ; Загрузим теневой регистр
        mov SS,AX ; сегмента стека
        mov AX,32 ; Загрузим теневой регистр
        mov ES,AX ; сегмента видеобуфера
; Выполним дальний переход для того, чтобы заново
; загрузить селектор в регистр CS и модифицировать его теневой
        db 0EAh ; Командой дальнего перехода
        dw offset go ; загрузим теневой регистр
        dw 16 ; сегмента команд
; Переключим режим процессора
go:
        mov EAX,CRO ; Получим содержимое CRO
        and EAX,0FFFFFFFEh ; Сбросим бит PE
        mov CRO,EAX ; Запишем назад
        db 0EAh ; Код команды far jmp
        dw offset return ; Смещение
        dw text ; Сегмент
; Теперь процессор снова работает в реальном режиме
; Восстановим операционную среду
return:
        mov AX,data ; Восстановим
        mov DS,AX ; адресуемость данных
        mov AX,stk ; Восстановим

```

```
    mov SS,AX ; адресуемость стека
; Разрешим аппаратные(маскируемые) и немаскируемые прерывания
    sti ; Разрешение аппаратных
    mov AL,0 ; Разрешение немаксируемых
    out 70h,AL
; Проверим выполнение функций DOS после возврата в реальный режим
    mov AH,09h ; Вывод сообщения
    mov EDX,offset mes
    int 21h
    mov AX,4C00h ; Завершение программы
    int 21h
main endp

    code_size=$-main ; Размер сегмента кода(команд)
text ends

stk segment stack 'stack' ; Сегмент стека
    db 256 dup ('^')
stk ends

end main
```

## 19 билет

**19.1 Процессы Unix: создание процесса в ОС Unix и запуск новой программы.**  
Примеры программ из лабораторных работ, демонстрирующих эти действия.  
**Системные вызовы wait() и pipe(): назначение, примеры из лабораторных работ. Процессы «сироты», «зомби» и «демоны».**

### Создание процесса

Любой процесс создается системным вызовом fork(), который определяется в системе как потомок (child). В Unix используется иерархия процессов, которая строится в отношении предок-потомок.

В результате вызова fork создается процесс-потомок. В Unix все рассматривается как файл (файлы, директории, устройства).

В современных системах применяется, так называемая, оптимизация fork(): для процесса потомка создаются собственные карты трансляции адресов (таблицы страниц), но они ссылаются на адресное пространство процесса-предка (на страницы предка). При этом для страниц адресного пространства предка права доступа меняются на only-read и устанавливается флаг — copy-on-write. Если или предок или потомок попытаются изменить страницу, возникнет исключение по правам доступа. Выполняя это исключение супервизор обнаружит флаг copy-on-write и создаст копию страницы в адресном пространстве того процесса, который пытался ее изменить. Таким образом код процесса-предка не копируется полностью, а создаются только копии страниц, которые редактируются.

fork() возвращает 0 — для потомка, -1 — если ветвление невозможно, для родителя возвращается натуральное число (ID потомка). Любой процесс имеет предка, кроме **демонов**.

**Процессы демоны** — процессы которые не имеют родителей, они существуют сами и не входят ни в какие группы. Демон — процесс, выполняющий какую-то фоновую задачу, не имеющий управляющего терминала и, как следствие, обычно не интерактивный по отношению к пользователю.

Для Unix очень важно понятие group. Процесс предок создает группу. Основная группа — терминальная, предок всех процессов в терминальной группе процесс id=1.

### Процесс-сирота

Предок завершился, если не предпринять действий иерархия будет нарушена. Процесс сирота усыновляется терминальным процессом.

При завершении любого процесса система проверяет не осталось ли у него незавершенных потомков (по дескриптору процесса, где есть указатели на потомков). Если такие остались, то выполняется процесс усыновления, фактически изменения указатели: процесс-потомок получает указатель на нового процесса-предка.

```
int main(void)
{
    pid_t childpid1;
    pid_t childpid2;
    if ((childpid1 = fork()) == -1) /* если fork завершился успешно, pid > 0 в родительском процессе */
    {
        perror("Can't fork"); /* fork потерпел неудачу (например, память или какая-либо */
        /* таблица заполнена) */
        exit(1);
    }
    if (childpid1 == 0)
    { /* здесь располагается дочерний код */
        printf("Child1 forked \n");
        printf("Child1 id = %d, parent id = %d, group id = %d \n", getpid(), getppid()
            (), getpgrp());
        getchar();
        printf("Child1 id = %d, parent id = %d, group id = %d \n", getpid(), getppid()
            (), getpgrp());
        return 0;
    }

    if ((childpid2 = fork()) == -1) /* если fork завершился успешно, pid > 0 в родительском процессе */
    {
        perror("Can't fork"); /* fork потерпел неудачу (например, память или какая-либо */
        /* таблица заполнена) */
        exit(1);
    }
    if (childpid2 == 0)
```

```

{ /* здесь располагается дочерний код */
    printf("Child2 forked \n");
    printf("Child2 id = %d, parent id = %d, group id = %d \n", getpid(), getppid
        (), getpgrp());
    getchar();
    printf("Child2 id = %d, parent id = %d, group id = %d \n", getpid(), getppid
        (), getpgrp());
}

if (childpid1 != 0 && childpid2 != 0)
{ /* здесь располагается родительский код */
    printf("Parent id = %d, child1 id = %d, child2 id = %d, group id = %d \n",
        getpid(), childpid1, childpid2, getpgrp());
    getchar();
    printf("Parent exited \n");
}

return 0;
}

```

### Запуск новой программы

Чаще всего нет смысла в выполнении двух одинаковых процессов и потомок сразу выполняет системный вызов `exec()`, параметрами которого является имя исполняемого файла и, если нужно, параметры, которые будут переданы этой программе. Говорят, что системный вызов `exec()` создает низкоуровневый процесс: создаются таблицы страниц для адресного пространства программы, указанной в `exec()`, но программа на выполнение не запускается, так как это не полноценный процесс, имеющий идентификатор и дескриптор. Системный вызов `exec()` создает таблицу страниц для адресного пространства программы, переданной ему в качестве параметра, а затем заменяет старый адрес новой таблицы страниц.

Бывает шести видов: `execlp`, `execvp`, `execl`, `execv`, `execle`, `execve`.

В результате системного вызова `exec()` адресное пространство процесса будет заменено на адресное пространство новой программы, а сам процесс будет возвращен в режим задачи с установкой указателя команд на первую выполняемую инструкцию этой программы.

Возникает **процесс-зомби** – это процесс, у которого отобраны все ресурсы, кроме последнего – строки в таблице процессов. Если потомок завершился (аварийно завершился `exec`) до вызовы `wait` у предка, то для того чтобы предок не завис, возникает зомби. Зомби существует пока не будет выполнен `wait` у родителя.

```

#include <stdio.h>
#include <stdlib.h>

int main(void)
{
    pid_t childpid1;
    pid_t childpid2;
    if ((childpid1 = fork()) == -1) /* если fork завершился успешно, pid > 0 в роди
        тельском процессе */
    {
        perror("Can't fork"); /* fork потерпел неудачу (например, память или какая-ли
            бо */                                         /* таблица заполнена */
        exit(1);
    }
    if (childpid1 == 0)
    { /* здесь располагается дочерний код */
        printf("Child1 forked \n");
        printf("Child1 id = %d, parent id = %d, group id = %d \n", getpid(), getppid
            (), getpgrp());

        char msg[] = "exec() called from child1";

        /* обращаемся к программе echo, параметрами передаём её имя и текст для печат
            и */
        /* NULL - завершает список параметров */
        if (execlp("echo", "echo", msg, NULL) == -1) /* при успешном вызове exec
            () заменяет адресное пространство процесса */
        {

```

```

        perror("Can't exec"); /* - эти вызовы произойдут только в том случае,
        если exec() не выполнится*/
    exit(1);
}
return 0;
}

if ((childpid2 = fork()) == -1) /* если fork завершился успешно, pid > 0 в роди-
тельском процессе */
{
    perror("Can't fork"); /* fork потерпел неудачу (например, память или какая-ли-
бо)*/
    exit(1); /* таблица заполнена)*/
}
if (childpid2 == 0)
{ /* здесь располагается дочерний код */
    printf("Child2 forked \n");
    printf("Child2 id = %d, parent id = %d, group id = %d \n", getpid(), getppid()
(), getpgrp());
}

char msg[40] = "exec() called from child2";

/* обращаемся к программе echo, параметрами передаём её имя и текст для печат-
и*/
/* NULL - завершает список параметров */
if (execlp("echo", "echo", msg, NULL) == -1) /* при успешном вызове exec() за-
меняет адресное пространство процесса */
{
    perror("Can't exec"); /* - эти вызовы произойдут только в том случае,
если exec() не выполнится*/
    exit(1);
}

if (childpid1 != 0 && childpid2 != 0)
{ /* здесь располагается родительский код */
    printf("Parent id = %d, child1 id = %d, child2 id = %d, group id = %d \n",
getpid(), childpid1, childpid2, getpgrp());
    int status1;
    childpid1 = wait(&status1);
    printf("Child has finished: PID = %d\n", childpid1);

    int status2;
    childpid2 = wait(&status2);
    printf("Child has finished: PID = %d\n", childpid2);

    if (WIFEXITED(status1) && WIFEXITED(status2))
        printf("Children exited with code %d and %d\n", WEXITSTATUS(status1),
WEXITSTATUS(status2));
    else
        printf("Child terminated abnormally\n");
}
return 0;
}

```

**Системный вызов `wait(&status)`** вызывается в теле предка. Системный вызов `wait()` блокирует родительский процесс до момента завершения дочернего. При их завершении предок получает статус завершения ПОТОМКА.

```

#include <stdio.h>
#include <stdlib.h>

int main(void)
{

```

```

pid_t childpid1;
pid_t childpid2;
if ((childpid1 = fork()) == -1) /* если fork завершился успешно, pid > 0 в роди
тельском процессе */
{
    perror("Can't fork"); /* fork потерпел неудачу (например, память или какая-ли
бо */                                /* таблица заполнена) */
}
if (childpid1 == 0)
{ /* здесь располагается дочерний код */
    printf("Child1 forked \n");
    printf("Child1 id = %d, parent id = %d, group id = %d \n", getpid(), getppid
        (), getpgrp());
    getchar();
    printf("Child1 id = %d, parent id = %d, group id = %d \n", getpid(), getppid
        (), getpgrp());
    return 0;
}

if ((childpid2 = fork()) == -1) /* если fork завершился успешно, pid > 0 в роди
тельском процессе */
{
    perror("Can't fork"); /* fork потерпел неудачу (например, память или какая-ли
бо */                                /* таблица заполнена) */
}
if (childpid2 == 0)
{ /* здесь располагается дочерний код */
    printf("Child2 forked \n");
    printf("Child2 id = %d, parent id = %d, group id = %d \n", getpid(), getppid
        (), getpgrp());
    getchar();
    printf("Child2 id = %d, parent id = %d, group id = %d \n", getpid(), getppid
        (), getpgrp());
}

if (childpid1 != 0 && childpid2 != 0)
{ /* здесь располагается родительский код */
    printf("Parent id = %d, child1 id = %d, child2 id = %d, group id = %d \n",
        getpid(), childpid1, childpid2, getpgrp());
    int status1;
    childpid1 = wait(&status1);
    printf("Child has finished: PID = %d\n", childpid1);

    int status2;
    childpid2 = wait(&status2);
    printf("Child has finished: PID = %d\n", childpid2);

    if (WIFEXITED(status1) && WIFEXITED(status2))
        printf("Children exited with code %d and %d\n", WEXITSTATUS(status1),
            WEXITSTATUS(status2));
    else
        printf("Child terminated abnormally\n");
}
return 0;
}

```

**Системный вызов pipe()** создает неименованный программный канал. Неименованные программные каналы могут использоваться для обмена сообщениями между процессами родственниками. В отличие от именованных программных каналов неименованные не имеют идентификатора, но имеют дескриптор. Процесс-потомок наследует все дескрипторы открытых файлов процесса-предка, в том числе и неименованных программных каналов.

Программные каналы имеют встроенные средства взаимоисключений: из канала нельзя читать, если в него

пишут, и в канал нельзя писать, если из него читают.

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

#define ANSI_COLOR_GREEN "\x1b[32m"
#define ANSI_COLOR_RESET "\x1b[0m"

#define N 20

int main()
{
    int status;
    pid_t childpid1, childpid2;
    char buffer[N];
    char message1[N] = "hello";
    char message2[N] = "goodbye";
    int channel1[2], channel2[2];

    if (pipe(channel1) == -1)
    {
        perror("Can not pipe.\n");
        return 1;
    }

    childpid1 = fork();

    if (childpid1 == -1)
    {
        perror("Can not fork.\n");
        return 1;
    }
    else if (childpid1 == 0)
    {
        close(channel1[0]);
        write(channel1[1], message1, sizeof(message1));
        exit(0);
    }

    if (pipe(channel2) == -1)
    {
        perror("Can not pipe.\n");
        return 1;
    }

    childpid2 = fork();

    if (childpid2 == -1)
    {
        perror("Can't fork.\n");
        return 1;
    }
    else if (childpid2 == 0)
    {
        close(channel2[0]);
        write(channel2[1], message2, sizeof(message2));
        exit(0);
    }
    else
    {
        wait(&status);
```

```

if (WIFEXITED(status))
{
    printf(ANSI_COLOR_GREEN "child process exit success\n"
           ANSI_COLOR_RESET);

    close(channel1[1]);
    read(channel1[0], buffer, sizeof(buffer));
    printf("message = %s\n", buffer);

    close(channel2[1]);
    read(channel2[0], buffer, sizeof(buffer));
    printf("message = %s\n", buffer);
}

return 0;
}

```

## 19.2 Взаимодействие параллельных процессов: мониторы – определение; монитор Хоара «читатели-писатели», реализация в для ОС Windows – пример из лабораторной работы.

Чтобы не терять значение разделяемой переменной, необходимо обеспечить **монопольный доступ** процессов к **разделяемым переменным ресурсам** методами взаимоисключения. Только один процесс может получить доступ к данным в один момент. Те участки кода, в которых выполняется обращение к разделяемым данным называются **критическими**. Только один процесс может находиться в критическом участке.

Существуют следующие способы взаимоисключения:

1. Программный способ
2. Аппаратный способ
3. С помощью семафоров
4. С использованием мониторов

**Активное ожидание на процессоре** – ситуация, когда процесс занимает процессорное время, проверяя значение флага (занятости ресурса другим процессом). Активное ожидание на процессоре является неэффективным использованием процессорного времени.

Возможные варианты развития событий:

1. Возможно, что оба процесса пройдут цикл ожидания и попадут в свои критические секции.
2. Возможно бесконечное откладывание (зависание) – ситуация, когда разделённый ресурс снова захватывается тем же процессом.
3. Тупик (deadlock, взаимоблокировка) – ситуация, когда оба процесса установили флаги занятости и ждут. Т.е. каждый ожидает освобождения ресурса, занятого другим процессом.

### Мониторы

Проблема связанная с семафорами и аналогичными средствами которые дают ОС состоит в том, что они не структурные, программа может утратить блокировку и синхронизацию.

Чтобы устранить эту проблему была придумана концепция мониторов, которая была реализована на нескольких языках программирования.

Мониторы придуманы для структурирования средств взаимоисключения. В результате синхронизация и связь выполняется в мониторе.

Потенциальные ошибки программирования связаны с программированием самого монитора. Их легче выявлять.

Отрицательные стороны: на обращение к функциям монитора тратится больше времени.

Идея монитора заключается в создании механизма, который унифицировал бы создание параллельных процессов по данным и функциям, которые обрабатывают эти данные.

Монитор – это языковая конструкция, состоящая из структуры данных и набора подпрограмм, которые могут обращаться к полям этой структуры.

Мониторы обозначаются ключевым словом monitor.

Монитор защищает свои переменные, так как доступ к полям мониторов может быть осуществлён только с помощью подпрограммы (функции) монитора.

На мониторах чаще всего определяется переменная типа условия. Используются две функции wait и signal.

### Монитор: читатели-писатели

2 типа процессов:

- Читатели – читают, поскольку они информацию не меняют, то они не мешают друг другу и могут читать параллельно.
- Писатель (только 1) – изменяет данные в режиме взаимоисключения (читать и писать нельзя).

Классический монитор Хоара. В нем описывается 4 функции: start\_read, stop\_read, start\_write, stop\_write.

```
resource: monitor;

var
    nr:int; // читатели
    wrt:logical; // писатели
    can_read,can_write:conditional;

procedure start_read;
begin
    if wrt or turn(can_write) then // очередь ждущих писателей
        wait(can_read);
    nr:=nr+1;
    signal(can_read); // цепная реакция читателей, каждый активизирует следующего
                       // читателя
end;

procedure stop_read;
begin
    nr:=nr-1;
    if (nr == 0) then // если читателей нет
        signal(can_write); // можно писать
end;

procedure start_write;
begin
    if nr > 0 or wrt then // есть ли активный читатель или активный писатель
        wait(can_write);
    wrt:=true;
end;

procedure stop_write;
begin
    wrt:=false;
    if (turn(can_read)) then // есть ли ждущие читатели (очередь)
        signal(can_read); // активизируются ждущие читатели
    else
        signal(can_write); // активизируется читатель
end;

begin
    nr:=0;
    wrt:=false;
end;

#include <stdio.h>
#include <windows.h>
#include <iostream>
using namespace std;

#define OK 0
#define ERROR 1

#define WRITERS 3
```

```

#define READERS 5
#define ITERS 5

int val = 0;
bool activewriter = false;
int readercount = 0;

int waiting_writers = 0;
int waiting_readers = 0;

HANDLE writers[WRITERS];
HANDLE readers[READERS];
HANDLE can_write;
HANDLE can_read;

void start_read() {
    InterlockedIncrement(&waiting_readers);
    if (true == activewriter || waiting_writers > 0) {
        WaitForSingleObject(can_read, INFINITE);
    }
    InterlockedDecrement(&waiting_readers);
    InterlockedIncrement(&readercount);
    SetEvent(can_read);
}

void stop_read() {
    InterlockedDecrement(readercount);
    if (0 == readercount) {
        SetEvent(can_write);
    }
}

void start_write() {
    InterlockedIncrement(&waiting_writers);
    if (readercount > 0 || true == activewriter) {
        WaitForSingleObject(can_write, INFINITE);
    }
    InterlockedDecrement(&waiting_writers);
    activewriter = true;
}

void stop_write() {
    activewriter = false;
    ResetEvent(can_write);
    if (waiting_readers > 0) {
        SetEvent(can_read);
    } else {
        SetEvent(can_write);
    }
}

DWORD WINAPI writer(LPVOID) {
    for (int i = 0; i < ITERS; i++) {
        start_write();
        val++;
        cout << "Writer" << GetCurrentThreadId() << " write " << val << endl;
        stop_write();
        Sleep(100);
    }
    return OK;
}

DWORD WINAPI reader(LPVOID mutex) {

```

```

for (int i = 0; i < ITERS + 7; i++) {
    start_read();
    WaitForSingleObject(mutex, INFINITE);
    cout << "Reader" << GetCurrentThreadId() << " read " << val << endl;
    ReleaseMutex(mutex);
    stop_read();
    Sleep(100);
}
return OK;
}

int create_mutex_threads() {
HANDLE mutex = CreateMutex(NULL, FALSE, NULL);
if (NULL == mutex) {
    cout << "Can't create mutex\n";
    return ERROR;
}
// создание писателей
for (int i = 0; i < WRITERS; i++) {
    // данный аргумент определяет, может ли создаваемый поток быть унаследован
    // дочерним процессом
    // размер стека в байтах. Если передать 0, то будет использоваться значение
    // по - умолчанию (1 мегабайт)
    // адрес функции, которая будет выполняться потоком
    // указатель на переменную, которая будет передана в поток
    // флаги создания
    // указатель на переменную, куда будет сохранён идентификатор потока

    writers[i] = CreateThread(NULL, 0, &writer, NULL, 0, NULL);
    if (NULL == writers[i]) {
        cout << "Can't create threads\n";
        return ERROR;
    }
}
// создание читателей
for (int i = 0; i < READERS; i++) {
    readers[i] = CreateThread(NULL, 0, &reader, mutex, 0, NULL);
    if (NULL == readers[i]) {
        cout << "Can't create threads\n";
        return ERROR;
    }
}

return OK;
}

int create_events() {
// атрибут защиты
// тип сброса TRUE - ручной
// начальное состояние TRUE - сигнальное
// имя объекта

// с автосбросом
can_read = CreateEvent(NULL, FALSE, FALSE, TEXT("ReadEvent"));
if (can_read == NULL) {
    cout << "Can't create event\n";
    return ERROR;
}

// с ручным сбросом
can_write = CreateEvent(NULL, TRUE, FALSE, TEXT("WriteEvent"));
if (can_write == NULL) {
    cout << "Can't create event\n";
}

```

```
        return ERROR;
    }
    return OK;
}

int main() {

    if (create_events() != OK) {
        return ERROR;
    }
    if (create_mutex_threads() != OK) {
        return ERROR;
    }

    WaitForMultipleObjects(WRITERS, writers, TRUE, INFINITE);
    WaitForMultipleObjects(READERS, readers, TRUE, INFINITE);

    return OK;
}
```

Семейство Interlocked-функций используются в случае, если разным потокам необходимо изменять одну и ту же переменную.

## 20 билет

**20.1 Процессы: взаимодействие параллельных процессов – монопольный доступ и взаимоисключение; программная реализация взаимоисключения – примеры, семафоры – определение, виды семафоров, примеры использования множественных семафоров из лабораторных работ «производство-потребление» и «читатели-писатели».**

Чтобы не терять значение разделяемой переменной, необходимо обеспечить **монопольный доступ** процессов к **разделяемым переменным ресурсам** методами взаимоисключения. Только один процесс может получить доступ к данным в один момент. Те участки кода, в которых выполняется обращение к разделяемым данным называются **критическими**. Только один процесс может находиться в критическом участке.

Существуют следующие способы взаимоисключения:

1. Программный способ
2. Аппаратный способ
3. С помощью семафоров
4. С использованием мониторов

### Программная реализация взаимоисключения. Флаги

```
flagp1, flagp2 : logical
P1:
while(1)
{
    while (flagp2);
    flagp1 = 1;
    CR1;
    flagp1 = 0;
}
PR1;

P2:
while(1)
{
    while (flagp1);
    flagp2 = 1;
    CR2;
    flagp2 = 0;
}
PR2;

// начальные значения
flagp1 = 0;
flagp2 = 0;
parbegin;
    P1; P2;
parend;
```

Способ не работает, также теряются единицы.

Пусть процесс устанавливает флаг до проверки.

```
P1:
while(1)
{
    flagp1 = 1;
    while (flagp2);
    CR1; // критическая секция
    flagp1 = 0;
}
PR1;

P2:
```

```

while(1)
{
    flagp2 = 1;
    while (flagp1);
    CR2; // критическая секция
    flagp2 = 0;
}
PR2;

// начальные значения
flagp1 = 0;
flagp2 = 0;
parbegin;
    P1; P2;
parend;

```

Не могут выполняться, оба ждут освобождения (тупиковая ситуация).

Проблемы:

1. Для проверки флага тратится процессорное время (активное ожидание на процессоре) непроизводительный расход процессорного времени
2. Можно попасть в тупиковую ситуацию, каждый процесс ждёт освобождения флага другого процесса и ни один из них не может продолжить свою работу

**Активное ожидание на процессоре** – ситуация, когда процесс занимает процессорное время, проверяя значение флага (занятости ресурса другим процессом). Активное ожидание на процессоре является неэффективным использованием процессорного времени.

Возможные варианты развития событий:

1. Возможно, что оба процесса пройдут цикл ожидания и попадут в свои критические секции.
2. Возможно бесконечное откладывание (зависание) – ситуация, когда разделённый ресурс снова захватывается тем же процессом.
3. Тупик (deadlock, взаимоблокировка) – ситуация, когда оба процесса установили флаги занятости и ждут. Т.е. каждый ожидает освобождения ресурса, занятого другим процессом.

### 20.1.1 Семафоры

(ввел Дейкстра в 1965 г.)

Семафор – неотрицательная защищённая переменная S, над которой определено 2 неделимые операции: P (от датск. passeren - пропустить) и V (от датск. vrugeven - освободить). Переменная s является защищённой, значит изменять значение этой переменной могут только неделимые команды p и v.

**Основное свойство семафоров:** Одной неделимой операцией осуществляется выборка переменной s, инкремент и ее запоминание.

Если какая-либо операция не может быть выполнена над одним семафором, то неудачной считается операция над всеми семафорами.

Операция  $V(s) : s = s + 1$ , выполняется как одно неделимое действие

если  $s = 0$ , то операция V(s) может активизировать некоторый процесс блокированный на семафоре

операция  $P(s) : s = s - 1$ , процесс пытающийся войти в критическую секцию пытается декрементировать семафор

если  $s = 0$ , то декремент невозможен и процесс переводится в состояние ожидания (блокировки) до тех пор, пока другой процесс не освободит ресурс (не выйдет из своего критического участка).

Семафоры исключают бесконечное ожидание на процессоре, но платой за это является переход в режим ядра, т.е. команды определенные на семафоре являются системными вызовами, зато исключается активное ожидание на процессоре.

Процесс может создать семафор и изменять его. Удалить семафор может только процесс, создавший его, либо привилегированный процесс. Осуществляется переход в режим ядра при захвате и освобождении семафора, т.е. происходит переключение контекста.

- Если семафор может принимать 0 и 1 то называется бинарным.
- Если может принимать неотрицательные целые значения, то считающий.
- Множественные семафоры – массивы считающих семафоров (могут работать и как бинарные). Одной неделимой операцией можно изменить все или часть семафоров набора.

## Способы взаимодействий:

1. Взаимоисключени, организация монопольного доступа процесса к разделяемой переменной.

Чистое взаимоисключение реализуется в задаче читателя-писателя осуществляется монопольный доступ писателя.

2. Синхронизация, когда процесс заинтересован в действиях другого процесса.

Задача производства потребления, если потребитель работает быстрее производителя возникнет ситуация когда буфер пуст, потребитель будет ожидать когда производитель положит что-нибудь в буфер, это видно при передаче сообщений, сообщения несут информацию которая интересует процесс, для того чтобы продолжить свое выполнение.

## Производство-потребление

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
#include <sys/stat.h>
#include <stdio.h>
#include <sys/shm.h>
#include <signal.h>

#define COUNT 3
#define PRODUCER 0
#define CONSUMER 1

#define PERMS S_IRWXU | S_IRWXG | S_IRWXO

#define EMPTYCOUNT 0
#define FULLCOUNT 1
#define BIN 2

int semaphore;
int shared_memory;
char **addr_shared_memory;

// Массив структур
struct sembuf producer_grab[2] = { {EMPTYCOUNT, -1, SEM_UNDO}, {BIN, -1, SEM_UNDO} };
struct sembuf producer_free[2] = { {BIN, 1, SEM_UNDO}, {FULLCOUNT, 1, SEM_UNDO} };
struct sembuf consumer_grab[2] = { {FULLCOUNT, -1, SEM_UNDO}, {BIN, -1, SEM_UNDO} };
struct sembuf consumer_free[2] = { {BIN, 1, SEM_UNDO}, {EMPTYCOUNT, 1, SEM_UNDO} };

// Потребитель
void consumer(int semaphore, int value)
{
    while(1)
    {
        sleep(1);
        int sem_op_p = semop(semaphore, consumer_grab, 2);
        if (sem_op_p == -1)
        {
            perror("Can't semop \\n");
            exit(1);
        }

        if ((char*)(*(addr_shared_memory + sizeof(int *))) == ((char*)(addr_shared_memory) + 2 * sizeof(int *) + 5 * sizeof(int)))
            *(addr_shared_memory + sizeof(int *)) = (char*)addr_shared_memory + 2 * sizeof(int *);
    }
}
```

```

printf("Consumer%d get %d\n", value, **(addr_shared_memory + sizeof(int *)));
(*addr_shared_memory + sizeof(int *))++;

int sem_op_v = semop(semaphore, consumer_free, 2);
if (sem_op_v == -1)
{
    perror("Can't semop \n");
    exit(1);
}
}

// Производитель
void producer(int semaphore, int value)
{
    while(1)
    {
        sleep(2);
        int sem_op_p = semop(semaphore, producer_grab, 2);
        if (sem_op_p == -1)
        {
            perror("Can't semop \\n");
            exit(1);
        }

        if ((char*)(*addr_shared_memory) == ((char*)(addr_shared_memory) + 2 * sizeof
            (int *) + 5 * sizeof(int)))
            (*addr_shared_memory) = (char*)addr_shared_memory + 2 * sizeof(int *);

        *((*addr_shared_memory) = ((char*)(*addr_shared_memory) - (char*)
            addr_shared_memory) - 16;
        printf("Producer%d put %d\n", value, *((*addr_shared_memory)));
        (*addr_shared_memory)++;

        int sem_op_v = semop(semaphore, producer_free, 2);
        if (sem_op_v == -1)
        {
            perror("Can't semop \n");
            exit(1);
        }
    }
}

int main()
{
    int process;
    int consumers[3];
    int producers[3];
    // Создание семафора
    semaphore = semget(IPC_PRIVATE, 3, IPC_CREAT | PERMS);
    int se = semctl(sem, EMPTYCOUNT, SETVAL, COUNT);
    int sf = semctl(sem, FULLCOUNT, SETVAL, 0);
    int sb = semctl(sem, BIN, SETVAL, 1);
    // Объявление разделяемого сегмента
    shared_memory = shmget(IPC_PRIVATE, 2 * sizeof(int *) + 5 * sizeof(int),
        IPC_CREAT | PERMS);
    addr_shared_memory = shmat(shared_memory, 0, 0);
    *((addr_shared_memory) = (char*)addr_shared_memory + 2 * sizeof(int *));
    *((addr_shared_memory + sizeof(int *)) = (char*)addr_shared_memory + 2 * sizeof(
        int *));

    // Создание процессов
    for (int i = 0; i < COUNT; i++) {

```

```

if (-1 == (producers[i] = fork()))
{
    return 1;
}
else if (0 == producers[i])
{
    producer(semaphore, i);
    exit(0);
}

if (-1 == (consumers[i] = fork()))
{
    return 1;
}
else if (0 == consumers[i])
{
    consumer(semaphore, i);
    exit(0);
}

signal(SIGINT, catch_sigp);
int status;
wait(&status);

// Очистка памяти
shmctl(shared_memory, IPC_RMID, NULL);
semctl(semaphore, 0, IPC_RMID, 0);
return 0;
}

```

## Читатели-писатели

```

        {READERCOUNT, 1, SEM_UNDO},
        {READERWAIT, -1, SEM_UNDO} };
struct sembuf stop_read[] = { {READERCOUNT, -1, SEM_UNDO} };
struct sembuf start_write[] = { {WRITERCOUNT, 1, SEM_UNDO},
                                {READERCOUNT, 0, SEM_UNDO},
                                {ACTIVEWRITER, -1, SEM_UNDO},
                                {WRITERCOUNT, -1, SEM_UNDO}, };
struct sembuf stop_write[] = { {ACTIVEWRITER, 1, SEM_UNDO} };

// собственный обработчик сигнала ctrl-c
void catch_sigp(int sig_numb)
{
    signal(sig_numb, catch_sigp);
    shmctl(shared_memory, IPC_RMID, NULL);
    semctl(semaphore, 0, IPC_RMID, 0);
}

void StartRead()
{
    semop(semaphore, start_read, 3);
}

void StopRead()
{
    semop(semaphore, stop_read, 1);
}

void StartWrite()
{
    semop(semaphore, start_write, 4);
}

void StopWrite()
{
    semop(semaphore, stop_write, 1);
}

// Читатель
void Reader(int value)
{
    while (1)
    {
        StartRead();
        printf("Reader%d get = %d\n", value, *addr_shared_memory);
        StopRead();
        sleep(1);
    }
}

// Писатель
void Writer(int value)
{
    while (1)
    {
        StartWrite();
        (*addr_shared_memory)++;
        printf("Writer%d put = %d\n", value, *addr_shared_memory);
        StopWrite();
        sleep(2);
    }
}

int main()

```

```

{
    // Создание семафора
    semaphore = semget(IPC_PRIVATE, 4, IPC_CREAT | PERMS);
    int sr = semctl(sem, READERCOUNT, SETVAL, 0);
    int sa = semctl(sem, ACTIVEWRITER, SETVAL, 1);
    int sw = semctl(sem, WRITERCOUNT, SETVAL, 0);
    // Объявление разделяемого сегмента
    shared_memory = (int*)shmget(IPC_PRIVATE, sizeof(int), IPC_CREAT | PERMS);
    addr_shared_memory = shmat(shared_memory, 0, 0);

    // Создание процессов
    int processes[READER + WRITER];
    int parent = getpid();

    for (int i = 0; i < WRITER + READER; i++)
    {
        processes[i] = fork();
        if (getpid() != parent)
        {
            for (int j = 0; j < i; j++)
                processes[j] = -1;
            break;
        }
    }

    for (int i = 0; i < WRITER + READER; i++)
    {
        if (processes[i] != 0)
            continue;

        if (i < WRITER)
        {
            Writer(i);
            return 0;
        }
        else
        {
            Reader(i - WRITER);
            return 0;
        }
    }
}

signal(SIGINT, catch_sigp);
int status;
wait(&status);

// Очистка памяти
shmctl(shared_memory, IPC_RMID, NULL);
semctl(semaphore, 0, IPC_RMID, 0);
return 0;
}

```

## 20.2. Приоритетное планирование в ОС Windows (лабораторная работа).

В Windows реализуется **приоритетная, вытесняющая система планирования**, при которой всегда выполняется хотя бы один работоспособный (готовый) поток с самым высоким приоритетом.

Потоку разрешено работать в течение кванта времени. Но поток может и не израсходовать свой квант времени, поскольку в Windows реализуется вытесняющий планировщик: если становится готов к запуску другой поток с более высоким приоритетом, текущий выполняемый поток может быть вытеснен еще до окончания его кванта времени.

Windows использует 32 уровня приоритета от 0 до 31.

– 16 уровней реального времени;

- 16 изменяющихся уровней, уровень 0 зарезервирован.

Уровни приоритета потоков назначаются исходя из двух разных позиций: одной от Windows API и другой от ядра Windows:

- Windows API систематизирует процессы по классу приоритета, который им присваивается при создании: Реального времени — Real-time (4), Высокий — High (3), Выше обычного — Above Normal (7), Обычный — Normal (2), Ниже обычного — Below Normal (5) и Простая — Idle (1).
- Затем назначается относительный приоритет отдельных потоков внутри этих процессов. Здесь номера представляют изменение приоритета, применяемое к базовому приоритету процесса: Критичный по времени — Time-critical (15), Наивысший — Highest (2), Выше обычного — Above-normal (1), Обычный — Normal (0), Ниже обычного — Below-normal (-1), Самый низкий — Lowest (-2) и Простая — Idle (-15).

Таким образом, у каждого потока два приоритета: текущий и базовый. Решения, связанные с планированием, принимаются на основе текущего приоритета. Отображение Windows-приоритета на внутренние номерные приоритеты Windows показано в таблице.

<b>Класс приоритета/ Относительный приоритет</b>	<b>Realtime</b>	<b>High</b>	<b>Above</b>	<b>Normal</b>	<b>Below Normal</b>	<b>Idle</b>
Time Critical (+ насыщение)	31	15	15	15	15	15
Highest (+2)	26	15	12	10	8	6
Above Normal (+1)	25	14	11	9	7	5
Normal (0)	24	13	10	8	6	4
Below Normal (-1)	23	12	9	7	5	3
Lowest (-2)	22	11	8	6	4	2
Idle (- насыщение)	16	1	1	1	1	1

В то время как у процесса имеется только одно базовое значение приоритета, у каждого потока имеется два значения приоритета: текущее и базовое. Решения по планированию принимаются исходя из текущего приоритета.

Однако система при определенных обстоятельствах на короткие периоды времени повышает приоритет потоков в динамическом диапазоне (от 0 до 15). Windows никогда не регулирует приоритет потоков в диапазоне реального времени (от 16 до 31), поэтому они всегда имеют один и тот же базовый и текущий приоритет.

Исходный базовый приоритет потока наследуется из базового приоритета процесса. Процесс по умолчанию наследует свой базовый приоритет у того процесса, который его создал.

Как правило, пользовательские приложения и службы запускаются с обычным базовым приоритетом (normal), поэтому их исходный поток чаще всего выполняется с уровнем приоритета 8.

Повысить или понизить приоритет потока в динамическом диапазоне можно в любом приложении, но у вас должны быть привилегии, позволяющие повышать приоритет, использующийся при планировании для ввода значения в пределах динамического диапазона.

#### **Сценарии повышения приоритета (и их причины):**

- Повышение вследствие событий планировщика или диспетчера (сокращение задержек). Следующие сценарии имеют дело с объектом диспетчера, входящим в сигнальное состояние: В очередь потока поставлен APC-вызов, событие установлено или отправлено сигнал, системное время изменилось (таймеры должны быть перезапущены), мьютекс или семафор освобождены, изменилось состояние потока.
- Повышения связанные с завершением ожидания. Пытаются уменьшить время задержки между потоком, пробуждающимся по сигналу объекта. Поскольку событие, которого ждал поток, может дать информацию на данный момент времени. В противном случае информация может стать неактуальной.
- Повышение вследствие завершения ввода-вывода (сокращение задержек). Windows дает временное повышение приоритета при завершении определенных операций ввода-вывода, при этом потоки, ожидавшие ввода-вывода, имеют больше шансов сразу же запуститься и обработать то, чего они ожидали.

Рекомендуемые значения повышения приоритета представлены в таблице.

<b>Устройство</b>	<b>Повышение приоритета</b>
Жесткий диск, привод компакт-дисков, параллельный порт, видеоустройство	1
Сеть, почтовый слот, именованный канал, последовательный порт	2
Клавиатура, мышь	6
Звуковое устройство	8

- Повышение вследствие ввода из пользовательского интерфейса (сокращение задержек и времени отклика).

Потоки, владеющие окнами, получают при пробуждении дополнительное повышение приоритета на 2 уровня, из-за активности системы работы с окнами, например при поступлении сообщений. Приоритет для создания преимуществ, содействия интерактивным приложениям.

- Повышение вследствие слишком продолжительного ожидания ресурса исполняющей системы (предотвращение зависания).

Когда поток пытается получить ресурс исполняющей системы, который уже находится в исключительном владении другого потока, он должен войти в состояние ожидания до тех пор, пока другой поток не освободит ресурс. Для ограничения риска взаимных исключений исполняющая система выполняет это ожидание, не входя в бесконечное ожидание ресурса, а интервалами по 5 секунд.

Если по окончании этих 5 секунд ресурс все еще находится во владении, исполняющая система пытается предотвратить зависание центрального процессора путем завладения блокировкой диспетчера, повышения приоритета потока или потоков, владеющих ресурсом, до значения 14, перезапуска их квантов и выполнения еще одного ожидания.

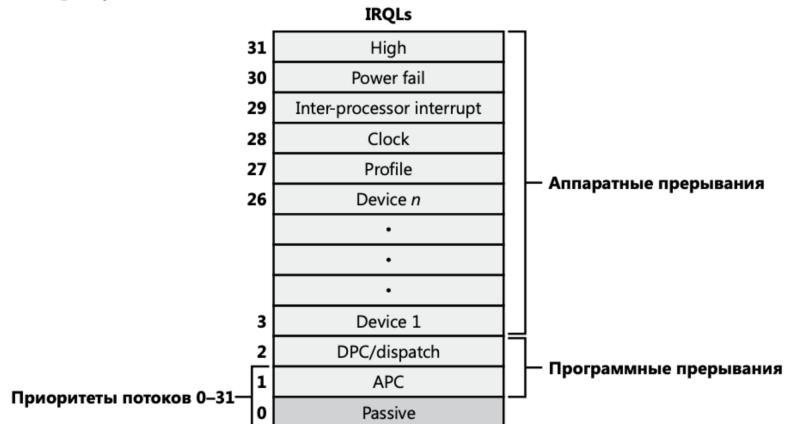
- Повышение в случае, когда готовый к запуску поток не был запущен в течение определенного времени (предотвращение зависания и смены приоритетов).

Один раз в секунду диспетчер настройки баланса сканирует очередь готовых потоков в поиске тех из них, которые находятся в состоянии ожидания (то есть не были запущены) около 4 секунд. Если такой поток будет найден, диспетчер настройки баланса повышает его приоритет до 15 единиц. Как только квант истекает, приоритет потока тут же снижается до обычного базового приоритета. Если поток не был завершен и есть готовый к запуску поток с более высоким уровнем приоритета, поток с пониженным приоритетом возвращается в очередь готовых потоков, где он опять становится подходящим для еще одного повышения приоритета, если будет оставаться в очереди следующие 4 секунды.

Прерывания обслуживаются в порядке их приоритета. При возникновении прерывания с высоким приоритетом процессор сохраняет информацию о состоянии прерванного потока и активизирует сопоставленный с данным прерыванием диспетчер ловушки. Последний повышает IRQL и вызывает процедуру обслуживания прерывания (ISR). После выполнения ISR прерванный поток возобновляется с той точки, где он был прерван.

Потоки обычно запускаются на уровне IRQL0 (который называется пассивным уровнем, потому что никакие прерывания не обрабатываются и никакие прерывания не заблокированы) или на уровне IRQL1 (APC-уровень). Код пользовательского режима всегда запускается на пассивном уровне. Поэтому никакие потоки пользовательского уровня независимо от их приоритета не могут даже заблокировать аппаратные прерывания (хотя высокоприоритетные потоки реального времени могут заблокировать выполнение важных системных потоков).

См. рисунок.



## 21 билет

**21.1 Взаимодействие параллельных процессов: монопольное использование – реализация; типы реализации взаимоисключения. Мониторы – определение, примеры: простой монитор, монитор «кольцевой буфер» и монитор «читатели-писатели». Пример реализации монитора Хоара «читатели-писатели» для ОС Windows.**

Чтобы не терять значение разделяемой переменной, необходимо обеспечить **монопольный доступ** процессов к **разделяемым переменным ресурсам** методами взаимоисключения. Только один процесс может получить доступ к данным в один момент. Те участки кода, в которых выполняется обращение к разделяемым данным называются **критическими**. Только один процесс может находиться в критическом участке.

Существуют следующие способы взаимоисключения:

1. Программный способ
2. Аппаратный способ
3. С помощью семафоров
4. С использованием мониторов

**Активное ожидание на процессоре** – ситуация, когда процесс занимает процессорное время, проверяя значение флага (занятости ресурса другим процессом). Активное ожидание на процессоре является неэффективным использованием процессорного времени.

Возможные варианты развития событий:

1. Возможно, что оба процесса пройдут цикл ожидания и попадут в свои критические секции.
2. Возможно бесконечное откладывание (зависание) – ситуация, когда разделённый ресурс снова захватывается тем же процессом.
3. Тупик (deadlock, взаимоблокировка) – ситуация, когда оба процесса установили флаги занятости и ждут. Т.е. каждый ожидает освобождения ресурса, занятого другим процессом.

### Мониторы

Проблема связанная с семафорами и аналогичными средствами которые дают ОС состоит в том, что они не структурные, программа может утратить блокировку и синхронизацию.

Чтобы устранить эту проблему была придумана концепция мониторов, которая была реализована на нескольких языках программирования.

Мониторы придуманы для структурирования средств взаимоисключения. В результате синхронизация и связь выполняется в мониторе.

Потенциальные ошибки программирования связаны с программированием самого монитора. Их легче выявлять.

Отрицательные стороны: на обращение к функциям монитора тратится больше времени.

Идея монитора заключается в создании механизма, который унифицировал бы создание параллельных процессов по данным и функциям, которые обрабатывают эти данные.

Монитор – это языковая конструкция состоящая из структуры данных и набора подпрограмм, которые могут обращаться к полям этой структуры.

Мониторы обозначаются ключевым словом monitor.

Монитор защищает свои переменные, так как доступ к полям мониторов может быть осуществлён только с помощью подпрограммы (функции) монитора.

На мониторах чаще всего определяется переменная типа условия. Используются две функции wait и signal.

### Простой монитор

Для выделение единственного ресурса n-параллельным процессам или предназначен для обеспечения взаимоисключающего доступа к одной переменной.

Сам монитор является ресурсом. Процессам, которым удалось вызвать функцию монитора, говорят что находятся в мониторе, остальные выстраиваются в очередь к монитору.

```
resource: monitor;
var
    busy: logical;
    x: conditional;

procedure require
begin
```

```

if busy then
    wait(X);
busy:=true;
end;

```

```

procedure release
begin
    busy:=false;
    signal(x);
end;

```

```

begin
    busy:=false;
end.

```

### Кольцевой буфер

Решает задачу производства-потребления. Характерно наличие процессов производителей и процессов потребителей. Производитель помещает данные в массив, а потребитель берет и они перестают существовать.

```

resource: monitor;
var
    bcircle:array[0...n-1] of <type>;
    pos:0...n;
    j:0...n-1;
    k:0...n-1;
    bufferfull, bufferempty:conditional;

```

```

procedure producer(data:<type>)
begin

```

```

    if pos = n then
        wait(bufferempty);
    bcircle[j] := data;
    pos:=pos+1;
    j:=(j+1)mod n;
    signal(bufferfull);

```

```

end;

```

```

procedure consumer(var data <type>)
begin

```

```

    if pos = 0 then
        wait(bufferfull);
    data := bcircle[j];
    pos:=pos-1;
    k:=(k+1)mod n;
    signal(bufferempty);

```

```

end;

```

```

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

```

### Монитор: читатели-писатели

2 типа процессов:

- Читатели – читают, поскольку они информацию не меняют, то они не мешают друг другу и могут читать параллельно.
- Писатель (только 1) – изменяет данные в режиме взаимоисключения (читать и писать нельзя).

Классический монитор Хоара. В нем описывается 4 функции: start\_read, stop\_read, start\_write, stop\_write.

```

resource: monitor;

```

```

var

```

```

nr:int; // читатели
wrt:logical; // писатели
can_read, can_write:conditional;

procedure start_read;
begin
    if wrt or turn(can_write) then // очередь ждущих писателей
        wait(can_read);
    nr:=nr+1;
    signal(can_read); // цепная реакция читателей, каждый активизирует следующего
    читателя
end;

procedure stop_read;
begin
    nr:=nr-1;
    if (nr == 0) then // если читателей нет
        signal(can_write); // можно писать
end;

procedure start_write;
begin
    if nr > 0 or wrt then // есть ли активный читатель или активный писатель
        wait(can_write);
    wrt:=true;
end;

procedure stop_write;
begin
    wrt:=false;
    if (turn(can_read)) then // есть ли ждущие читатели (очередь)
        signal(can_read); // активизируются ждущие читатели
    else
        signal(can_write); // активизируется читатель
end;

begin
    nr:=0;
    wrt:=false;
end;

#include <stdio.h>
#include <windows.h>
#include <iostream>
using namespace std;

#define OK 0
#define ERROR 1

#define WRITERS 3
#define READERS 5
#define ITERS 5

int val = 0;
bool activewriter = false;
int readercount = 0;

int waiting_writers = 0;
int waiting_readers = 0;

HANDLE writers[WRITERS];
HANDLE readers[READERS];

```

```

HANDLE can_write;
HANDLE can_read;

void start_read() {
    InterlockedIncrement(&waiting_readers);
    if (true == activewriter || waiting_writers > 0) {
        WaitForSingleObject(can_read, INFINITE);
    }
    InterlockedDecrement(&waiting_readers);
    InterlockedIncrement(&readercount);
    SetEvent(can_read);
}

void stop_read() {
    InterLockedDecrement(readercount);
    if (0 == readercount) {
        SetEvent(can_write);
    }
}

void start_write() {
    InterlockedIncrement(&waiting_writers);
    if (readercount > 0 || true == activewriter) {
        WaitForSingleObject(can_write, INFINITE);
    }
    InterlockedDecrement(&waiting_writers);
    activewriter = true;
}

void stop_write() {
    activewriter = false;
    ResetEvent(can_write);
    if (waiting_readers > 0) {
        SetEvent(can_read);
    } else {
        SetEvent(can_write);
    }
}

DWORD WINAPI writer(LPVOID) {
    for (int i = 0; i < ITERS; i++) {
        start_write();
        val++;
        cout << "Writer" << GetCurrentThreadId() << " write " << val << endl;
        stop_write();
        Sleep(100);
    }
    return OK;
}

DWORD WINAPI reader(LPVOID mutex) {
    for (int i = 0; i < ITERS + 7; i++) {
        start_read();
        WaitForSingleObject(mutex, INFINITE);
        cout << "Reader" << GetCurrentThreadId() << " read " << val << endl;
        ReleaseMutex(mutex);
        stop_read();
        Sleep(100);
    }
    return OK;
}

int create_mutex_threads() {

```

```

HANDLE mutex = CreateMutex(NULL, FALSE, NULL);
if (NULL == mutex) {
    cout << "Can't create mutex\n";
    return ERROR;
}
// создание писателей
for (int i = 0; i < WRITERS; i++) {
    // данный аргумент определяет, может ли создаваемый поток быть унаследован
    // и дочерним процессом
    // размер стека в байтах. Если передать 0, то будет использоваться значение
    // по - умолчанию (1 мегабайт)
    // адрес функции, которая будет выполняться потоком
    // указатель на переменную, которая будет передана в поток
    // флаги создания
    // указатель на переменную, куда будет сохранён идентификатор потока

    writers[i] = CreateThread(NULL, 0, &writer, NULL, 0, NULL);
    if (NULL == writers[i]) {
        cout << "Can't create threads\n";
        return ERROR;
    }
}
// создание читателей
for (int i = 0; i < READERS; i++) {
    readers[i] = CreateThread(NULL, 0, &reader, mutex, 0, NULL);
    if (NULL == readers[i]) {
        cout << "Can't create threads\n";
        return ERROR;
    }
}

return OK;
}

int create_events() {
// атрибут защиты
// тип сброса TRUE - ручной
// начальное состояние TRUE - сигнальное
// имя объекта

// с автосбросом
can_read = CreateEvent(NULL, FALSE, FALSE, TEXT("ReadEvent"));
if (can_read == NULL) {
    cout << "Can't create event\n";
    return ERROR;
}

// с ручным сбросом
can_write = CreateEvent(NULL, TRUE, FALSE, TEXT("WriteEvent"));
if (can_write == NULL) {
    cout << "Can't create event\n";
    return ERROR;
}
return OK;
}

int main() {

if (create_events() != OK) {
    return ERROR;
}
if (create_mutex_threads() != OK) {
    return ERROR;
}

```

```

    }

    WaitForMultipleObjects(WRITERS, writers, TRUE, INFINITE);
    WaitForMultipleObjects(READERS, readers, TRUE, INFINITE);

    return OK;
}

```

Семейство Interlocked-функций используются в случае, если разным потокам необходимо изменять одну и ту же переменную.

## 21.2 Процессы Unix: создание процесса в ОС Unix и запуск новой программы. Примеры из лабораторной работы (код).

### Создание процесса

Любой процесс создается системным вызовом fork(), который определяется в системе как потомок (child). В Unix используется иерархия процессов, которая строится в отношении предок-потомок.

В результате вызова fork создается процесс-потомок. В Unix все рассматривается как файл (файлы, директории, устройства).

В современных системах применяется, так называемая, оптимизация fork(): для процесса потомка создаются собственные карты трансляции адресов (таблицы страниц), но они ссылаются на адресное пространство процесса-предка (на страницы предка). При этом для страниц адресного пространства предка права доступа меняются на only-read и устанавливается флаг – copy-on-write. Если или предок или потомок попытаются изменить страницу, возникнет исключение по правам доступа. Выполняя это исключение супервизор обнаружит флаг copy-on-write и создаст копию страницы в адресном пространстве того процесса, который пытался ее изменить. Таким образом код процесса-предка не копируется полностью, а создаются только копии страниц, которые редактируются.

fork() возвращает 0 – для потомка, -1 – если ветвление невозможно, для родителя возвращается натуральное число (ID потомка). Любой процесс имеет предка, кроме демонов.

Для Unix очень важно понятие group. Процесс предок создает группу. Основная группа – терминальная, предок всех процессов в терминальной группе процесс id=1.

```

int main(void)
{
    pid_t childpid1;
    pid_t childpid2;
    if ((childpid1 = fork()) == -1) /* если fork завершился успешно, pid > 0 в родительском процессе */
    {
        perror("Can't fork"); /* fork потерпел неудачу (например, память или какая-либо */
        /* таблица заполнена) */
        exit(1);
    }
    if (childpid1 == 0)
    { /* здесь располагается дочерний код */
        printf("Child1 forked \n");
        printf("Child1 id = %d, parent id = %d, group id = %d \n", getpid(), getppid()
              (), getpgrp());
        getchar();
        printf("Child1 id = %d, parent id = %d, group id = %d \n", getpid(), getppid()
              (), getpgrp());
        return 0;
    }

    if ((childpid2 = fork()) == -1) /* если fork завершился успешно, pid > 0 в родительском процессе */
    {
        perror("Can't fork"); /* fork потерпел неудачу (например, память или какая-либо */
        /* таблица заполнена) */
        exit(1);
    }
    if (childpid2 == 0)
    { /* здесь располагается дочерний код */
        printf("Child2 forked \n");
    }
}

```

```

printf("Child2 id = %d, parent id = %d, group id = %d \n", getpid(), getppid
      (), getpgrp());
getchar();
printf("Child2 id = %d, parent id = %d, group id = %d \n", getpid(), getppid
      (), getpgrp());
}

if (childpid1 != 0 && childpid2 != 0)
{ /* здесь располагается родительский код */
    printf("Parent id = %d, child1 id = %d, child2 id = %d, group id = %d \n",
           getpid(), childpid1, childpid2, getpgrp());
    getchar();
    printf("Parent exited \n");
}

return 0;
}

```

### Запуск новой программы

Чаще всего нет смысла в выполнении двух одинаковых процессов и потомок сразу выполняет системный вызов exec(), параметрами которого является имя исполняемого файла и, если нужно, параметры, которые будут переданы этой программе. Говорят, что системный вызов exec() создает низкоуровневый процесс: создаются таблицы страниц для адресного пространства программы, указанной в exec(), но программа на выполнение не запускается, так как это не полноценный процесс, имеющий идентификатор и дескриптор. Системный вызов exec() создает таблицу страниц для адресного пространства программы, переданной ему в качестве параметра, а затем заменяет старый адрес новой таблицы страниц.

Бывает шести видов: execlp, execvp, execl, execv, execle, execve.

В результате системного вызова exec() адресное пространство процесса будет заменено на адресное пространство новой программы, а сам процесс будет возвращен в режим задачи с установкой указателя команд на первую выполняемую инструкцию этой программы.

```

#include <stdio.h>
#include <stdlib.h>

int main(void)
{
    pid_t childpid1;
    pid_t childpid2;
    if ((childpid1 = fork()) == -1) /* если fork завершился успешно, pid > 0 в роди
        тельском процессе */
    {
        perror("Can't fork"); /* fork потерпел неудачу (например, память или какая-ли
            бо */
        exit(1); /* таблица заполнена) */
    }
    if (childpid1 == 0)
    { /* здесь располагается дочерний код */
        printf("Child1 forked \n");
        printf("Child1 id = %d, parent id = %d, group id = %d \n", getpid(), getppid
              (), getpgrp());

        char msg[] = "exec() called from child1";

        /* обращаемся к программе echo, параметрами передаём её имя и текст для печат
           и */
        /* NULL - завершает список параметров */
        if (execlp("echo", "echo", msg, NULL) == -1) /* при успешном вызове exec
           () заменяет адресное пространство процесса */
        {
            perror("Can't exec"); /* - эти вызовы произойдут только в том случае,
               если exec() не выполнится*/
            exit(1);
        }
    }
    return 0;
}

```

```

}

if ((childpid2 = fork()) == -1) /* если fork завершился успешно, pid > 0 в роди-
тельском процессе */
{
    perror("Can't fork"); /* fork потерпел неудачу (например, память или какая-ли-
бо *)
    exit(1); /* таблица заполнена) */

}

if (childpid2 == 0)
{ /* здесь располагается дочерний код */
    printf("Child2 forked \n");
    printf("Child2 id = %d, parent id = %d, group id = %d \n", getpid(), getppid()
(), getpgrp());

    char msg[40] = "exec() called from child2";

    /* обращаемся к программе echo, параметрами передаём её имя и текст для печат-
и */

    /* NULL - завершает список параметров */
    if (execlp("echo", "echo", msg, NULL) == -1) /* при успешном вызове exec() за-
меняет адресное пространство процесса */
    {
        perror("Can't exec"); /* - эти вызовы произойдут только в том случае,
если exec() не выполнится*/
        exit(1);
    }
}

if (childpid1 != 0 && childpid2 != 0)
{ /* здесь располагается родительский код */
    printf("Parent id = %d, child1 id = %d, child2 id = %d, group id = %d \n",
getpid(), childpid1, childpid2, getpgrp());
    int status1;
    childpid1 = wait(&status1);
    printf("Child has finished: PID = %d\n", childpid1);

    int status2;
    childpid2 = wait(&status2);
    printf("Child has finished: PID = %d\n", childpid2);

    if (WIFEXITED(status1) && WIFEXITED(status2))
        printf("Children exited with code %d and %d\n", WEXITSTATUS(status1),
WEXITSTATUS(status2));
    else
        printf("Child terminated abnormally\n");
}
return 0;
}

```

## 22 билет

### 22.1 Процессы: взаимодействие параллельных процессов – монопольный доступ и взаимоисключение; аппаратная реализация взаимоисключения, спин-блокировка – реализация.

Процесс - программа в стадии выполнения. Единица декомпозиции ОС (именно ему выделяются ресурсы ОС).

Чтобы не терять значение разделяемой переменной, необходимо обеспечить **монопольный доступ** процессов к **разделяемым переменным ресурсам** методами взаимоисключения. Только один процесс может получить доступ к данным в один момент. Те участки кода, в которых выполняется обращение к разделяемым данным называются **критическими**. Только один процесс может находиться в критическом участке.

Существуют следующие способы взаимоисключения:

1. Программный способ
2. Аппаратный способ
3. С помощью семафоров
4. С использованием мониторов

**Активное ожидание на процессоре** – ситуация, когда процесс занимает процессорное время, проверяя значение флага (занятости ресурса другим процессом). Активное ожидание на процессоре является неэффективным использованием процессорного времени.

Возможные варианты развития событий:

1. Возможно, что оба процесса пройдут цикл ожидания и попадут в свои критические секции.
2. Возможно бесконечное откладывание (зависание) – ситуация, когда разделённый ресурс снова захватывается тем же процессом.
3. Тупик (deadlock, взаимоблокировка) – ситуация, когда оба процесса установили флаги занятости и ждут. Т.е. каждый ожидает освобождения ресурса, занятого другим процессом.

#### Аппаратная реализация test-and-set

Неделимая команда – прервать выполнение команды нельзя.

Команда реализует проверку и установку содержимого памяти по определенному адресу. Если переменная типа 1 байт, то можно назвать эту переменную байтом блокировки.

Соглашение: байт блои=кировки 0 – критическая секция доступна, 1 – недоступна.

Зашитить критическую секцию на однопроцессорной машине можно запретом прерывания (ф при длительном, приведут к нарушению работы машины, из-за того, что некоторые системные функции перестанут выполняться).

test-and-set в рамках одной неделимой операции:

1. Читает значение логической переменной b
2. Копирует его в a
3. Устанавливает для b значение истина

Выполнение связано с обращением к памяти, то неделимость команды достигается путём блокировки шины памяти на время выполнения этой команды.

```
program use_testandset;
    flag,c1,c2:logical;
p1:
    while(1)
    {
        c1=1;
        while(c1==1)
            test_and_set(c1,flag);
        CR1;
        flag=0;
        PR1;
    }
p2:
    while(1)
```

```

{
    c2=1;
    while(c2==1)
        test_and_set(c2,flag);
    CR2;
    flag=0;
    PR2;
}
// начальная установка
flag=0;
parbegin
    p1;p2;
parend;

```

Логическая переменная flag имеет значение 1, когда любой из процессов находится в критическом участке. Рассмотрим случай когда p1 хочет войти в свой критический участок. А процесс p2 уже находится в своём критическом участке. P1 устанавливает с1 в 1 и входит в цикл проверки переменной flag. Поскольку p2 находится в своём критическом участке то flag равен 1. Команда обнаруживает этот факт и устанавливает с1 в 1. В результате p1 находится в своём цикле активного ожидания пока p2 не выйдет из своего критического участка.

Отмечается, что этот способ реализации не исключает бесконечного откладывания, но его вероятность считается маленькой. Когда процесс выходит из своего критического участка и устанавливает флаг false., то скорее всего другой процесс сможет перехватить инициативу и установить переименованную флаг в true.

Использование команды test-and-set в цикле называется циклической блокировкой (simple mutex).

Использование команды в spin-блокировке записывается следующим образом:

```

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

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

```

Команда test-and-set связана с блокировкой шины памяти. Таким образом длительный цикл команды может привести к занятию шины памяти один потоком. И следовательно к существенному снижению производительности системы. Эта проблема решается путём использования двух вложенных циклов.

```

void spin_lock(spin_lock_t * c)
{
    while(test_and_set(*c) != 0)
        while(*c != 0); // вложенный цикл без захвата шины данных
}

```

Spin-блокировки активно используются в ядре системы. Часть кода ядра не может быть заблокирована и вход в критические секции осуществляется с использованием spin блокировок.

spin\_lock – микрокоманды ядра.

## 22.2 Процессы: бесконечное откладывание, зависание, тупиковая ситуация – анализ на примере задачи об обедающих философах и примеры аналогичных ситуаций в ОС. Множественные семафоры в Linux: системные вызовы и поддержка в ОС Linux; примеры из лабораторных работ.

**Бесконечное откладывание (зависание)** – ситуация, когда разделённый ресурс снова захватывается тем же процессом. **Тупик (deadlock, взаимоблокировка)** – ситуация, когда оба процесса установили флаги занятости и ждут, то есть каждый ожидает освобождения ресурса, занятого другим процессом

### Множественные семафоры

Современные ос поддерживают именно множественные семафоры, представляются как массивы считающих семафоров. И Windows, и Linux ,и Unix поддерживают. Рассмотрим на примере задачи обедающие философы.

На круглом столе стоит 5 тарелок, 5 приборов. Напротив каждой тарелки сидит философ. Философу нужно 2 прибора.

В связи с этим, различаются три способа действия философов:

1. Каждый пытается взять две вилки, если ему это удаётся, то он может есть.

- Философ пытается взять правую вилку, после чего, если удалось взять правую, удерживая ее, пытается взять левую.
- Философ пытается взять правую, если не может взять левую, то кладёт правую на место.

Это три негативные ситуации в системе:

- Бесконечное откладывание. Сколько философов умрет голодной смертью? Голодание - бесконечное откладывание
- Все философы одновременно взял правую вилку, и никто не может взять левую, они блокируют друг друга.
- Захват и освобождение одних и тех же ресурсов.

Самым важным набором семафоров: одной неделимой командой можно изменить все или часть семафоров набора. Если в программе используется большое число семафоров, которые освобождаются и объявляются в разных функциях, очень сложно отладить программу и найти когда она входит в тупик.

Win, Unix, современный Linux поддерживают наборы считающих семафоров, они обладают важнейшим свойством: одной неделимой операцией можно изменить все или часть семафоров набора.

Если какая-либо операция не может быть выполнена над одним семафором, то неудачной считается операция над всеми семафорами.

Семафоры поддерживаются в ядре таблицей семафоров. Каждая строка описывает 1 набор из всех созданных в системе наборов семафоров.

- Имя – целое число, присваивается процессам создавшим набор семафоров. Другие процессы могут по имени открыть.
- UID – идентификатор создателя набора семафора и его группы. Если UID процесса совпадает с UID создателя, то может удалять набор и изменять его управляющие параметры.
- Права доступа
- Количество семафоров в наборе
- Время изменения одного или нескольких значений семафора последним процессом
- Время изменения управляющих параметров
- Указатель на массив семафоров

О каждом семафоре известно:

- Значение семафора
- ID процесса, который оперировал с семафором в последний раз
- Число процессов заблокированных на семафоре

В отличие от семафора Дейкстры, на семафоре в Unix определено 3 операции:

- Захват, декrement
- Освобождение, инкремент
- Проверка семафора на 0. Процесс выполняющий такую проверку переходит в состояние ожидания освобождения ресурса.

В система определены структуры для работы с семафорами: <sys/sem.h>

```
struct sembuf
{
    n_short sem_num; // индекс
    short sem_op; // операция
    short sem_flg; // флаги
}
```

3 операции:

- sem\_op < 0. Захват, декrement. Если не может то блокируется в операции.
- sem\_op > 0. Освобождение, инкремент

- sem\_op = 0. Проверка семафора на 0.

На семафорах определены флаги:

IPC\_NOWAIT – информирует ядро о нежелании процесса переходить в состояние ожидания.

Это позволяет избежать очереди к семафору, в случае аварийного завершения или kill.

В силу того, что kill невозможно прекратить, убиваемый процесс не может осуществить освобождение семафора.

SEM\_UNDO – указывает ядру, что необходимо отслеживать значение семафора, в результате завершения процесса, вызвавшего sem\_op, ядро отменяет все изменения, чтобы процессы не были блокированы навсегда.

1. Взаимоисключения, организация монопольного доступа процесса к разделяемой переменной.

Чистое взаимоисключение реализуется в задаче читателя-писателя осуществляется монопольный доступ писателя.

2. Синхронизация, когда процесс заинтересован в действиях другого процесса.

Задача производства потребления, если потребитель работает быстрее производителя возникнет ситуация когда буфер пуст, потребитель будет ожидать когда производитель положит что-нибудь в буфер, это видно при передаче сообщений, сообщения несут информацию которая интересует процесс, для того чтобы продолжить свое выполнение.

**23** билет

## 24 билет

### 24.1 Процессы: взаимодействие параллельных процессов – монопольный доступ и взаимоисключение; алгоритм Лампорта «Булочная» и «Логические часы» Лампорта.

Процесс - программа в стадии выполнения. Единица декомпозиции ОС (именно ему выделяются ресурсы ОС).

Чтобы не терять значение разделяемой переменной, необходимо обеспечить **монопольный доступ** процессов к **разделяемым переменным ресурсам** методами взаимоисключения. Только один процесс может получить доступ к данным в один момент. Те участки кода, в которых выполняется обращение к разделяемым данным называются **критическими**. Только один процесс может находиться в критическом участке.

Существуют следующие способы взаимоисключения:

1. Программный способ
2. Аппаратный способ
3. С помощью семафоров
4. С использованием мониторов

**Активное ожидание на процессоре** – ситуация, когда процесс занимает процессорное время, проверяя значение флага (занятости ресурса другим процессом). Активное ожидание на процессоре является неэффективным использованием процессорного времени.

Возможные варианты развития событий:

1. Возможно, что оба процесса пройдут цикл ожидания и попадут в свои критические секции.
2. Возможно бесконечное откладывание (зависание) – ситуация, когда разделённый ресурс снова захватывается тем же процессом.
3. Тупик (deadlock, взаимоблокировка) – ситуация, когда оба процесса установили флаги занятости и ждут. Т.е. каждый ожидает освобождения ресурса, занятого другим процессом.

#### Алгоритм bakery(булочная) – Лампорта

Каждому клиенту выдается листок с номером, новому – с большим номером. Когда продавец освобождается, то обслуживает клиента с меньшим номером.

Бывает, что одновременно приходят 2 клиента, в этом случае им выдаются одинаковые номера, но при обслуживании, обслуживает клиента с меньшим номером паспорта, к примеру.

```
1 var choosing: shared array [0..n-1] of boolean;
2       number: shared array [0..n-1] of integer;
3 repeat

4   choosing[i]:=true;
5   number[i]:=max(number[0],...,number[n-1])+1;
6   choosing[i]:=false;

7   for j:=0 to n-1 do
8     begin
9       while choosing[j] do /*nothing*/;
10      while(number[j] <> 0) and
11          (number[j],j)<(number[i],i) do
12        /*nothing*/;
13    end;
14  /*critical section*/
15  number[i]:=0;
16  /*remainder section*/

17 until false;
```

1 и 2 определяет массивы флагов и целых чисел.

choosing позволяет нам определять критическую секцию choosing[i]=true, если процесс p[i] выбирает номер. Номер, которые использует p[i] для входа в критический участок, это number[i]. number[i]=0, если p[i] не пытается войти в свой критический участок.

4,5,6 строки показывают, что процесс выбирает номер и устанавливает свой флаг `choosing[i]` в true. После этого он пытается получить уникальный номер, максимальный среди выданных +1, если это возможно. Получив номер, он сбрасывает свой флаг (строчка 6).

7-12 определяет какой процесс входит в критический участок. Процесс `p[i]` ждет пока имеется процесс, у которого меньший номер, для того чтобы войти в критический участок. Если 2 процесса имеют одинаковые номера, то выбирается процесс с меньшим идентификатором.

Запись 10 – лексикографический порядок  $((a,b) < (c,d)$ , если  $(a < c)$  или  $(a = c, b < d)$  )

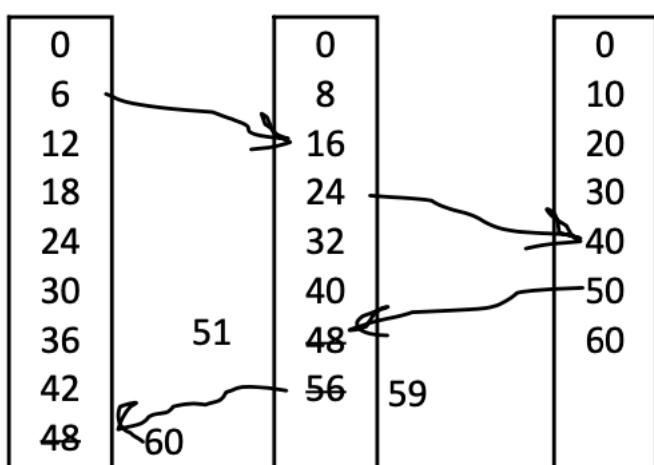
Также необходимо обратить внимание на то, когда 2 процесса выбирают номер, то 2 процесс будет ждать, пока 1 не сделает (строчка 8).

14 строка – процесс `p[i]` больше не заинтересован во входе в критический участок.

#### Алгоритм синхронизации логических часов (алг. Лампорта)

Проблема: локальные часы компьютера, имеющие ограниченную точность. Сообщение получено раньше, чем было отправлено – НЕ МОЖЕТ!

1. Процесс, отправивший сообщение, отправляет и время отправки по локальным часам.
2. Получивший процесс приравнивает свое время и время отправки + 1, при условии, что оно больше времени получения.



## 24.2 Процессы: бесконечное откладывание, зависание, тупиковая ситуация – анализ на примере задачи об обедающих философах и примеры аналогичных ситуаций в ОС. Множественные семафоры в Linux: системные вызовы и поддержка в системе; пример из лабораторной работы «производство-потребление».

**Бесконечное откладывание (зависание)** – ситуация, когда разделённый ресурс снова захватывается тем же процессом. **Тупик (deadlock, взаимоблокировка)** – ситуация, когда оба процесса установили флаги занятости и ждут, то есть каждый ожидает освобождения ресурса, занятого другим процессом

#### Множественные семафоры

Современные ОС поддерживают именно множественные семафоры, представляются как массивы считающих семафоров. И Windows, и Linux, и Unix поддерживают. Рассмотрим на примере задачи обедающие философы.

На круглом столе стоит 5 тарелок, 5 приборов. Напротив каждой тарелки сидит философ. Философу нужно 2 прибора.

В связи с этим, различаются три способа действия философов:

1. Каждый пытается взять две вилки, если ему это удаётся, то он может есть.
2. Философ пытается взять правую вилку, после чего, если удалось взять правую, удерживая ее, пытается взять левую.
3. Философ пытается взять правую, если не может взять левую, то кладёт правую на место.

Это три негативные ситуации в системе:

1. Бесконечное откладывание. Сколько философов умрет голодной смертью? Голодание – бесконечное откладывание
2. Все философы одновременно взяли правую вилку, и никто не может взять левую, они блокируют друг друга.

### 3. Захват и освобождение одних и тех же ресурсов.

Самым важным набором семафоров: одной неделимой командой можно изменить все или часть семафоров набора. Если в программе используется большое число семафоров, которые освобождаются и объявляются в разных функциях, очень сложно отладить программу и найти когда она входит в тупик.

Win, Unix, современный Linux поддерживают наборы считающих семафоров, они обладают важнейшим свойством: одной неделимой операцией можно изменить все или часть семафоров набора.

Если какая-либо операция не может быть выполнена над одним семафором, то неудачной считается операция над всеми семафорами.

Семафоры поддерживаются в ядре таблицей семафоров. Каждая строка описывает 1 набор из всех созданных в системе наборов семафоров.

1. Имя – целое число, присваивается процессам создавшим набор семафоров. Другие процессы могут по имени открыть.
2. UID – идентификатор создателя набора семафора и его группы. Если UID процесса совпадает с UID создателя, то может удалять набор и изменять его управляющие параметры.
3. Права доступа
4. Количество семафоров в наборе
5. Время изменения одного или нескольких значений семафора последним процессом
6. Время изменения управляющих параметров
7. Указатель на массив семафоров

О каждом семафоре известно:

1. Значение семафора
2. ID процесса, который оперировал с семафором в последний раз
3. Число процессов заблокированных на семафоре

В отличие от семафора Дейкстры, на семафоре в Unix определено 3 операции:

- Захват, декремент
- Освобождение, инкремент
- Проверка семафора на 0. Процесс выполняющий такую проверку переходит в состояние ожидания освобождения ресурса.

В системе определены структуры для работы с семафорами: <sys/sem.h>

```
struct sembuf
{
    n_short sem_num; // индекс
    short sem_op; // операция
    short sem_flg; // флаги
}
```

3 операции:

- sem\_op < 0. Захват, декремент. Если не может то блокируется в операции.
- sem\_op > 0. Освобождение, инкремент
- sem\_op = 0. Проверка семафора на 0.

На семафорах определены флаги:

IPC\_NOWAIT – информирует ядро о нежелании процесса переходить в состояние ожидания.

Это позволяет избежать очереди к семафору, в случае аварийного завершения или kill.

В силу того, что kill невозможно прекратить, убиваемый процесс не может осуществить освобождение семафора.

SEM\_UNDO – указывает ядру, что необходимо отслеживать значение семафора, в результате завершения процесса, вызвавшего sem\_op, ядро отменяет все изменения, чтобы процессы не были заблокированы навсегда.

### Производство-потребление

Задача производства потребления, если потребитель работает быстрее производителя возникнет ситуация когда буфер пуст, потребитель будет ожидать когда производитель положит что-нибудь в буфер, это видно при передаче сообщений, сообщения несут информацию которая интересует процесс, для того чтобы продолжить свое выполнение.

```

#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
#include <sys/stat.h>
#include <stdio.h>
#include <sys/shm.h>
#include <signal.h>

#define COUNT 3
#define PRODUCER 0
#define CONSUMER 1

#define PERMS S_IRWXU | S_IRWXG | S_IRWXO

#define EMPTYCOUNT 0
#define FULLCOUNT 1
#define BIN 2

int semaphore;
int shared_memory;
char **addr_shared_memory;

// Массив структур
struct sembuf producer_grab[2] = { {EMPTYCOUNT, -1, SEM_UNDO}, {BIN, -1, SEM_UNDO} };
struct sembuf producer_free[2] = { {BIN, 1, SEM_UNDO}, {FULLCOUNT, 1, SEM_UNDO} };
struct sembuf consumer_grab[2] = { {FULLCOUNT, -1, SEM_UNDO}, {BIN, -1, SEM_UNDO} };
struct sembuf consumer_free[2] = { {BIN, 1, SEM_UNDO}, {EMPTYCOUNT, 1, SEM_UNDO} };

// Потребитель
void consumer(int semaphore, int value)
{
    while(1)
    {
        sleep(1);
        int sem_op_p = semop(semaphore, consumer_grab, 2);
        if (sem_op_p == -1)
        {
            perror("Can't semop \\n");
            exit(1);
        }

        if (((char*)(*(addr_shared_memory + sizeof(int *))) == ((char*)(addr_shared_memory) + 2 * sizeof(int *) + 5 * sizeof(int)))
            *(addr_shared_memory + sizeof(int *)) = (char*)addr_shared_memory + 2 * sizeof(int *);

        printf("Consumer%d get %d\\n", value, **(addr_shared_memory + sizeof(int *)));
        (*(addr_shared_memory + sizeof(int *)))++;

        int sem_op_v = semop(semaphore, consumer_free, 2);
        if (sem_op_v == -1)
        {
            perror("Can't semop \\n");
            exit(1);
        }
    }
}

// Производитель

```

```

void producer(int semaphore, int value)
{
    while(1)
    {
        sleep(2);
        int sem_op_p = semop(semaphore, producer_grab, 2);
        if (sem_op_p == -1)
        {
            perror("Can't semop \\n");
            exit(1);
        }

        if ((char*)(*addr_shared_memory) == ((char*)(addr_shared_memory) + 2 * sizeof
            (int *) + 5 * sizeof(int)))
            (*addr_shared_memory) = (char*)addr_shared_memory + 2 * sizeof(int *);

        *((*addr_shared_memory) = ((char*)(*addr_shared_memory) - (char*)
            addr_shared_memory) - 16;
        printf("Producer%d put %d\\n", value, *((*addr_shared_memory)));
        (*addr_shared_memory)++;

        int sem_op_v = semop(semaphore, producer_free, 2);
        if (sem_op_v == -1)
        {
            perror("Can't semop \\n");
            exit(1);
        }
    }
}

int main()
{
    int process;
    int consumers[3];
    int producers[3];
    // Создание семафора
    semaphore = semget(IPC_PRIVATE, 3, IPC_CREAT | PERMS);
    int se = semctl(sem, EMPTYCOUNT, SETVAL, COUNT);
    int sf = semctl(sem, FULLCOUNT, SETVAL, 0);
    int sb = semctl(sem, BIN, SETVAL, 1);
    // Объявление разделяемого сегмента
    shared_memory = shmget(IPC_PRIVATE, 2 * sizeof(int *) + 5 * sizeof(int),
        IPC_CREAT | PERMS);
    addr_shared_memory = shmat(shared_memory, 0, 0);
    *((addr_shared_memory) = (char*)addr_shared_memory + 2 * sizeof(int *));
    *((addr_shared_memory + sizeof(int *)) = (char*)addr_shared_memory + 2 * sizeof(
        int *));

    // Создание процессов
    for (int i = 0; i < COUNT; i++) {
        if (-1 == (producers[i] = fork()))
        {
            return 1;
        }
        else if (0 == producers[i])
        {
            producer(semaphore, i);
            exit(0);
        }

        if (-1 == (consumers[i] = fork()))
        {
            return 1;
        }
    }
}

```

```
}

else if (0 == consumers[i])
{
    consumer(semaphore, i);
    exit(0);
}

int status;
wait(&status);

// Очистка памяти
shmctl(shared_memory, IPC_RMID, NULL);
semctl(semaphore, 0, IPC_RMID, 0);
return 0;
}
```

**25** билет

## 26 Дополнительные вопросы

### 26.1 XMS

XMS (extended memory specification (дополнительная)) - правила работы с различными областями физической памяти.

### 26.2 Методы организации ввода-вывода: программируемый, с прерываниями, прямой доступ к памяти.

### 26.3 Кэши TLB и данных.

TLB (translation look-aside buffer, буфер быстрого преобразования адреса, буфер предварительной трансляции) – таблица в блоке управл. памятью, отвечающая за преобразование виртуальных адресов в физические (в чипе).

TLB представляет собой четырехканальный ассоциативный по множеству буфер – кэш. В КЭШе TLB хранятся адреса страниц, к которых были последние обращения.

### 26.4 ОС с монолит. ядром. Переключение в режим ядра. Система прерываний. Точные и неточные прерывания.

**Монолитное ядро** – это программа имеющая модульную структуру, то есть состоящая из подпрограмм. Системы Windows, Unix, Linux имеют монолитные ядра. Unix имеет минимизированное ядро.

Единственный способ изменить его конфигурацию, например, добавить или удалить компоненты в ядро, можно только путём его перекомпиляции. Взаимодействие приложений с ядром выполняется с помощью системных вызовов. Все функции в иерархической машине и структуре Unix bsd выполняются монолитным ядром, то есть монолитное ядро включает в себя все.

Определяется состав системы прерываний. Системные вызовы (API) для того, чтобы приложения могли запрашивать функции системы. Ос переходит в режим ядра. Исключения делятся на исправимые и неисправимые.

### 26.5 Спецификация XM ( XMS ): Conventional, HMA, UMA, EMA.

**XMS** (eXtended Memory Specification, спецификация расширенной памяти) – спецификация Microsoft на расширенную память (XMS 2.0), позволяющая DOS-программам с помощью диспетчера расширенной памяти (XMM) использовать расширенную память ПК на процессорах 80286 и более новых. (добавленная, продленная) XMS оговаривает все вопросы, связанные с дополнительной памятью (сверх 1 Мб).

**EMS** (Expanded Memory Specification, спецификация отображаемой памяти) – стандарт, разработанный в 1985 г. фирмами Lotus, Intel и Microsoft для доступа из DOS к областям памяти выше 1 Мбайт в системах на базе процессоров 80386 и более поздних. (расширенная, растягиваемая)

- 0-640 Кб: основная память (conventional) – память, доступная DOS и программам реального режима. Стандартная память является самой дефицитной в PC, когда речь идет о работе в среде ОС типа MS-DOS. На ее небольшой объем (типичное значение 640 Кбайт) претендуют и BIOS, и ОС реального режима, а остатки отдаются прикладному ПО. Стандартная память используется для векторов прерываний, области переменных BIOS; области DOS; предоставляется пользователю (до 638 Кбайт).
- 640-1024 Кб: UMA (upper memory area) – обл. верхней памяти – зарезервирована для системных нужд. Размещаются обл. буферной памяти адаптеров (пр. – видеопамять) и постоянная память (BIOS с расширениями).
- С 1024 Кб –... (заш.) XMA (extended memory area) – непосредственно доступна только в защищенном режиме для компьютеров с процессорами 286 и выше.
- 1024-1088 Кб (Реал): HMA (high memory area) – верхняя область памяти (1-й сегмент размером 64 Кбайт, расположенный выше мегабайтной отметки памяти PC с операционной системой MS-DOS). Единственная область расширенной памяти, доступная 286+ в реальном режиме при открытом вентиле Gate A20.

**XMS** – программная спецификация использования дополнительной памяти DOS-программами для компьютеров на процессорах 286 и выше. Позволяет программе получить в распоряжение одну или несколько областей дополнительной памяти, а также использовать область HMA. Распределением областей ведает диспетчер расширенной памяти - драйвер HIMEM.SYS. Диспетчер позволяет захватить или освободить область HMA (65 520 байт, начиная с 100000h), а также управлять вентилем линии адреса A20. Функции XMS позволяют программе:

- определить размер максимального доступного блока памяти;

- захватить или освободить блок памяти;
- копировать данные из одного блока в другой, причем участники копирования могут быть блоками как стандартной, так и дополнительной памяти в любых сочетаниях;
- запереть блок памяти (запретить копирование) и отпереть его;
- изменить размер выделенного блока.

Спецификации EMS и XMS отличаются по принципу действия: в EMS для доступа к дополнительной памяти выполняется отображение (страничная переадресация) памяти, а в XMS – копирование блоков данных.

## **26.6 Управление памятью: выделение памяти разделами фиксированного размера, выделение памяти разделами переменного размера, стратегии выделения памяти, фрагментация памяти.**

### **Выделение памяти разделами фиксированного размера**

В первых мультипрограммных ос. При запуске системы определялись разделы определённого размера который определялся из собираемой статистики. Статические разделы не менялись.

### **Выделение памяти разделами переменного размера**

В этом подходе никакие статические разделы не определялись. В начальный момент системы программы загружались одна за другой.

В раздел мы можем положить только программу меньшего размера. В результате постоянной выгрузки и загрузки программ в памяти возникает большое кол-во разделов очень маленького размера куда нельзя положить программу, это называется **фрагментация памяти**.

Второе важнейшее это выбор стратегии выбора раздела для загрузки программы:

1. Первый подходящий
2. Самый тесный раздел
3. Самый широкий раздел

Перезагрузка компьютера справляется с дефрагментацией оперативной памяти. Цена – потеря проделанной работы.

## **26.7 Тупики: Обнаружение тупиков для повторно используемых ресурсов методом редукции графа, способы представления графа и методы восстановление работоспособности системы.**

**Повторно-используемые ресурсы** – количество в системе постоянно и при использовании они не изменяются (или редко): аппаратура (ОП, ЦП), реenterабельные коды, системные таблицы (изменения в них могут вноситься только супервизором), процедуры ОС (так как они являются реenterабельными).

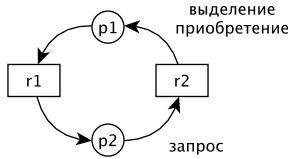
**Потребляемые ресурсы** – количество в ОС переменно и произвольно: сообщения. Процесс может создать любое количество сообщений. Процесс получения сообщения заканчивается его уничтожением.

**Тупик** – ситуация, возникающая в результате монопольного использования разделяемых ресурсов, когда процесс, владея ресурсом, запрашивает другой ресурс, занятый непосредственно или через цепочку запросов другими процессами, ожидающими освобождение ресурса, занятого 1-м процессом.

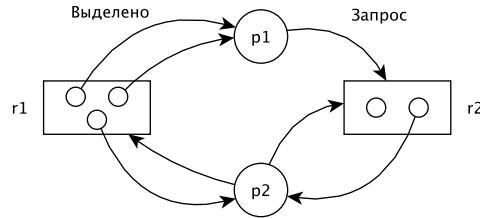
### **Методы борьбы с тупиками**

1. Недопущение тупиков (в системе создаются такие условия, когда тупик в принципе невозможен)
2. Обход тупиков тупик (возможен, но выполняются действия, которые предотвращают возникновение тупика)
3. Обнаружение тупиков (тупики возможны и они возникают, если они возникли, то из надо обнаружить, чтобы восстановить работоспособность системы)

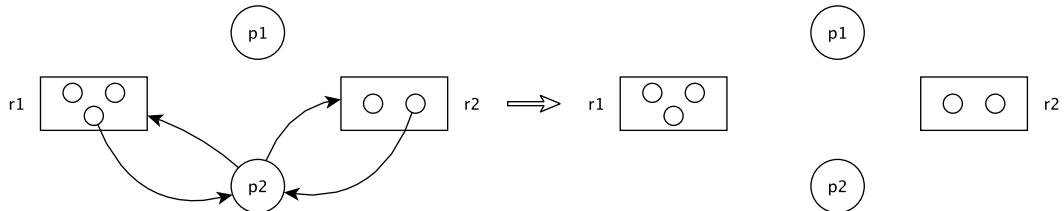
Обнаружение тупиков происходит с помощью графовой модели. Система может быть написана двудольным направленным графом. Два непересекающихся множества вершин (вершины процессов и вершины ресурсов) При этом вершины графа соединяются дугами причём никакая дуга не соединяет вершины одного подмножества. Дуга направленная из вершины принадлежащей множеству ресурсов вершине принадлежащей подмножеству процессов называется **выделением или приобретением ресурсов**. Дуга направленная из вершины относящейся к процессам к ресурсам называется **запросом**.



Обнаружение тупиков по графовой модели выполняется методом редукции графа. Этот метод основан на том эгоистическом предположении, что незаблокированный в тупике процесс может запрашивать приобретать любые нужные ему ресурсы, а затем освобождать их. Освободившиеся ресурсы могут быть выделены или распределены другим процессам, которые их ожидают.



Процессу 1 выделены ресурсы 1 и он запрашивает ещё ресурсы 2. Процессу 2 выделено 1 и 2 и он запрашивает ещё одну 1 и одну 2. Данный граф может быть сокращён по вершине  $p_1$ , поскольку он запрашивает ресурс 1 а он у системы есть. И в результате он освободит занимаемые им единицы ресурсов. Теперь и процесс 2 может запросить ресурсы и завершиться.



Данный граф является полностью сокращаемым. Значит система не находилась в тупике. Для представления графа используются матрица распределений и матрица запросов.

$$S \xrightarrow{i} T$$

Кол-во единиц  $j$ -го ресурса выданное  $i$ -му процессу.

$$|b_{ij}| - \text{матрица выделений}$$

$$|a_{ij}| - \text{матрица запросов}$$

Вектор свободных единиц ресурсов.

$$\mathcal{F} = (\dots, f_i, \dots)$$

Для повторно используемых ресурсов в системе хранится информация о типах.

Кол-во выделенных + кол-во свободных = кол-во единиц данного ресурса в системе.

### Восстановление системы

Существует два самых общих подхода.

- Прекращение процессов при этом завершаются процессы попавшие в тупик одни за другим. Тогда будут доступны ресурсы для других процессов попавших в тупик.
- Перехват ресурсов у процессов которые в тупик не попали. Отобрать ресурс можно только в результате отката. Процесс возвращается в состояние до запроса ресурса, но такой откат должен быть программно описан.

Критерии завершения процесса

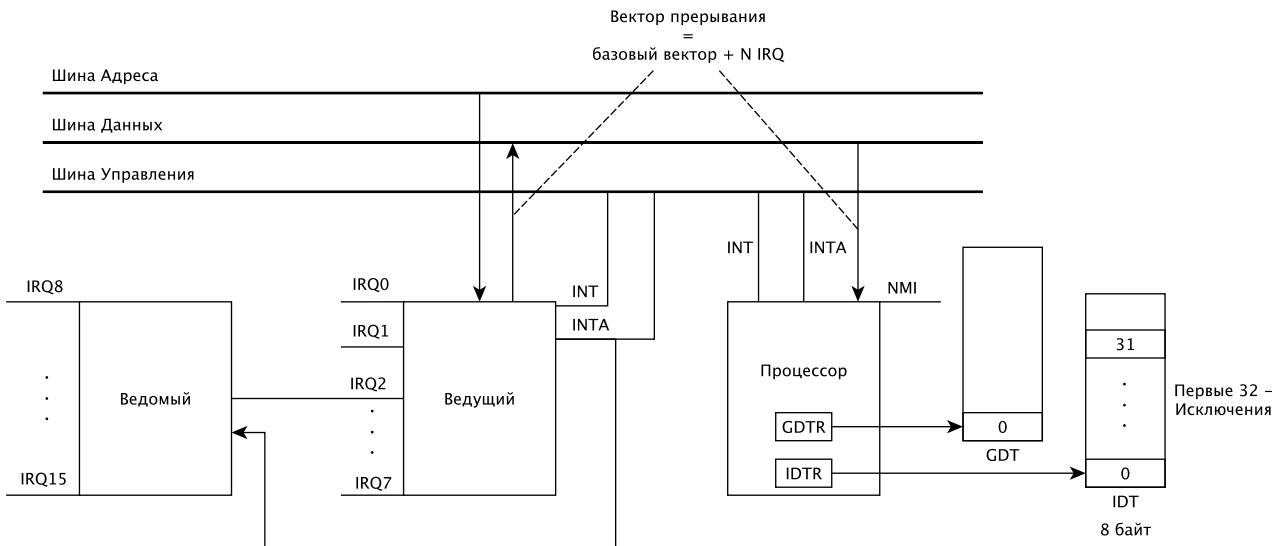
- Приоритет

2. Цена повторного запуска процесса

3. Внешняя цена

## 26.8 Последовательность операций при выполнении аппаратного прерывания. Прерывания точные и неточные

Любой обработчик аппаратного прерывания входит в драйвер и является его точкой входа.



Когда устройство заканчивает свою работу, оно инициирует прерывание (если они разрешены ОС). Для этого устройство посыпает сигнал на выделенную этому устройству специальную линию шины. Этот сигнал распознается контроллером прерываний.

При отсутствии других необработанных запросов прерывания контроллер обрабатывает его сразу.

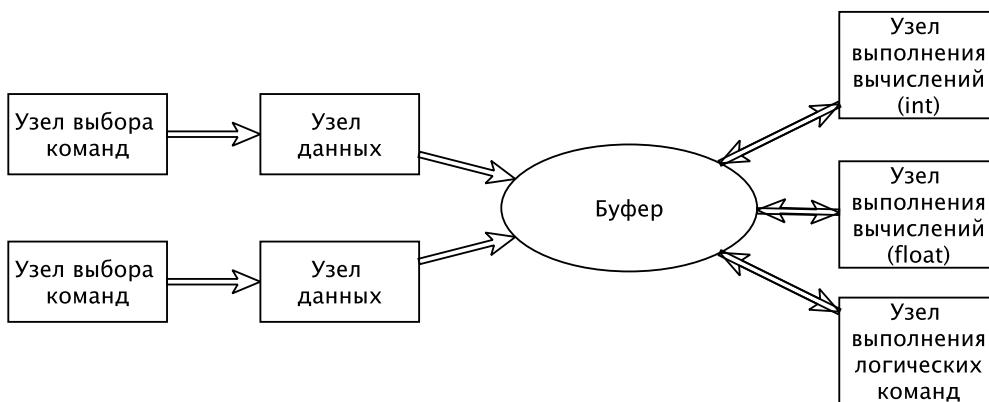
В конце цикла выполнение каждой команды процессор проверяет наличие сигнала прерывания и переходит в обработчик, прежде посыпая через INT A сигнал.

Получив INT A контроллер формирует вектор прерывания, который по шине данных поступает в процессор. Вектор используется для адресации обработчика.

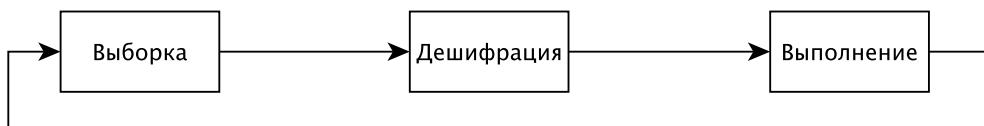
У контроллера прерываний есть порт, значит он адресуется.

### Точные и неточные прерывания

Современные процессоры являются суперскалярными.



Этапы выполнения программ:



Замещение счетчика команд не отражает истинное значение, между выполненными и еще не выполненными.

При возврате из прерывания невозможно просто начать выполнения с адреса, находящегося в счетчике команд.

В современных системах вводится понятие точного прерывания.

Точное прерывание – прерывание, оставляющее машину в строго определенном состоянии.

Точное прерывание условие:

- Счетчик команд – указывает на команду, до которой все команды выполнены полностью.
- Ни одна команда, на которую указывает счетчик не выполнена.
- Состояние команды, на которую указывается счетчик известно.

Следует, что все изменения связанные с этими командами должны быть отменены.

В конце цикла выполнения каждой команды процессор проверяет наличие прерывания Int на своей ножке.

При исключениях (при страничном прерывании) счетчик команд будет содержать адрес команды, на котором возникло страничное прерывание.

Для того, чтобы обработать прерывания как точное, машины с суперскалярными процессорами при каждом прерывании должны сохранять большие объемы данных, аппаратные прерывания выполняются медленно.

В итоге суперскалярные процессоры становятся непригодными для практических целей, из-за длительных прерываний.

Но не все прерывания необходимо делать точными: например, обработка деления на 0, так как процесс завершается.

Суперскалярные процессоры Pentium 0 поддерживают точные прерывания, ценой за точные прерывания является сложная внутрипроцессорная логика.

## 26.9 EMS

EMS (Expanded Memory Specification, спецификация отображаемой памяти, растянутая) – стандарт, разработанный в 1985 г. фирмами Lotus, Intel и Microsoft для доступа из DOS к областям памяти выше 1 Мбайт в системах на базе процессоров 80386 и более поздних.

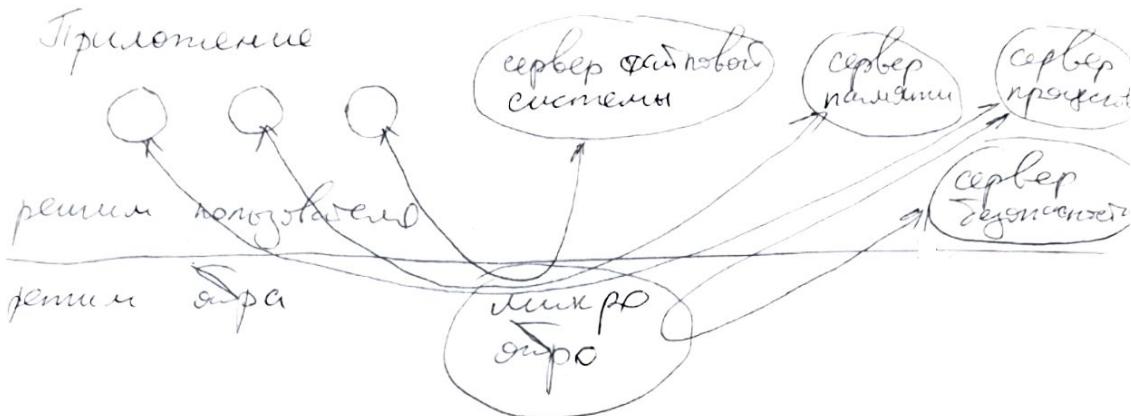
## 26.10 Классификация структур ядер ОС. Особенности ОС с микроядром. Модель клиент-сервер. Три состояния процесса при передаче сообщений. Достоинства и недостатки микро-ядерной архитектуры.

Существует 2 типа ядер: монолитные и микроядра.

Монолитное ядро – программа, имеющая модульную структуру, состоящая из подпрограмм. Windows, Unix, Linux – монолитные ядра. Linux – минимизированное (вынесен графический интерфейс).

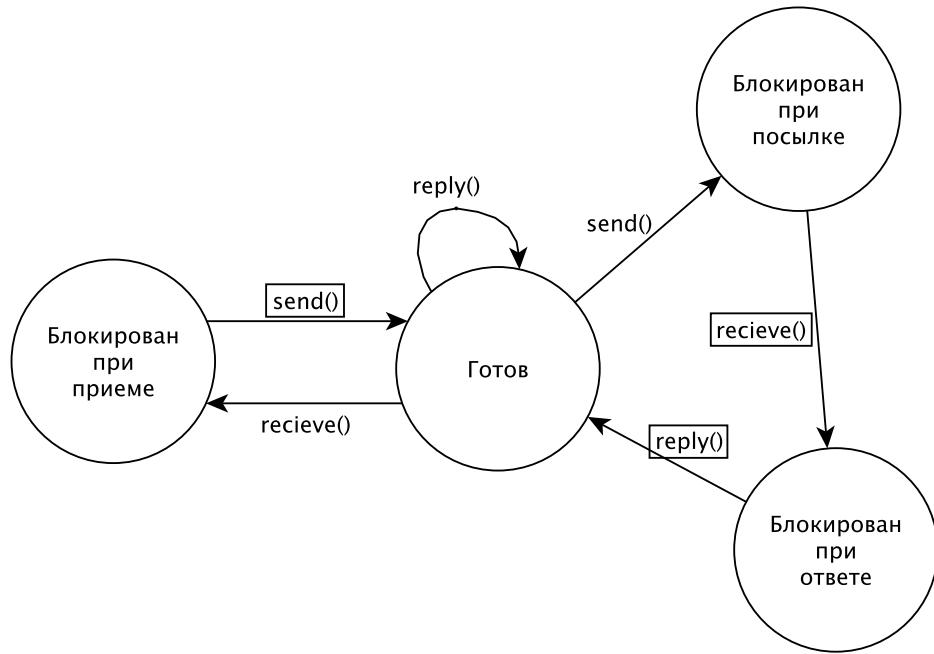
Микроядерная архитектура: все компоненты самостоятельные программы, которые выполняются в собственных адресных пространствах.

Взаимодействие между компонентами ОС выполняется с помощью посылки и приема сообщений, причем обеспечивается модулем – микроядро.

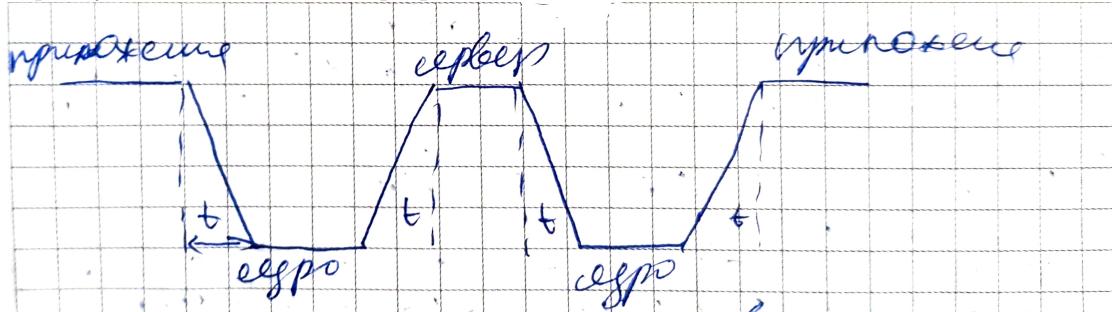


Микроядерная архитектура работает по модели клиент-сервер.

Взаимодействие в такой системе обеспечивается с помощью сообщений:



В прямоугольниках команды сервера, без – клиент.



До сих пор не существует ОС с микроядром общего назначения.

Mach – первая ОС с микроядром.

Ядро управляет 5 абстракциями:

- Процессы
- Потоки
- Объекты памяти – объединение нескольких страниц
- Порты – поддерживает очередь (упорядоченный список сообщений)
- Сообщения

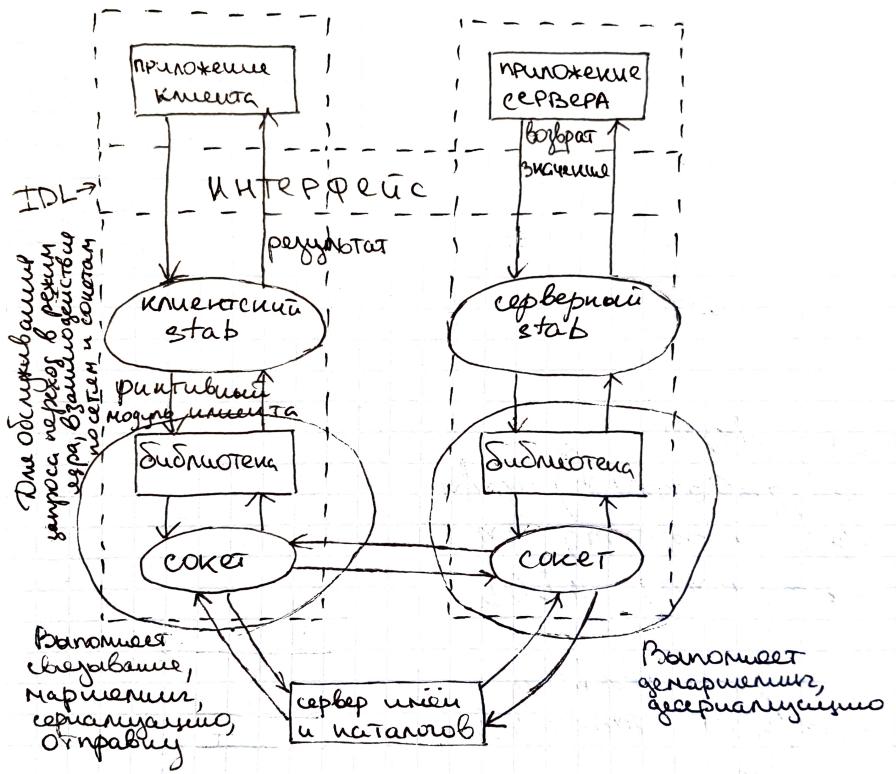
### Достоинства

- Выделено микроядро (низкоуровневые функции, связанные с аппаратной частью), а высокуюровневые функции вынесены
- Позволяет изменять функции неперекомпилируя ядро

### Недостатки

- Низкая производительность (в связи с тем, что количество переключений в режим ядра возрастает в 2 раза)
- Возникают задержки при передаче сообщений моделью клиент-сервер, поскольку взаимодействие должно быть надежным.

### Модель клиент-сервер



### RPC (вызов удаленных процедур).

Механизм реализованный первоначально в Unix, с помощью которого один процесс активизирует другой процесс на удаленной или той же самой машине для запуска/выполнения функции от своего имени.

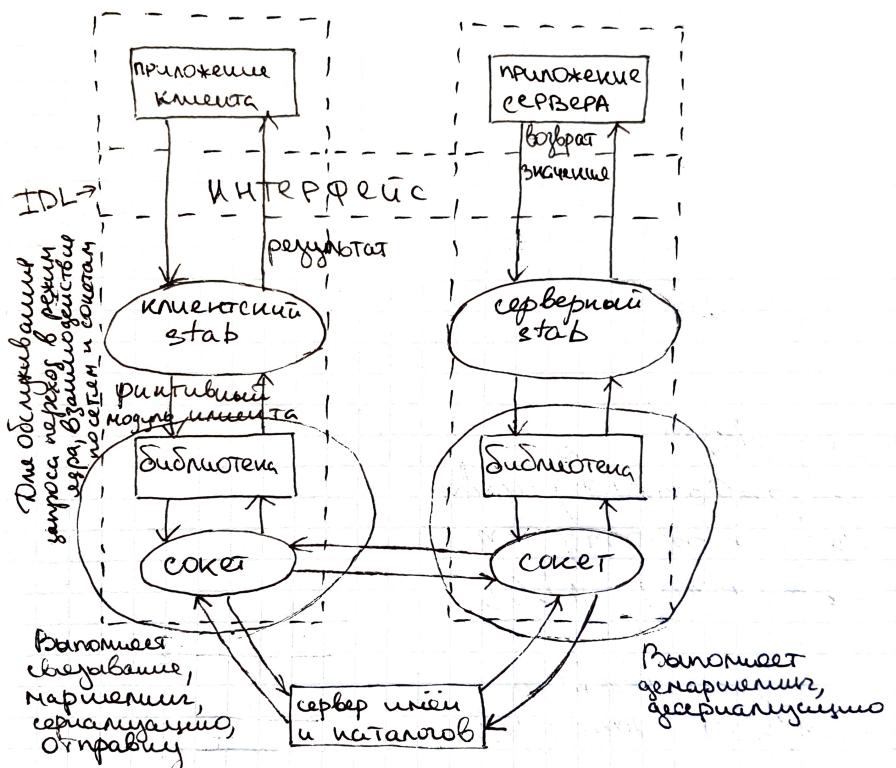
Вызов локальной процедуры – процесс вызывает функцию и передает ей данные (фактические параметры) и собственно ждет, когда будет возвращен результат выполнения этой функции.

Спецификация RPC – эту функцию выполняет другой процесс.

Такое взаимодействие выполняется по **схеме клиент-сервер**.

Процесс,зывающий RPC, является клиентским, а процесс, который выполняет RPC функцию – является серверным.

RPC является низкоуровневым средством взаимодействия, от этого не становится менее интересным. У RPC есть особенности, главное из которых является то, что механизм RPC скрывает от пользователя этот механизм удаленного доступа. Пользователь RPC может не заморачиваться на то, что он обращается к удаленной машине. RPC появились в 80-е.



Клиент и сервер связаны с стаб на этапе компоновки.

Стаб (фиктивный модуль) – генерируется компилятором из определения интерфейса, который используется клиентом.

В результате действий система создает пакет RPC, а также создает одно или несколько обращений к серверу.

Выполнение интерфейсов с помощью специального языка – IDL.

Клиентский сервер подключается через RPC.

Передаются форматированные данные (маршлинг и сериализация).

Маршлинг – перекомпоновка и упаковка данных в связи с требованиями системы.

Сериализация – преобразование сообщения в серию байтов.

Клиент и сервер могут быть написаны на разных языках, поскольку действия выполняются специальными переходниками.

2 способа связи:

- Статические: указание в клиентском приложении сетевого адреса сервера.

Такой подход имеет недостаток, т.е. его крайнюю не гибкость, а именно, сервер может быть перемещен или может быть увеличено число серверов, кроме того, может быть изменен интерфейс.

- Динамические: Binding, создается специализированная служба отвечающая за локализацию серверов.

Чтобы данные были машинно независимыми их преобразуют в XDR – external data representation.

## 26.11 Тупики: определение тупиковой ситуации для повторно используемых ресурсов, четыре условия возникновения тупика, обход тупиков - алгоритм банкира.

**Повторно-используемые ресурсы** – количество в системе постоянно и при использовании они не изменяются (или редко): аппаратура (ОП, ЦП), реenterабельные коды, системные таблицы (изменения в них могут вноситься только супервизором), процедуры ОС (так как они являются реenterабельными).

**Потребляемые ресурсы** – количество в ОС переменно и произвольно: сообщения. Процесс может создать любое количество сообщений. Процесс получения сообщения заканчивается его уничтожением.

**Тупик** – ситуация, возникающая в результате монопольного использования разделяемых ресурсов, когда процесс, владея ресурсом, запрашивает другой ресурс, занятый непосредственно или через цепочку запросов другими процессами, ожидающими освобождение ресурса, занятого 1-м процессом.

Были сформулированы **4 условия возникновения тупиков**, то есть чтобы мог возникнуть тупик необходимо выполнение 4 условий:

1. Наличие взаимоисключения (когда процессы монопольно используют предоставляемые им ресурсы)
2. Ожидание (процессыдерживают занятые им ресурсы, ожидая предоставления дополнительных ресурсов)
3. Неперераспределемость (ресурс нельзя отобрать у процесса до завершения процесса или добровольного освобождения ресурса процессом)
4. Круговое ожидание (если возникает замкнутая цепь запросов на ресурсы, в которой каждый процесс занимает ресурс необходимый другому процессу для продолжения выполнения)

Запросы ресурсов специальным образом анализируются. Наиболее известным является **алгоритм банкира** (Дейкстры). По его мнению действия алгоритма похожи на действия банкира. У банкира есть некоторый капитал и ему интересно его наращивать. Очень часто происходит ситуация, когда заемщик на может вернуть деньги, но чтобы иметь такую возможность ему нужна дополнительная сумма. Банкир должен анализировать запросы заемщиков и удовлетворять только те, кто вернёт ему деньги. В системе в виде банкира выступает менеджер ресурсов. Процессы это заемщики, они делают заявки на ресурсы. В заявке процесс указывает свою максимальную потребность в ресурсе каждого типа. В процессе выполнения процесс не может запросить количество ресурса контрактного типа больше, чем он указал в своей заявке. Такой подход позволяет проводить анализ возникающей ситуации удовлетворять только запросы тех процессов, которые могут гарантированно завершиться и не создавать тупиковые ситуации. Существуют ограничения:

1. Число процессов известно и фиксировано
2. Число ресурсов известно и фиксировано
3. Процесс не может в своей заявке указать число ресурсов больше, чем имеется в системе
4. Процесс не может подучить ресурсов больше, чем указано у него в заявке
5. Сумма всех ресурсов указанных в заявке не может превышать количество ресурсов в системе

Каждый запрос проверяется по отношению к числу ресурсов в заявке а также по отношению к числу свободных ресурсов которые в данный момент имеются в системе, при этом менеджер ресурсов ищет такую последовательность процессов, которая может гарантированно завершиться.

Таким образом всякий раз когда процесс делает новый запрос менеджер должен найти успешно завершающуюся последовательность процессов и только в этом случае запрос процесса может быть удовлетворён. Поэтому необходимо каждый раз исследовать  $n!$  Последовательностей прежде чем будет принято заключение о состоянии системы. Очевидно, что система должна удовлетворять только те запросы в результате удовлетворения которых ее состояние остаётся надежным. Запросы которые могут привести систему в ненадёжное состояние откладываются до момента когда такой запрос может быть выполнен. Система всегда поддерживает я в надежном состоянии и рано им поздно все запросы могут быть удовлетворены. Ограничения являются крайне жесткими, алгоритм также пожирает процессорное время, поэтому только в теории. Кроме того процессы должны знать свою максимальную потребность в ресурсах различных типов.

## 26.12 Прерывание int 8h (реальный режим) - функции.

### 1. Инкремент счетчика реального времени

В DOS есть область данных BIOS там находится по известному адресу счетчик реального времени (оперативная память). В компьютере есть энергонезависимая память CMOS микросхема (там и находится счетчик реального времени, при включении компьютера копируется в область данных BIOS).

2. Посылка в порт дисковода команды на отключение моторчика дисковода.
3. Вызов пользовательского прерывания 1Ch.