1. **Общее представление об операционных системах. Местоположение ОС. Функции ОС. Развитие операционных систем.**

   Операционная система — абстракция, которая помогает пользователю без особых усилий пользоваться компьютером, а программисту облегчает жизнь, при написании прикладных программ.

   Операционная система является надстройкой над уродливым аппаратным обеспечением, т.е. операционная система — прикладная программа, содержащая очень много строк кода, но находящаяся в режиме работы ядра, это её отличает от простых программ.

   Функции ОС — увеличение уровня абстракции и управление ресурсами (мультиплексирование во времени и в пространстве, сохранение целостности данных и обеспечение защиты при многопользовательском режиме).

   Этапы развития:

   1) Электронные лампы
   2) Транзисторы и системы пакетной обработки данных (*Системы пакетной обработки* предназначены для решения задач, которые не требуют быстрого получения результатов. Главной целью ОС пакетной обработки является максимальная пропускная способность или решение максимального числа задач в единицу времени.)
   3) Интегральные схемы и многозадачность (многозадачность, spooling, разделение времени)
   4) Персональные компьютеры (GUI, сетевые ОС, рапспределнные ОС)
   5) Компьютеры пятого поколения

2. **Компоненты ЭВМ. Процессоры. Конвеер. Суперскалярные процессоры. Многопоточность и многоядерность.**

   Компоненты ЭВМ: ЦП, ОЗУ, Монитор, Клавиатура (устройства ввода/вывода), USB, Жесткий диск, Шины, Контроллеры.

   Процессор: регистры, счетчик команд, указатель стека, PSW (Биты условия, биты управления приоритетом ЦП, биты режима ядра)

   Конвейер имеет несколько стадий: получение инструкций, декодирование, выполнение, доступ к памяти, запись в регистр. Минус в том, что могут появляться пузыри — момент, когда один из

этапов не успел закончить свою работу, а последующий уже закончил.

Суперскалярность — структура, схожая с конвейером, но имеющая несколько блоков для декодирования, получения инструкций и т.д.

Многопоточность — свойство операционной системы, состоящее в том, что процесс, порождённый в операционной системе, может состоять из нескольких потоков, выполняющихся параллельно либо псевдопараллельно, то есть без предписанного порядка во времени.

Многоядерность — центральный процессор содержащит два и более вычислительных ядра на одном процессорном кристалле или в одном корпусе.

3. **Компоненты ЭВМ. Память в ЭВМ. Шины. Системные вызовы.**

Компоненты ЭВМ: ЦП, ОЗУ, Монитор, Клавиатура (устройства ввода/вывода), USB, Жесткий диск, Шины, Контроллеры.

Память: регистры, кэш, ОЗУ, SSD, HDD, flash, магнитная лента.

Шины: кэш, локальная, памяти, оперативная, IDE, SCSI, USB, PCI.

Системные вызовы: для управления процессами, для управления файлами, для управления каталогами и другое. Команда помещается в стек, затем помещаются её аргумент, кстати стек растет вниз, далее осуществляется переход в режим ядра, выполнение и возврат в вызвавший процесс.

4. **Классификация ОС по функциональным характеристикам.**

Мэйнфреймов — больших компьютеров, которые еще используются в центрах данных корпораций. Мэйнфреймы отличаются от персональных компьютеров по возможностям ввода-вывода.

Серверные ОС — системы, которые работают на серверах и представляют собой очень большие персональные компьютеры, рабочие станции или мэйнфреймы.

Многопроцессорные ОС — системы способные выполнять несколько задач одновременно, но их устройство и алгоритмы работы гораздо сложнее, чем у однопроцессорных.

ПК — их работа заключается в предоставлении удобного интерфейса для одного пользователя.

КПК — схожи с ПК ОС, но ПЗУ не HDD, а flash memory, в настоящий момент их вытеснили смартфоны.

Встроенные ОС — простые операционные системы, устанавливаемые в принтерах, кассовых аппаратах и других внешних устройствах.

ОС сенсорных узлов — каждый сенсорный узел является настоящим компьютером, оснащенным процессором, оперативной памятью и постоянным запоминающим устройством, а также одним или несколькими датчиками. На нем работает небольшая, но настоящая операционная система, обычно управляемая событиями — откликающаяся на внешние события или периодически производящая измерения по сигналам встроенных часов.

ОС реального времени — системы с гарантированным временем реакции на событие, используются в системах технологического управления атомными станциями, химическими производствами и пр. Существует два вида таких ОС: жесткогая и габкая система real time.

ОС смарт–карт — самые маленькие операционные системы, которые работают на смарт–картах. Смарт–карты представляют собой устройства размером с кредитную карту, содержащие центральный процессор.

5. **Структурная классификация ОС.**

Монолитные ОС — По сути дела, ОС — это обычная программа, поэтому было бы логично и организовать ее так же, как устроено большинство программ, то есть составить из процедур и функций. В этом случае компоненты операционной системы являются не самостоятельными модулями, а составными частями одной большой программы. Такая структура операционной системы называется монолитным ядром (monolithic kernel). Монолитное ядро представляет собой набор процедур, каждая из которых может вызвать каждую. Все процедуры работают в привилегированном режиме. Таким образом, монолитное ядро — это такая схема операционной системы, при которой все ее компоненты являются составными частями одной программы, используют общие структуры данных и взаимодействуют друг с другом путем непосредственного вызова процедур. Для монолитной операционной системы ядро

совпадает со всей системой.

Многоуровневые ОС — Продолжая структуризацию, можно разбить всю вычислительную систему на ряд более мелких уровней с хорошо определенными связями между ними, так чтобы объекты уровня N могли вызывать только объекты уровня N - 1. Нижним уровнем в таких системах обычно является hardware, верхним уровнем — интерфейс пользователя. Чем ниже уровень, тем более привилегированные команды и действия может выполнять модуль, находящийся на этом уровне. Впервые такой подход был применен при создании системы THE (Technishe Hogeschool Eindhoven) Дейкстрой (Dijkstra) и его студентами в 1968 г.

Микроядра — Современная тенденция в разработке операционных систем состоит в перенесении значительной части системного кода на уровень пользователя и одновременной минимизации ядра. Речь идет о подходе к построению ядра, называемом микроядерной архитектурой (microkernel architecture) операционной системы, когда большинство ее составляющих являются самостоятельными программами. В этом случае взаимодействие между ними обеспечивает специальный модуль ядра, называемый микроядром. Микроядро работает в привилегированном режиме и обеспечивает взаимодействие между программами, планирование использования процессора, первичную обработку прерываний, операции ввода—вывода и базовое управление памятью. Основное достоинство микроядерной архитектуры – высокая степень модульности ядра операционной системы. Это существенно упрощает добавление в него новых компонентов. В микроядерной операционной системе можно, не прерывая ее работы, загружать и выгружать новые драйверы, файловые системы и т. д. Существенно упрощается процесс отладки компонентов ядра, так как новая версия драйвера может загружаться без перезапуска всей операционной системы. Компоненты ядра операционной системы ничем принципиально не отличаются от пользовательских программ, поэтому для их отладки можно применять обычные средства. Микроядерная архитектура повышает надежность системы, поскольку ошибка на уровне непривилегированной программы менее опасна, чем отказ на уровне режима ядра.

В то же время микроядерная архитектура операционной системы вносит дополнительные накладные расходы, связанные с передачей сообщений, что существенно влияет на производительность. Для того чтобы микроядерная операционная система по скорости не

уступала операционным системам на базе монолитного ядра, требуется очень аккуратно проектировать разбиение системы на компоненты, стараясь минимизировать взаимодействие между ними. Таким образом, основная сложность при создании микроядерных операционных систем – необходимость очень аккуратного проектирования.

Клиент–серверная ОС — Небольшая вариация идеи микроядер выражается в обособлении двух классов процессов: серверов, каждый из которых предоставляет какую—нибудь службу, и клиентов, которые пользуются этими службами. Эта модель известна как клиент—серверная. Достаточно часто самый нижний уровень представлен микроядром, но это не обязательно. Вся суть заключается в наличии клиентских процессов и серверных процессов.

Связь между клиентами и серверами часто организуется с помощью передачи сообщений. Чтобы воспользоваться службой, клиентский процесс составляет сообщение, в котором говорится, что именно ему нужно, и отправляет его соответствующей службе. Затем служба выполняет определенную работу и отправляет обратно ответ. Если клиент и сервер запущены на одной и той же машине, то можно провести определенную оптимизацию, но концептуально здесь идет речь о передаче сообщений. Сервер выделяет отдельный поток/процесс для работы с клиентом.

Виртуальные машины — программная и/или аппаратная система, эмулирующая аппаратное обеспечение некоторой платформы; или виртуализирующая некоторую платформу и создающая на ней среды, изолирующие друг от друга программы и даже операционные системы (песочница — Песочница обычно представляет собой жёстко контролируемый набор ресурсов для исполнения гостевой программы — например, место на диске или в памяти); также спецификация некоторой вычислительной среды (например: «виртуальная машина языка программирования Си»).

Стоит сказать про гипервизоры: Гипервизор (англ. Hypervisor) или Монитор виртуальных машин (в компьютерах) — программа или аппаратная схема, обеспечивающая или позволяющая одновременное, параллельное выполнение нескольких операционных систем на одном и том же хост–компьютере. Гипервизор также обеспечивает изоляцию операционных систем друг от друга, защиту и безопасность, разделение ресурсов между различными запущенными ОС и управление ресурсами.

Автономный гипервизор (Тип 1)

Имеет свои встроенные драйверы устройств, модели драйверов и планировщик и поэтому не зависит от базовой ОС. Так как автономный гипервизор работает непосредственно в окружении усечённого ядра, то он более производителен, но проигрывает в производительности виртуализации на уровне ОС и паравиртуализации. Xen может запускать виртуальные машины в паравиртуальном режиме (зависит от ОС).

Примеры: VMware ESX , Citrix, XenServer.

На основе базовой ОС (Тип 2, V)

Это компонент, работающий в одном кольце с ядром основной ОС (кольцо 0). Гостевой код может выполняться прямо на физическом процессоре, но доступ к устройствам ввода-вывода компьютера из гостевой ОС осуществляется через второй компонент, обычный процесс основной ОС — монитор уровня пользователя.

Примеры: Microsoft Virtual PC, VMware Workstation, QEMU, Parallels, VirtualBox. Гибридный (Тип 1+)

Гибридный гипервизор состоит из двух частей: из тонкого гипервизора, контролирующего процессор и память, а также работающей под его управлением специальной сервисной ОС в кольце пониженного уровня. Через сервисную ОС гостевые ОС получают доступ к физическому оборудованию.

Примеры: Microsoft Virtual Server, Sun Logical Domains, Xen, Citrix XenServer, Microsoft Hyper-V.

Экзоядро — Концепция экзоядра была предложена около 15 лет назад. Её создатели исходили из того, что современные операционные системы слишком сильно «виртуализируют» аппаратуру, лишая прикладные программы гибкости, а заодно сильно теряя в производительности и надёжности из-за чрезмерного усложнения самих ОС. Чтобы избежать этого, предлагается предоставлять прикладным программам лишь минимально необходимый слой абстракций, дозволяя им чуть ли не прямой доступ к управлению аппаратурой. Для упрощения же процесса создания прикладных программ, которые теперь вынуждены выполнять для себя те функции, которые обычно реализовывались где-то в недрах традиционных ОС, предназначены системные библиотеки (libOS) пользовательского режима.

Unix	**Windows**
Процесс создается *fork*,а далее exec* заменяет образ памяти	Процесс создается одним вызовом CreateProcess
Есть иерархия процессов (Наличие «зомби», «процессов—сирот»)	Каждый процесс независим

6. **Понятие процесса. Память процесса. Инициализация и завершение процесса. Отличие между процессами в Windows и Unix.**

   Процесс — это абстракция, описывающая выполняемую программу. Обеспечение параллелизма и псевдопараллелизма; Экземпляр выполняемой программы, включая значение счетчика команд, регистров и переменных.

   Память процесса — состоит из трех частей: код программы (текст), переменные (данные) и стек. Данные растут вверх, а стек вниз, между ними находится неиспользованное адресное пространство.

   Инициализация процесса — Инициализация системы, Работающий процесс осуществляет системный вызов создания нового процесса(иерархия процессов), Создание процесса по запросу пользователя, Инициализация пакетного задания

   Завершение процесса — Обычный выход, Выход по обрабатываемой ошибке, Возникновение фатальной ошибки, Уничтожение другим процессом

7. **Понятие процесса. Состояния процесса. Таблица процессов. Механизм прерываний. Моделирование режима многозадачности.**

   Процесс — это абстракция, описывающая выполняемую программу. Обеспечение параллелизма и псевдопараллелизма; Экземпляр выполняемой программы, включая значение счетчика команд, регистров и переменных.

   Состояния процесса — готов, действует, блокирован.

   Таблица процессов — Управление процессом (счетчик команд, регистры ...), Управление памятью (указатели на сегмент данных, стека...), Управление файлами (дескрипторы файлов, корневой каталог ...)

Механизм прерывания:

1) Оборудование помещает в стек счетчик команд и т.д.
2) Оборудование загружает из вектора прерываний новый счетчик команд
3) Процедура на ассемблере сохраняет регистры
4) Процедура на ассемблере устанавливает указатель на новый стек
5) Запускается процедура на Си, обслуживающая прерывание
6) Планировщик принимает решение какой процесс запускать следующим
7) Процедура на языке Си возвращает управление ассемблерному коду
8) Процедура на ассемблере запускает новый процесс

Моделирования режима многозадочности — ЦП $= 1 - p^n$, где p — ожидание ввода/вывода, а n — количество процессов. Пока процесс ждет ввода вывода, ЦП не простаивает, а работает.

8. **Понятие потока. Причины создания потоков. Реализация сервера для обработки запросов через однопоточный процессы и множество потоков. Реализация сервера для обработки запросов через машину с конечным числом состояний.**

Поток — Разновидность процесса внутри процесса; При старте у каждого процесса есть единственный поток управления.

Причины создания потоков — Упрощение модели программирования, когда в одной программе может выполняться несколько действий сразу; Создание потоков быстрее, чем создание процессов; Эффективность их использования в мультипроцессорных системах

Реализация многозадачности сервера с помощью одного потока — Основной цикл веб—сервера получает запрос, анализирует его и завершает обработку до получения следующего запроса. Ожидая завершения дисковой операции сервер простаивает и не обрабатывает запросы. В конечном итоге происходить значительное сокращение запросов, обрабатываемых в секунду.

Реализация многозадачности сервера с помощью нескольких потоков — диспетчер(поток) читает входящие запросы, анализирует и выбирает простаивающий поток (заблокированный), затем пробуждает поток (перевод из заблокированного в готовность)

Реализация многозадачности сервера через машину с конечным числом состояний — Сервер записывает состояние текущего запроса в таблицу, затем приступает к получению следующего события (новая задача, либо ответ от диска, связанный с предыдущей операцией). Если это новая задача ⇒ выполняется, если ответ от диска ⇒ в таблице ищется соответствующая информация и происходит обработка ответа.

Модель	**Характеристики**
Потоки	Параллельная работа, блокирующие системные вызовы
Однопоточный процесс	Отсутствие параллельной работы, блокирующие системные вызовы
Машина с конечным числом состояний	Параллельная работа, неблокирующие системные вызовы, прерывания

9. **Понятие потока. Причины создания потоков. Объекты, относящиеся к процессам и потокам. Стратегии реализации потоков (плюсы и минусы каждого подхода). Всплывающие потоки.**

Поток — Разновидность процесса внутри процесса; При старте у каждого процесса есть единственный поток управления.

Причины создания потоков — Упрощение модели программирования, когда в одной программе может выполняться несколько действий сразу; Создание потоков быстрее, чем создание процессов; Эффективность их использования в мультипроцессорных системах

Процесс	**Поток**
Адресное пространство	Счетчик команд
Глобальные переменные	Регистры
Открытые файлы	Стек
Дочерние процессы	Состояние
Необработанные аварийные сигналы	
Сигналы и обработчики сигналов	
Учетная информация	

В пользовательском пространстве:

Плюсы:

- Может быть реализован в ОС, которая не поддерживает потоки
- Переключение между потоками быстрее, чем при системных вызовах
- Позволяют каждому процессу иметь свои собственные настройки алгоритма планирования

Минусы:

- Реализация блокировок
- Ошибка отсутствия страницы
- Никакой поток не сможет выполниться, пока текущий не предоставит ему ЦП

В ядре:

Плюсы:

- Таблица потоков и таблица процессов в ядре
- Все вызовы, блокирующие поток, реализованы в ядре
- Повторное использование поток, т.к. создание/удаление очень дорогие операции в ядре
- Не требуются неблокирующие системные вызовы

Минусы:

- Много времени на системные вызовы
- ХЗ

Гибридный — алгоритм заключается в использовании потоков в ядре и в дальнейшем их разветвлении в пространстве пользователя.

Плюсы:

- Быстрое переключение между потоками
- Ядро планирует работу только своих потоков
- Гибкость, относительно предыдущих способов

Всплывающий поток — на каждое новое сообщение мы выделяем отдельный новый поток. Нам не надо запоминать регистры, стеки и прочее, т.е. ничего не надо восстанавливать. Значительно уменьшаем задержку между поступлением и началом обработки сообщения. Но требуется проводить предварительное планирование.

10. **Понятие критической области. Взаимоисключение с активным ожиданием. Запрещение прерываний, блокирующие переменные и строгое чередование.**

Критическая секция — часть программы, в которой используется доступ к общей памяти

Взаимоисключение с активным ожиданием:

1) Запрещение прерываний — после входа в критическую область процесс невозможно будет прервать, т.е. мы заберем все время работы ЦП себе, тем самым мы решим проблему, но есть и минусы: не разумно давать пользовательским процессам полномичия выключать все прерывания, т.к. может произойти крах системы. С другой стороны, выключение прерываний на несколько секунд в режиме ядра очень полезная вещь, когда ядро обновляет свои списки и т.д. Но сейчас кругом многопроцессорные компьютеры и в нашем случае, если ЦП нас не может прервать, то любой другой процесс может.
2) Блокирующие переменные — используется одна общая переменная (исходное значение равно 0). При входе в крит.обл. процесс сначала проверяет её значение, если оно равно нулю, то он устанавливает значение 1 и входит в крит область, если значение равно 1, то процесс в глухом цикле ждет пока оно не станет равным 0. Если между этими действиями планировщик переключит процессы, то будет беда.
3) Строгое чередование — у нас есть переменная *turn*, она равна 0, каждый процесс проверяет значение *turn*, если оно равно 0 $\Rightarrow$ процесс 0 заходит в крит. область, а остальные крутятся в цикле (активное ожидание — плохо). Процесс 0 отрабатывает и устанавливает *turn* в 1. Минус: если один процесс работает существенно быстрее другого, то данный алгоритм не подойдет.

4) Алгоритм Петерсона

```
#define FALSE   0
#define TRUE    1
#define N       2
int turn;
int interested[N];
void enter_region (int process) {
    int other = 1 - process;
    interested[process] = TRUE;
    turn = process;
    while (turn == process && interested[other] == TRUE);
}
void leave_region (int process) {
    interested[process] = FALSE;
}
```

5) Команды TSL — неделимая операция (никакой другой процесс не может получить доступ во время чтения и записи), считывает значение из *lock* в регистр, а в *lock* пишет ненулевое значение. TSL блокирует шину памяти и никто не может получить доступ.

```
enter_region:
    TSL REGISTER.LOCK | копируем lock и присваем ей 1
    CMP REGISTER.#0   | было ли lock = 0?
    JNE enter_region  | если да, то блокировка уже была
                      | установлена и мы уходим в цикл

    RET               | вход в критическую область
leave_region:
    MOVE LOCK.#0      | присваиваем lock = 0
    RET               | возврат управление программе
```

11. Понятие критической области. Приостановка и активизация. Семафор и условная переменная.

Приостановка и активизация — семафор, мьютекс, монитор.

Семафор — Целое неотрицательное число, Down, Up.

```
#include <cstdlib>
#include <iostream>

#include <thread>
#include <mutex>
```

```
6  #include <condition_variable>
7  #include <chrono>
8
9  #define STORAGE_MIN 10
10 #define STORAGE_MAX 20
11
12 using namespace std;
13
14 int storage = STORAGE_MIN;
15
16 mutex globalMutex;
17 condition_variable condition;
18
19 /* Функция потока потребителя */
20 void consumer()
21 {
22     int toConsume = 0;
23
24     while(1)
25     {
26         unique_lock<mutex> lock(globalMutex);
27         /* Если значение общей переменной меньше максимального,
28          * то поток входит в состояние ожидания сигнала о
    достижении
29          * максимума */
30         if (storage < STORAGE_MAX)
31         {
32             condition.wait(lock);
33             toConsume = storage-STORAGE_MIN;
34         }
35         /* "Потребление" допустимого объема из значения общей
36          * переменной */
37         storage -= toConsume;
38     }
39 }
40
41 /* Функция потока производителя */
42 void producer()
43 {
44
45     while (1)
46     {
47             this_thread::sleep_for(chrono::milliseconds(200));
```

```
48
49          unique_lock<mutex> lock(globalMutex);
50          ++storage;
51          /* Если значение общей переменной достигло или
    превысило
52           * максимум, поток потребитель уведомляется об этом */
53          if (storage >= STORAGE_MAX)
54              condition.notify_one();
55      }
56  }
57
58  int main(int argc, char *argv[])
59  {
60      thread thProducer(producer);
61      thread thConsumer(consumer);
62
63      thProducer.join();
64      thConsumer.join();
65
66      return 0;
67  }
```

12. **Понятие взаимоблокировки. Условия возникновения. Моделирование взаимоблокировок.**

Взаимоблокировка может происходить когда несколько процессов борются за один ресурс.

Ресурсы бывают выгружаемые и невыгружаемые, аппаратные и программные.

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

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

Проблема взаимоблокировок процессов возникает при борьбе за невыгружаемые ресурсы.

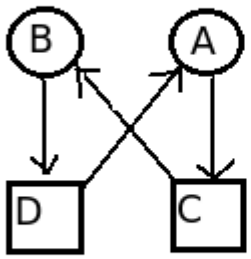
Условия необходимые для взаимоблокировки:

1) Условие взаимного исключения — в какой–то момент времени, ресурс занят только одним процессом или свободен.
2) Условие удержания и ожидания — процесс, удерживающий ресурс может запрашивать новые ресурсы.

3) Условие отсутствия принудительной выгрузки ресурса.

4) Условие циклического ожидания — должна существовать круговая последовательность из процессов, каждый, из которого ждет доступа к ресурсу, удерживаемому следующим членом последовательности.

Моделирвоание взаимоблокировок:



13. **Понятие взаимоблокировки. Способы борьбы со взаимоблокировками. Способ борьбы «Игнорирование».**

Взаимоблокировка может происходить когда несколько процессов борются за один ресурс.

Ресурсы бывают выгружаемые и невыгружаемые, аппаратные и программные.

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

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

Способы борьбы со взаимоблокировками:

1) Игнорирование

2) Обнаружение и восстановление

3) Динамическое уклонение

4) Предотвращение за счет подавления условий взаимоблокировок

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

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

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

14. **Понятие взаимоблокировки. Способы борьбы со взаимоблокировками. Обнаружение и восстановление.**

Взаимоблокировка может происходить когда несколько процессов борются за один ресурс.

Ресурсы бывают выгружаемые и невыгружаемые, аппаратные и программные.

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

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

Способы борьбы со взаимоблокировками:

1) Игнорирование
2) Обнаружение и восстановление
3) Динамическое уклонение
4) Предотвращение за счет подавления условий взаимоблокировок

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

Есть два способа обнаружения:

- Обнаружение взаимоблокировки при наличии одного ресурса каждого типа (запускаем обход в ширину от каждой вершины крафа, если находим цикл $\Rightarrow$ взаимоблокировка)
- Обнаружение взаимоблокировки при наличии нескольких ресурсов каждого типа (потребуется два вектора (Все ресурсы(E) и доступные(A)), две матрицы (текущее распред.(C) и

требуемые ресурсы(R)), находим $R_i \mid R_i \leq A_i$ и освобождаем C_i)

Восстановление

- За счет приоритетного овладения ресурса
- Rollback(откат)
- Уничтожение процесса

15. **Понятие взаимоблокировки. Способы борьбы со взаимоблокировками. Уклонение от взаимоблокировки. Траектория ресурса.**

Взаимоблокировка может происходить когда несколько процессов борются за один ресурс.

Ресурсы бывают выгружаемые и невыгружаемые, аппаратные и программные.

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

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

Способы борьбы со взаимоблокировками:

1) Игнорирование
2) Обнаружение и восстановление
3) Динамическое уклонение
4) Предотвращение за счет подавления условий взаимоблокировок

Уклонение от взаимоблокировки — В этом способе ОС должна знать, является ли предоставление ресурса безопасным или нет.

- Траектория
- Поддержание безопасного состояния

Траектория — Т.к. процессор предоставляется поочередно, траектория может продолжаться только параллельно осям. Чтобы избежать тупика, процессам надо обойти прямоугольник, охватывающий всю заштрихованную область.

16. **Понятие взаимоблокировки. Способы борьбы со взаимоблокировками. Уклонение от взаимоблокировки. Алгоритм банкира.**

Взаимоблокировка может происходить когда несколько процессов борются за один ресурс.

Ресурсы бывают выгружаемые и невыгружаемые, аппаратные и программные.

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

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

Способы борьбы со взаимоблокировками:

1) Игнорирование
2) Обнаружение и восстановление
3) Динамическое уклонение
4) Предотвращение за счет подавления условий взаимоблокировок

Уклонение от взаимоблокировки — В этом способе ОС должна знать, является ли предоставление ресурса безопасным или нет.

- Траектория
- Поддержание безопасного состояния

Алгоритм банкира:

- Банкиру поступает запрос от клиента на получение кредита.
- Банкир проверяет, приводит ли этот запрос к небезопасному состоянию.
- Банкир в зависимости от этого дает или отказывает в кредите.

Также алгоритм банкира можно применять для нескольких видов ресурсов.

17. **Понятие взаимоблокировки. Способы борьбы со взаимоблокировками. Предотвращение взаимоблокировки.**

Взаимоблокировка может происходить когда несколько процессов борются за один ресурс.

Ресурсы бывают выгружаемые и невыгружаемые, аппаратные и программные.

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

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

Способы борьбы со взаимоблокировками:

1) Игнорирование
2) Обнаружение и восстановление
3) Динамическое уклонение
4) Предотвращение за счет подавления условий взаимоблокировок

Предотвращения:

1) Предотвращение условия взаимного исключения — Можно минимизировать количество процессов борющихся за ресурсы. Например, с помощью спулинга для принтера, когда только демон принтера работает с принтером.
2) Предотвращение условия удержания и ожидания — Один из способов достижения этой цели, это когда процесс должен запрашивать все необходимые ресурсы до начала работы. Если хоть один ресурс недоступен, то процессу вообще ничего не предоставляется.
3) Предотвращение условия отсутствия принудительной выгрузки ресурса — Можно выгружать ресурсы, но могут быть проблемы с принтером.
4) Предотвращение условия циклического ожидания — a) Процесс сначала должен освободить занятый ресурс, прежде чем занять новый. b) Можно пронумеровать все ресурсы (и упорядочить), и процессы должны запрашивать ресурсы только по возрастающему порядку.

18. **Виды блокировок. Зависание.**

Виды блокировок:

- Оптимистическая — не ограничивает доступ к ресурсу, но перед модификацией запрашивает у каждого ресурса некую переменную V с первичным значением равным 0 и проводит модификацию. Перед моментом окончания берется атрибут V и

сравнивается с нашим, если во время нашей модификации объект был изменен, происходит rollback нашей транзакции, если объект не был изменен, тогда наши модификации сохраняются, а значение V увеличивается инкрементально.

- Пессимистическая — ограничение по доступу накладывается на все ресурсы, которые данный процесс может затронуть. Во время пессимистической блокировки невозможен доступ к заблокированным объектам из сторонних процессов. После завершения процесса гарантируется непротиворечивость модифицированных данных.
- При обмене данными — например, в сетях, в которых один и более процессов связываются путем обмена сообщениями. Общая договоренность предполагает, что процесс A отправляет сообщение–запрос процессу B, а затем блокируется до тех пор, пока B не пошлет назад ответное сообщение. Предположим, что сообщение-запрос где–то затерялось. Процесс A заблокирован в ожидании ответа. Процесс B заблокирован в ожидании запроса на какие-либо его действия. В результате возникает взаимоблокировка.
- Активные — появляются только в SMP(симметр. мультипроц.) системах, когда процессор в ожидании недоступного пока ресурса не переводится в блокированное состояние, а «накручивает» в ожидании освобождения ресурса «пустые» циклы. В этом случае, процессор не освобождается на выполнение другого ожидающего процесса в системе, а продолжает активное выполнение («пустых» циклов) в контексте текущего процесса.

Зависание — В динамической системе запросов ресурсов происходит постоянно. Для того чтобы принять решение, кто и когда какой ресурс получит, нужна определенная политика. Эта политика, хотя бы и разумная, может привести к тому, что некоторые процессы никогда не будут обслужены, даже если они не находятся в состоянии взаимоблокировки.

В качестве примера рассмотрим распределение принтера. Допусти он работает с помощью крутого алгоритма распределения и не приводит к взаимоблокировкам. А что будет если несколько процессов разом захотели овладеть принтером? Допустим принтер достанется тому, у кого самый маленький объем файла (для печати), тогда если поток процессов с короткими файлами не иссякает $\Rightarrow$ процесс

с огромным файлом зависнет. При помощи FIFO можно избежать зависаний.

19. **Планирование процессов. Условия планирования. Виды планирования, основные характеристики при планировании.**

Планирование — обеспечение поочередного доступа процессов к одному процессору.

Условия планирования:

1) Когда создается процесс
2) Когда процесс завершает работу
3) Когда процесс блокируется на операции ввода/вывода, семафоре, и т.д.
4) При прерывании ввода/вывода.

Виды планирования:

1) Все системы (Справедливость — каждому процессу справедливую долю процессорного времени. Контроль над выполнением принятой политики. Баланс — поддержка занятости всех частей системы (например: чтобы были заняты процессор и устройства ввода/вывода)
2) Пакетные системы (Пропускная способность — количество задач в час. Оборотное время — минимизация времени на ожидание обслуживания и обработку задач. Использование процесса — чтобы процессор всегда был занят.)
3) Интерактивные системы (Время отклика — быстрая реакция на запросы. Соразмерность — выполнение ожиданий пользователя (например: пользователь не готов к долгой загрузке системы))
4) Системы реального времени (Окончание работы к сроку — предотвращение потери данных Предсказуемость — предотвращение деградации качества в мультимедийных системах (например: потерь качества звука должно быть меньше чем видео)
5) Мультипроцессорные системы

20. **Планирование процессов. Планирование в пакетных системах.**

Первым пришел — первым ушел:

Плюсы:

- Простата
- Справедливость (как в очереди покупателей, кто последний пришел, тот оказался в конце очереди)

Минусы:

- Процесс, ограниченный возможностями процессора может затормозить более быстрые процессы, ограниченные устройствами ввода/вывода.

Кратчайшая задача — первая:

Преимущества:

- Уменьшение оборотного времени
- Справедливость (как в очереди покупателей, кто без сдачи проходит в перед)

Минусы:

- Длинный процесс занявший процессор, не пустит более новые краткие процессы, которые пришли позже.

Наименьшее оставшееся время выполнения — Аналог предыдущего, но если приходит новый процесс, его полное время выполнения сравнивается с оставшимся временем выполнения текущего процесса.

21. **Планирование процессов. Планирование в интерактивных системах. Циклическое и приоритетное планирование. Использование нескольких очередей. Выбор следующего самого короткого задания.**

Циклическое — Самый простой алгоритм планирования и часто используемый. Каждому процессу предоставляется квант времени процессора. Когда квант заканчивается процесс переводится планировщиком в конец очереди. При блокировке процессор выпадает из очереди.

Плюсы:

- Простата
- Справедливость (как в очереди покупателей, каждому только по килограмму)

Минусы:

- Если частые переключения (квант - 4мс, а время переключения равно 1мс), то происходит уменьшение производительности.
- Если редкие переключения (квант - 100мс, а время переключения равно 1мс), то происходит увеличение времени ответа на запрос.

Приоритетное — Каждому процессу присваивается приоритет, и управление передается процессу с самым высоким приоритетом. Приоритет может быть динамический и статический. Динамический приоритет может устанавливаться так: $P = \frac{1}{T}$, где T — часть использованного в последний раз кванта. Если использовано $\frac{1}{50}$ кванта, то приоритет 50. Если использован весь квант, то приоритет 1. Т.е. процессы, ограниченные вводом/вывода, будут иметь приоритет над процессами ограниченными процессором. Часто процессы объединяют по приоритетам в группы, и используют приоритетное планирование среди групп, но внутри группы используют циклическое планирование.

Использование нескольких очередей — наши процессы разбиты на приоритеты, внутри каждого приоритета находится очередь задач. Задачи с наивысшим приоритетом получали 1 квант времени, спускаясь по нисходящей — квант времени инкрементируется. Если процесс использовал все кванты времени $\Rightarrow$ его приоритет понижался.

Выбор самого короткого процесса — данный алгоритм применялся в пакетных системах. В нашем случае найти ту самую метрику, по которой будет определяться самый корокий процесс. Зная время работы на предыдущем шаге мы сможем определить какой процесс следует запустить сейчас:

- Пусть T_i — время работы i–ого процесса, тогда можно вычислять следующим образом: на первом шаге T_0, на втором $\frac{T_0 + T_1}{2}$, ..., на n–ом шаге — $\frac{T_0 + \ldots + T_n}{n}$

- Либо вот таким способом: T_0; $\frac{T_0}{2} + \frac{T_1}{2}$; $\frac{T_0}{4} + \frac{T_1}{4} + \frac{T_2}{2}$; ...

22. **Планирование процессов. Планирование в интерактивных системах. Лотерейное планирование. Справедливое планирование. Гарантированное планирование.**

Лотерейное планирование — Достойная идея, но труднореализуемая. Система распределяет «лотерейные билеты» между процессами, и «выигрывший» процесс получает 20 мс процессорного времни. В этом принципе возможна приоритетность — раздача нескольких билетов «выжным» процессам. Притом каждый процесс получает ресурсов примерно равные проценту имеющихся у него лотерейных билетов. Процессы могут передавать свои лотерейные билеты (клиент→сервер — клиент прервался, жёдт сервера, клиент отдаёт свои билеты серверному процессу). Данное планировае удобно при меняющихся неравнозначных по загруженности процессах. (Видеосервер с несколькими потоками разного битрейта, особенно переменного)

Справедливое планирование — Во внимание берётся тот, кто запускает процесс. Если один пользователь создал 9 процессов, а параллельный ему второй — 1, то при таком планировании система распределит время процессора между пользователями попалам, в то время как другие виды планирования отдали большинство процессорного времние первому пользователю. Данное планирование отведёт ровно 50% процессорного времени одному из дву пользоватлей, независимо от того, как он будет использовать эти 50%.

Гарантированное — Если в системе одновременно работают `k` пользователей, то одному будет предоставлено $\frac{1}{k}$ процессорного времени, а в системе с одним пользователем запущено `n` процессов, то каждому процессу достнется $\frac{1}{n}$ процессорного времени. Система сама следит за количестволм процессов и отслеживает промежутки процессорного времени, выделяемого каждому из процессов (или пользователей)

1) Каждому дается $K = \frac{X}{N}$ времени, где X — суммарное время работы процесса, N — количество процессов
2) Для каждого процесса считаем $M = \frac{U}{K}$, U — использованное время процессом

3) Когда наступает момент выбора следующего процесса, то выбирается тот, кто имеет min M.

4) Время работы процесса определяется пока его M не превысила M ближайшего конкурента

23. **Планирование процессов. Планирование в системах реального времени. Характеристики планирования в системах реального времени. Алгоритм RMS.**

Планирование в систеамх реального времени

Системы реального времени делятся на:

- жесткие (жесткие сроки для каждой задачи) — управление движением
- гибкие (нарушение временного графика не желательны, но допустимы) — управление видео и аудио

Внешние события, на которые система должна реагировать, делятся:

- периодические — потоковое видео и аудио
- непериодические (непредсказуемые) — сигнал о пожаре

Что бы систему реального времени можно было планировать, нужно чтобы выполнялось условие:

$$\sum_{i=1} \frac{T_i}{P_i} \leq 1$$

, где m — число периодических событий, i — номер события, P_i — период поступления события, T_i — время, которое уходит на обработку события.

Т.е. перегруженная система реального времени является не планируемой.

Статический алгоритм планирования RMS:

Процессы должны удовлетворять условиям:

1) Процесс должен быть завершен за время его периода

2) Один процесс не должен зависеть от другого

3) Каждому процессу требуется одинаковое процессорное время на каждом интервале

4) У непериодических процессов нет жестких сроков

5) Прерывание процесса происходит мгновенно

Приоритет в этом алгоритме пропорционален частоте.

Динамический алгоритм планирования EDF:

Наибольший приоритет выставляется процессу, у которого осталось наименьшее время выполнения. При больших загрузках системы EDF имеет преимущества.

Принцип работы EDF:

- Планировщик ведет список готовых процессов, отсортированный по крайним срокам выполнения
- В работу всегда берется процесс с самым близким крайним сроком выполнения
- При поступлении нового процесса на выполнение происходит перепланировка

Разница между RMS и EDF:

- Для EDF требуются накладные расходы на динамическую приоритезацию
- Для RMS есть рамки применимости —

$$\sum_{i=1}^{m} \frac{T_i}{P_i} \leq m(2^{\frac{1}{m}} - 1)$$