БЛЯТЬ ЕБАНЫЙ В РОТ ЭТОЙ ВИНДЫ (11) ПОСЛЕ ОБНОВЛЕНИЯ ПРИ ЗАПУСКЕ ЧЕРНЫЙ ЭКРАН АААААААААААААААААААААААА (простите)

Ответы на вопросы к экзамену по дисциплине "Операционные Систем"

Авторы:
Артемий (главный)((нет))
Ильюша (босяк)
Никита (второстепенный)
Наташка (вообще какая-то левая девка)
d20 aka ёбаный рот этого казино (какой-то левый чел)
Павел Ян
Lea_SK
_____Y__

2 января 2022 г.

Содержание

1 Назначение и функции ОС. 4

2 Основные понятие ОС. 5

3 Классификация ОС. 7

4 Иерархия запоминающих устройств. 9

5 Архитектура ОС. Назначение и функции основных подсистем ОС. 10

6 Прерывания. Классы и обработка прерываний. 12

7 Организация многопроцессорных и многоядерных систем. 13

8 Подсистема управления процессами ОС. Состояния процесса. 15

9 Описание и управление процессами. 17

10 Выполнение кода операционной системы. 18

11 Понятие потока. Различия потока и процесса. 19

12 Алгоритмы управления процессами. Алгоритмы, основанные на квантовании. 20

13 Алгоритмы управления процессами. Алгоритмы, основанные на приоритетах. 21

14 Типы процедур планирования процессов. 22

15 Проблема синхронизации процессов и потоков. Эффект «гонок». 23

16 Взаимоисключения: программный подход. 24

17 Организация взаимоисключения процессов и потоков. Критическая секция. Блокирующая переменная. 26

18 Организация взаимоисключения процессов и потоков. События. 27

19 Организация взаимоисключения процессов и потоков. Семафор. 28

20 Организация взаимоисключения процессов и потоков. Монитор. 29

21 Понятие тупика. Проблема тупиков. 31

22 Принципы взаимного блокирования. 32

23 Предотвращение и устранение взаимоблокировок. 34

24 Обнаружение взаимоблокировок 36

25 Задача об обедающих философах. 37

26 Задача читатель/писатель. 39

27 Задача производитель/потребитель. 41

28 Подсистема коммуникации. Базовые примитивы передачи сообщений. 44

29 Подсистема коммуникации. Блокирующие и неблокирующие примитивы. 45

30 Подсистема коммуникации. Буферизируемые и небуферизируемые примитивы передачи сообщений. 46

31 Подсистема коммуникации. Надежные и ненадежные примитивы передачи сообщений. 47

32 Алгоритмы взаимного исключения. Распределенный алгоритмы. 48

33 Алгоритмы взаимного исключения. Алгоритм Token Ring. 49

34 Алгоритмы взаимного исключения. Централизованный алгоритм. 50

35 Способы адресации в распределенных системах. 51

36 Алгоритм синхронизации логических часов. 52

37 Задачи подсистемы управления памятью ОС. 54

38 Методы загрузки программ в память. Виды адресов. 56

39 Алгоритмы распределения памяти без использования дискового пространства. Распределение памяти фиксированными разделами. 59

40 Алгоритмы распределения памяти без использования дискового пространства. Распределение памяти разделами переменной величины. Фрагментация. 61

41 Алгоритмы распределения памяти без использования дискового пространства. Распределение памяти перемещающимися разделами. 63

42 Алгоритмы распределения памяти с использованием дискового пространства. Страничное распределение памяти. 64

43 Алгоритмы распределения памяти с использованием дискового пространства. Сегментное распределение памяти. 67

44 Алгоритмы распределения памяти с использованием дискового пространства. Сегментно-страничное распределение памяти. 69

45 Свопинг 70

46 Организация кэш-памяти. 72

47 Виртуальная память. 75

48 Дисковое планирование. Параметры производительности диска. 76

49 RAID. Уровни RAID. 78

50 Определение файловой системы. Именование файлов. Атрибуты файлов. Методы обеспечения доступа к файлам. 79

51 Задачи файловой системы. Логическая организация файловой системы. 81

52 Логическая организация файла. 82

53 Физическая организация файла. 83

54 Отображаемые в память файлы. Использование файлов, проецируемых в память в разработке. 84

55 Распределенные файловые системы. Основные понятия. 85

56 Семантика разделения файлов. 86

57 Организация устройств ввода/вывода. 87

58 Подсистема управления вводом/выводом ОС. Физическая организация устройств ввода/вывода. 88

59 Основные проблемы организации ввода/вывода. Многослойная модель подсистемы ввода-вывода. 89

60 Организация программного обеспечения ввода/вывода. Драйверы устройств. 90

1 Назначение и функции ОС.

Операционная система (ОС) это набор системных управляющих программ, которые управляют устройствами компьютера (процессором, оперативной памятью, устройствами ввода/-вывода) и обеспечивают работу других программ, выполняя роль интерфейса между пользователем и компьютером.

Операционная система - это программа , контролирующая выполнение прикладных программ и исполняющая роль интерфейса между приложениями и аппаратным обеспечением компьютера.

Назначение ОС лучше всего передаётся её основной функцией - посреднической. Она заключается в обеспечении нескольких видов интерфейса:

1. между пользователем, ПО и АО компьютера
2. между ПО и АО
3. между разными ПО

Другие функции ОС:

1. распределительная - распределение ресурсов компьютера
2. разграничительная - разграничение доступа процессов к ресурсам
3. загрузка программ в оперативную память и их выполнение
4. стандартизированный доступ к устройствам ввода-вывода и периферийным устройствам
5. параллельное решение задач(многозадачность)
6. обеспечение взаимодействия между процессами

2 Основные понятие ОС.

Операционная система — это набор системных программ, управляющих аппаратным обеспечением компьютера и обеспечивающих работу других программ.

Операционная система предоставляет программный интерфейс для работы с аппаратным обеспечением компьютера но на более высоком уровне. Так в операционной системе есть понятие разделения времени — несколько процессов поочерёдно получают процессорное время для выполнения. С этим явлением также связано понятие многозадачности — несколько процессов может выполняться условно одновременно при наличии одного процессорного модуля (на компьютере процессор с одним ядром, но при этом на компьютере одновременно могут работать десятки программ).

Кроме аппаратных абстракций операционная система предоставляет различные механизмы безопасности. Так, например, существуют механизмы для защиты областей памяти процессов друг от друга (чтобы один процесс не нарушил работу другого).

2.1 Процесс

Процесс - это программа во время выполнения.

- У каждого процесса есть адресное пространство - список адресов виртуальной памяти, откуда процесс читает данные и куда их записывает.
- АП содержит выполняемую программу, ее данные и стек.

2.2 Файл

Файл – логически связанная совокупность данных или программ, для размещения которой во внешней памяти выделяется именованная область.

- У файла есть некоторые основные характеристики: имя файла, объём памяти, дата и время создания, спец. атрибуты - Read Only, Hidden, System

2.3 Архитектура операционной системы

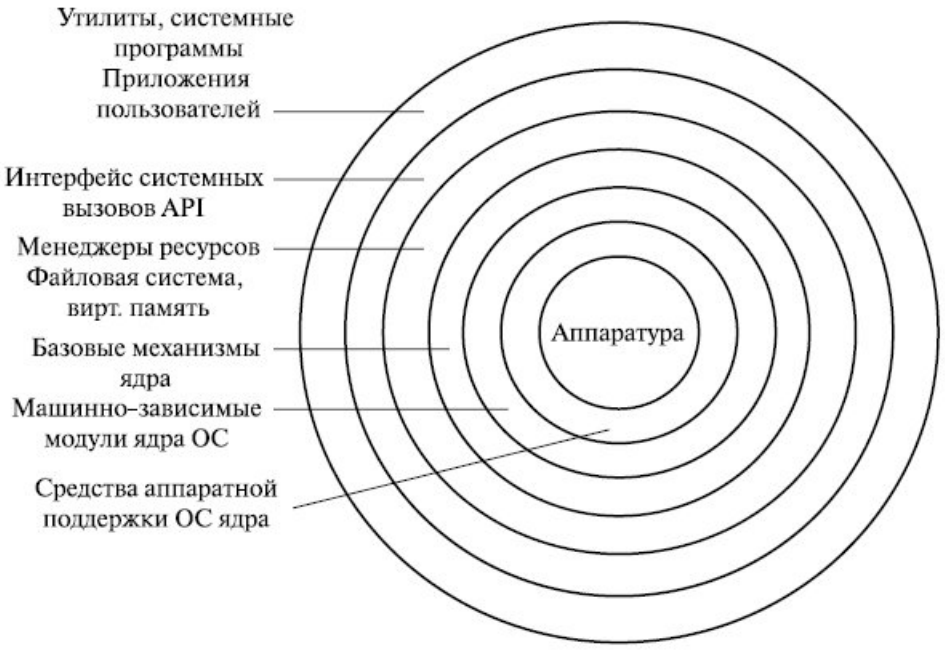

Рис. 1 – Общая архитектура операционной системы

3 Классификация ОС.

Операционные системы делятся по нескольким критериям.

1. Поддержка многозадачности. По числу одновременно выполняемых задач ОС делятся на:
   - однозадачные - MS-DOS. Однозадачные ОС выполняют функцию предоставления пользователю виртуальной машины. Они включают средства управления файлами, средства общения с пользователем.
   - многозадачные - UNIX. Многозадачные ОС, кроме упомянутых функций, управляют разделением совместно используемых ресурсов(процессор, оперативная память, файлы).
2. Поддержка многопользовательского режима. По числу одновременно работающих пользователей ОС делятся на:
   - однопользовательские - MS-DOS
   - многопользовательские - UNIX

   Главным отличием многопользовательских от однопользовательских систем является наличие средств защиты данных от несанкционированного доступа других пользователей.
3. Поддержка многонитиевости. Имеется в виду распараллеливание в рамках 1 задачи.
   - однонитевая
   - многонитевая

   Многонитевая ОС разделяет процессорное время между нитями задачи.
4. Многопроцессорная обработка(мультипроцессирование). По этому критерию ОС делятся на те, у которых есть средства поддержки многопроцессорной обработки, и те, у которых таких средств нет.
5. Вид многозадачности. По виду многозадачности(то, как ОС распределяет процессорное время между процессами или нитями) ОС делятся на:
   - с невытесняющей многозадачностью(активный процесс сам решает, когда отдать управление обратно операционной системе)
   - с вытесняющей многозадачностью(ОС может сама переключить активный процесс)

Таненбаум выделяет следующие виды операционных систем:

1. ОС для Мейнфреймов. Оптимизированы для обработки огромных объёмов данных и используются в корпоративном сегменте. Представителями являются OS/360 и OS390, но их вытесняют Unix-системы.
2. Серверные ОС. Служат для одновременной обработки большого количества запросов и применяются при создании веб-серверов. Примеры: Solaris, FreeBSD, Linux, Windows Server.
3. Многопроцессорные ОС. Применяются там, где система должна работать на аппаратном обеспечении состоящем из нескольких процессоров. Примеры: Linux, Windows.
4. ОС персональных компьютеров. Операционные системы для повседневного использования. Являются универсальными. Примеры: Linux, FreeBSD, Windows, OS X.

5. ОС для носимых устройств. Также являются универсальными, но оптимизированы для работы на каком-то конкретном оборудовании. Примеры: Android, IOS.

4 Иерархия запоминающих устройств.

Проектные ограничения на конфигурацию памяти компьютера в основном определяются тремя параметрами: объемом, быстродействием, стоимостью.

На любом этапе развития технологий производства запоминающих устройств выполняются следующие, достаточно устойчивые, соотношения.

- Чем меньше время доступа, тем дороже каждый бит.
- Чем выше емкость, тем ниже стоимость бита.
- Чем выше емкость, тем больше время доступа.

Уровни иерархии:

1. Внутренняя память процессора: регистры, кэш
2. Внутреняя память: оперативная память, жёсткие диски
3. Внешняя память: CD-ROM, DVD-RAM, Blu-Ray
4. Автономная память: магнитная лента

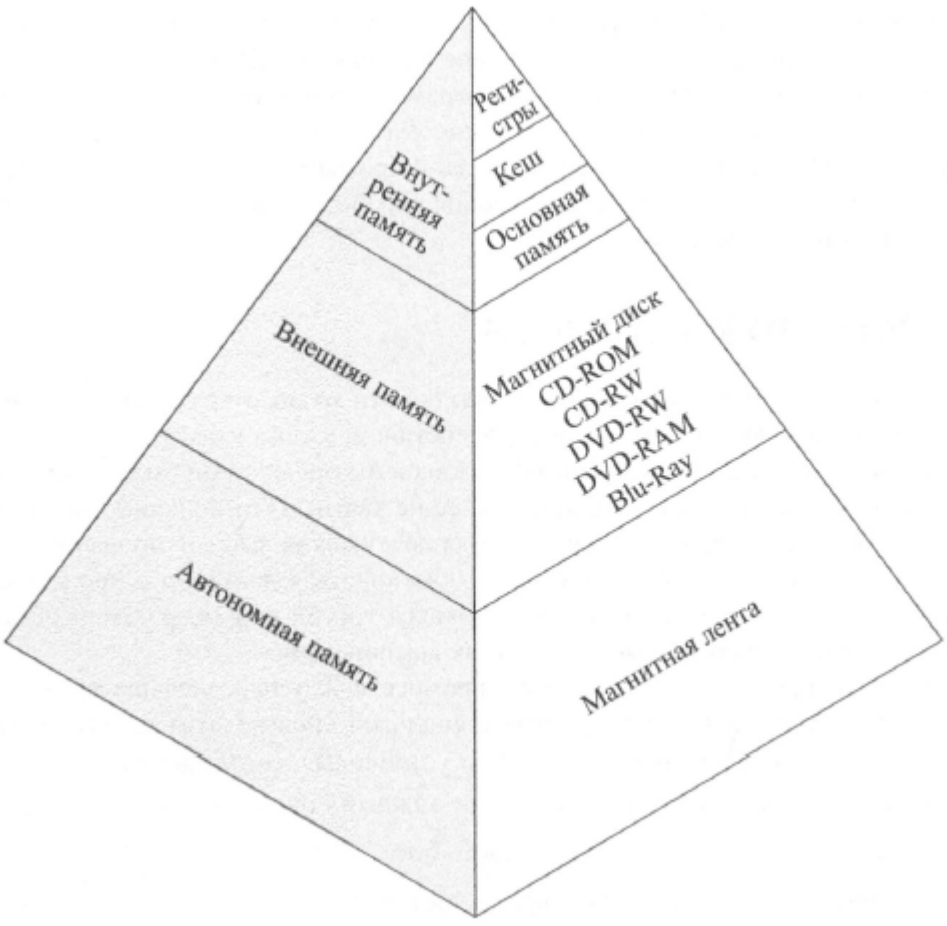

Рис. 2 – Иерархия запоминающих устройств

5 Архитектура ОС. Назначение и функции основных подсистем ОС.

5.1 Структура ОС

Наиболее общим подходом к структуризации операционной системы является разделение всех ее модулей на две группы:

- Ядро - модули, выполняющие основные функции ОС.
- Вспомогательные модули - модули, выполняющие вспомогательные функции ОС.

Рис. 3 – Классическая архитектура системы UNIX

5.2 Ядро

Ядро — центральная часть операционной системы (ОС), обеспечивающая приложениям координированный доступ к ресурсам компьютера, таким как процессорное время, память, внешнее аппаратное обеспечение, внешнее устройство ввода и вывода информации.

Одно из главных свойств ядра - работа в привилегированном режиме.

5.3 Вспомогательные модули

Современные операционные системы имеют сложную структуру, каждый элемент которой выполняет определенные функции по управлению компьютером.

- Управление файловой системой. Процесс работы компьютера сводится к обмену файлами между устройствами. В операционной системе имеются программные модули, управляющие файловой системой.
- Обработчик команд. Специальная программа, которая запрашивает у пользователя команды и выполняет их.
- Драйверы устройств. Специальные программы, которые обеспечивают управление работой устройств и согласование информационного обмена с другими устройствами, а также позволяют производить настройку некоторых параметров устройств. Технология «Plug ad Play» (подключай и играй) позволяет автоматизировать подключение к компьютеру новых устройств и обеспечивает их конфигурирование.
- Графический интерфейс. Используется для упрощения работы пользователя.
- Сервисные программы или утилиты. Программы, позволяющие обслуживать диски (проверять, сжимать, дефрагментировать и т.д.), выполнять операции с файлами (архивировать и т.д.), работать в компьютерных сетях и т.д.
- Справочная система. Позволяет оперативно получить информацию как о функционировании операционной системы в целом, так и о работе ее отдельных модулей.

5.4 Функции ОС

Можно выделить некоторые основные функции, которые выполняет операционная система:

1. Планирование заданий.
2. Использование процессора.
3. Обеспечение программ средствами коммуникации и синхронизации.
4. Управление памятью.
5. Управление файловой системой.
   - преобразует символьные имена файлов, с которыми удобно работать пользователю или программисту, в физические адреса данных на диске;
   - организует совместный доступ к файлам;
   - защищает файлы от несанкционированного доступа.
6. Управление вводом выводом.
7. Обеспечение безопасности.

6 Прерывания. Классы и обработка прерываний.

Прерывания - механизм, с помощью которого различные модули ОС могут прервать нормальную последовательность работы процессора.

Класс прерывания и его описание:

1. Программные - генерируются в случае арифметического переполнения, деления на НОЛЬ, обращения к закрытой области памяти.
2. Таймера - генерируется таймером процессора. Позволяет ОС выполнять функции периодически(т.е. с определенной периодичностью)
3. Ввода-вывода - генерируется контроллером ввода-вывода. Сигнализирует о нормальном завершении операции или о наличии ошибок
4. Аппаратный сбой - генерируется при сбоях и аварийных ситуациях. Например, падение напряжения в сети или ошибка контроля четности памяти

Прерывания предназначены в основном для повышения эффективности использования процессора. Например, большинство устройств ввода-вывода работают намного медленнее, чем процессор. То есть, процессор может не ждать, пока медленное устройство закончит свою работу и даст данные для обработки. Когда данные готовы к обработке сработает прерывание, сигнализирующее процессору, что пора работать.

Также прерывания используются для реализации многозадачности. Процессор в своей структуре имеет таймер, который срабатывает через равные промежутки времени. Если повесить на данное прерывание программу-планировщик. появится возможность чередовать процессы. которые сейчас выполняются на процессоре.

Рис. 4 – Схема выполнения кода с прерываниями

7 Организация многопроцессорных и многоядерных систем.

7.1 Многопроцессорная система

Многопроцессорная архитектура включает в себя два и более центральных процессоров (ЦП), совместно использующих общую память и периферийные устройства, Это увеличивает производительность системы, поскольку появляется возможность одновременно исполнять процессы на разных ЦП.

Каждый ЦП функционирует независимо от других, но все они работают с одним и тем же ядром операционной системы.

Поведение процессов в такой системе ничем не отличается от поведения в однопроцессорной системе.

7.2 Структура многопроцессорной системы

Общая структура МП-системы: связанная архитектура с общей памятью с распределенной обработкой данных и прерываний ввода-вывода. Эта структура полностью симметрична; т. е. все процессоры идентичны и имеют одинаковый статус. При этом каждый процессор может обмениваться данными с каждым другим процессором. Симметричность имеет два важных аспекта: симметричность памяти и ввода-вывода.

- Симметричность памяти - все процессоры совместно используют общее пространство памяти, а также имеют в этом пространстве доступ с одними и теми же адресами.
- Симметричность ввода-вывода - все процессоры имеют возможность доступа к одним и тем же подсистемам ввода-вывода, причем любой процессор может получить прерывание от любого источника.

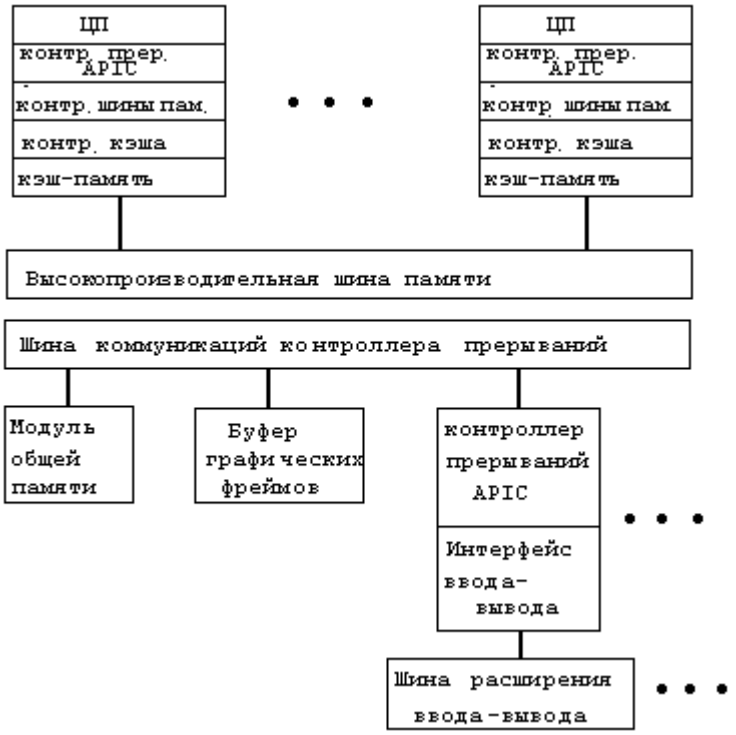

Рис. 5 – Архитектура многопроцессорной системы

7.3 Структура многоядерных системы

Структура многоядерной системы во многом повторяет структуру многопроцессорной, но в меньшем масштабе.

8 Подсистема управления процессами ОС. Состояния процесса.

8.1 Основные понятия

Процесс - это

- выполняемая программа;
- экземпляр программы, выполняющейся на компьютере;
- сущность, которая может быть назначена процессору и выполнена на нем;
- единица активности, характеризуемая выполнением последовательности команд, текущим состоянием и связанным с ней множеством системных ресурсов.

У процесса есть:

- Идентификатор. Уникальный идентификатор, связанный с этим процессом, чтобы отличать его от всех прочих процессов.
- Состояние. Если процесс выполняется в настоящее время, он находится в состоянии выполнения.
- Приоритет. Уровень приоритета по отношению к другим процессам.
- Указатели памяти. Включают указатели на программный код и данные, связанные с этим процессом, а также на любые блоки памяти, совместно используемые с другими процессами.
- Данные контекста. Это данные, присутствующие в регистрах процессора во время выполнения процесса.

8.2 Подсистема управления процессами ОС

Все процессы управляются подсистемой управления процессами ОС.

Основные функции подсистемы управления процессами:

1. создание процессов;
2. обеспечение процессов необходимыми ресурсами;
3. изоляция процессов;
4. планирование выполнения процессов(планирование заданий);
5. организация межпроцессного взаимодействия;
6. синхронизация процессов;
7. завершение и уничтожение процессов.

8.3 Состояния процесса

Состояния процесса:

1. Выполняющийся. Процесс, который выполняется в текущий момент времени. В данном случае предполагается, что на компьютере установлен только один процессор, поэтому в этом состоянии может находиться только один процесс.
2. Готовый к выполнению. Процесс, который может быть запущен, как только для этого представится возможность.

3. Блокированный/Ожидающий. Процесс, который не может выполняться до тех пор, пока не произойдет некоторое событие, например завершение операции ввода-вывода.
4. Новый. Только что созданный процесс, который еще не помещен операционной системой в пул выполнимых процессов. Обычно это новый процесс, который еще не загружен в основную память, хотя управляющий блок процесса уже создан.
5. Завершающийся. Процесс, удаленный операционной системой из пула выполнимых процессов из-за завершения его работы или аварийно прерванный по какой-либо иной причине.

Переходы между состояниями процесса:

- Нулевое состояние → Новый. Для выполнения программы создается новый процесс. Это событие может быть вызвано одной из причин, перечисленных раннее.
- Новый → Готовый. Операционная система переводит процесс из состояния нового в состояние готового к выполнению, когда она будет готова к обработке дополнительных процессов.
- Готовый → Выполняющийся. Когда наступает момент выбора нового процесса для запуска, операционная система выбирает один из готовых для выполнения процессов.
- Выполняющийся → Завершающийся. Если процесс сигнализирует об окончании своей работы или происходит его аварийное завершение, операционная система прекращает его выполнение.
- Выполняющийся → Готовый. Этот переход чаще всего происходит из-за того, что процесс выполняется в течение максимального промежутка времени, отведенного для непрерывной работы одного процесса.
- Выполняющийся → Блокированный. Процесс переводится в заблокированное состояние, если для продолжения работы требуется наступление некоторого события, которое он вынужден ожидать.
- Блокированный → Готовый. Заблокированный процесс переходит в состояние готовности к выполнению в тот момент, когда происходит ожидаемое им событие.
- Готовый → Завершающийся. В некоторых системах родительский процесс может в любой момент прервать выполнение дочернего процесса. Кроме того, дочерние процессы могут прекратиться при завершении родительского процесса.
- Блокированный → Завершающийся.

Рис. 6 – Состояния процесса

9 Описание и управление процессами.

9.1 Задачи управления процессами

К задачам управления процессами относятся:

- Планирование процессов - распределение процессорного времени: что, сколько, и когда выполняется;
- Создание и уничтожение процессов - ОС обеспечивает старты, выделяет ресурсы, обеспечивает уничтожение, освобождение ресурсов и т.д.
- Обеспечение процессов системными ресурсами (памятью, различными устройствами)
- Поддержка взаимодействия между процессами (обеспечение межпроцессорного взаимодействия)

9.2 Структуры описания процесса

Существуют две информационные структуры, по-разному описывающие процессы - контекст процесса и дескриптор процесса.

Контекст процесса

- Контекст процесса содержит информацию о внутреннем состоянии процесса, а также отражает состояние аппаратуры в момент прерывания процесса и включает параметры операционной среды.
- Содержит часть информации необходимой для возобновления выполнения процесса с прерванного места.

В состав контекста процесса входит:

1. содержимое регистров процессора (включает счётчик команд, т.е. на каком этапе находится процесс);
2. размещение кодового сегмента;
3. информацию об открытых данным процессом файлах;

Дескриптор процесса — это информационная структура, которая описывает внешнюю структуру (информацию) процесса. Она нужна планировщику для выполнения процесса, а также нужна ядру в течение всего жизненного цикла процесса

В состав дескриптора входят:

- Идентификатор процесса;
- Состояние процесса.
- Информация о привилегированности процесса.
- Информация о расположении кодового сегмента.

10 Выполнение кода операционной системы.

Операционная система по своей сути работает точно так же, как обычная программа, и она часто передает управление другим программам. Есть два вида работы ядра операционной системы — вне процессов и в их составе.

Ядро вне процессов

- При таком подходе прерывание выполняющегося в данное время процесса или вызов управляющей программы приводит к сохранению контекста данного процесса и передаче управления ядру.
- Операционная система имеет свою собственную область памяти и свой собственный системный стек, который используется для управления вызовами процедур и возвратами из них.
- Операционная система может выполнить все необходимые функции и восстановить контекст процесса, после чего его выполнение будет продолжено.
- После завершения сохранения контекста данного процесса операционная система может также перейти к планированию и диспетчеризации другого процесса.

Выполнение в составе пользовательских процессов

- На небольших машинах часто применяется подход, при котором почти все программы операционной системы выполняются в контексте пользовательского процесса.
- Операционная система - это в основном набор процедур, которые вызываются для выполнения различных функций пользовательского процесса.
- Каждый процесс, принятый операционной системой на обработку, включает в себя также области кода, данных и стека программ ядра.

Код и данные операционной системы находятся в совместно используемом адресном пространстве и доступны для использования всеми пользовательскими процессами.

11 Понятие потока. Различия потока и процесса.

Единицу диспетчеризации обычно называют потоком (thread) или облегченным процессом (lightweight process), а единицу владения ресурсами - процессом (process) или заданием (task).

Под ресурсами имеются в виду ресурсы процесса. Процесс включает виртуальное адресное пространство, в котором содержится образ процесса. Образ процесса представляет собой коллекцию из кода, данных, стека и атрибутов, определенных в управляющем блоке процесса.

Кроме того, термин "облегченнный процесс"используется в трех значениях:

1. Эквивалентен термину поток (thread),
2. Обозначает поток особого вида, известный как поток уровня ядра (kernel-level thread),
3. (в операционной системе Solaris) элемент, отображающий пользовательские потоки на потоки уровня ядра.

Многопоточностью (multithreading) называется способность операционной системы поддерживать в рамках одного процесса несколько параллельных путей выполнения. Традиционный подход, при котором каждый процесс представляет собой единый поток выполнения, называется однопоточным подходом.

11.1 Различия потока и процесса

В рамках процесса могут находиться один или несколько потоков, каждый из которых обладает следующими характеристиками:

1. Состояние выполнения потока (выполняющийся, готовый к выполнению и т.д.).
2. Сохраненный контекст не выполняющегося потока; один из способов рассмотрения потока - считать его независимым счетчиком команд, работающим в рамках процесса.
3. Стек выполнения.
4. Статическая память, выделяемая потоку для локальных переменных.
5. Доступ к памяти и ресурсам процесса, которому этот поток принадлежит; этот доступ разделяется всеми потоками данного процесса.

То есть, процесс — самостоятельный объект исполнения операционной системы, у которого есть собственная памяти. Поток же называют "облегчённым процессов"по той причине, что он использует память процесса-родителя.

12 Алгоритмы управления процессами. Алгоритмы, основанные на квантовании.

Алгоритмы, основанные на квантовании времени, заключаются в том, что любому процессу на выполнение отводится определенный квант времени (несколько миллисекунд).

Переключение активного процесса происходит в случае, если:

- Истек срок времени, выделенного на выполнение процесса.
- Процесс завершился.
- Процесс перешел в состояние ожидания.

По истечении выделенного времени планировщик ставит другой процесс. Если до истечения времени процесс находится в режиме ожидания, запускается другой процесс.

Кванты времени, выделенные процессам, могут быть для разных процессов одинаковыми или различными. Кванты, выделяемые одному процессу, могут быть фиксированной величины, а могут и изменяться в разные периоды жизни процесса.

Некоторые из процессов используют полученные кванты времени не полностью из-за необходимости выполнить операции ввода-вывода. Тогда возникает ситуация, когда процессы с интенсивными обращениями к вводу-выводу используют только небольшую часть выделенного им процессорного времени. В качестве компенсации за не полностью использованные кванты процессы получают привилегии при последующем обслуживании (создают две очереди готовых процессов, прежде всего просматривается вторая очередь; если она пуста, квант выделяется процессу из первой очереди)

Выбор новых процессов может быть построен по принципам:

- FIFO (очередь).
- LIFO (стек).

13 Алгоритмы управления процессами. Алгоритмы, основанные на приоритетах.

13.1 Приоритеты процессов

Приоритет – число, характеризующее степень привилегированности процесса.

В каждой ОС это число трактуется по своему, оно может быть фиксированным или изменяться.

Чем выше приоритет, тем выше привилегия, тем меньше времени проводит поток в очереди.

Существует 2 разновидности таких алгоритмов:

1. Использующие относительные приоритеты.
2. Использующие абсолютные приоритеты.

Алгоритмы планирования с относительными приоритетами – активный процесс выполняется пока не завершится или не перейдет в состояние ожидания.

Алгоритмы планирования с абсолютными приоритетами – смена процесса происходит в тот момент, когда в системе появляется процесс, приоритет которого выше приоритета выполняемого процесса.

Реально используются смешанные схемы планирования.

13.2 Приоритеты Linux

В Linux используется шкала приоритетов от 0 до 139. При это она разделена на две части

1. 0-99 — это системные приоритеты.
2. 100-139 — приоритеты уровня пользователя. Также данные приоритеты называют термином nice.

Значение nice процесса может принимать значения от -20 (наивысший приоритет) до +19 (наименьший приоритет). По умолчанию приоритет равен 0, то есть, значение 120 общей шкалы приоритетов.

14 Типы процедур планирования процессов.

Существует два основных типа процедур планирования процессов - вытесняющие и невытесняющие.

- Невытесняющая многозадачность - это способ планирования процессов, при котором активный процесс выполняется до тех пор, пока он сам, по собственной инициативе, не отдаст управление планировщику операционной системы для того, чтобы тот выбрал из очереди другой, готовый к выполнению процесс.
- Вытесняющая многозадачность - это такой способ, при котором решение о переключении процессора с выполнения одного процесса на выполнение другого процесса принимается планировщиком операционной системы, а не самой активной задачей.

Основным различием между вытесняющей и невытесняющей вариантами многозадачности является степень централизации механизма планирования задач.

- При вытесняющей многозадачности механизм планирования задач целиком сосредоточен в операционной системе, и программист пишет свое приложение, не заботясь о том, что оно будет выполняться параллельно с другими задачами. При этом операционная система выполняет следующие функции: определяет момент снятия с выполнения активной задачи, запоминает ее контекст, выбирает из очереди готовых задач следующую и запускает ее на выполнение, загружая ее контекст.
- При невытесняющей многозадачности механизм планирования распределен между системой и прикладными программами. Прикладная программа, получив управление от операционной системы, сама определяет момент завершения своей очередной итерации и передает управление ОС с помощью какого-либо системного вызова, а ОС формирует очереди задач и выбирает в соответствии с некоторым алгоритмом (например, с учетом приоритетов) следующую задачу на выполнение. Такой механизм создает проблемы, как для пользователей, так и для разработчиков. Для пользователей это означает, что управление системой теряется на произвольный период времени, который определяется приложением (а не пользователем).

15 Проблема синхронизации процессов и потоков. Эффект «гонок».

15.1 Проблемы многопоточного программирования

Синхронизация - это процесс, при котором параллельно исполняющиеся процессы или потоки выполняют некоторый участок кода последовательно. Синхронизация используется для того, чтобы предотвратить состояние или эффект "гонок"при работе с общими ресурсами(данными). Потоки читают и записывают элемент общих данных и конечный результат зависит от относительного времени выполнения этих потоков. В случае, когда мы пытаемся решить состояние гонки при помощи синхронизации, есть шанс наткнуться на проблему взаимной блокировки.

Взаимоблокировка – ситуация, когда два и более процессов не в состоянии работать, поскольку каждый из процессов ожидает выполнения некоторого действия другим процессом.

Взаимоблокировка также может иметь динамический характер.

Динамическая взаимоблокировка — ситуация, когда два и более процессов постоянно изменяют свои состояния в ответ на изменения в других процессах без выполнения полезной работы.

15.2 Эффект гонок

Эффект гонок - ошибка, возникающая при создании многопоточных приложений. Под данное понятие попадает целый спектр ошибок, но все они являются следствием неконтролируемого доступа к общей памяти. Например, есть некоторая структура из двух целочисленных полей {x=1,y=2}, при этом x по нашей предметной области всегда меньше, чем y. В первом потоке мы решили записать в структуру {x=3,y=4}, а во втором потоке {x=5,y=6}. Однако, вполне может быть, что сначала выполнится операция x=3, потом x=5, потом y=6, и затем y=4. В итоге, структура будет иметь вид {x=5,y=4}. Такого результата явно никто не ожидал. К тому же, он нарушает правила предметной области.

Для избежания подобных ошибок используется синхронизация.

16 Взаимоисключения: программный подход.

16.1 Взаимоисключение

Взаимоисключение — требование, чтобы, когда один процесс находится в критическом участке, который получает доступ к общим ресурсам, никакой другой процесс не мог находиться в критическом участке, который обращается к любому из этих общих ресурсов. То есть, в программе существует участок кода, который одновременно может выполнять только один процесс. Такой участок также называется "критическая секция".

Программный подход может быть реализован для параллельных процессов, которые выполняются как в однопроцессорной, так и в многопроцессорной системе с общей основной памятью.

Обычно такие подходы предполагают элементарные взаимоисключения на уровне доступа к памяти. То есть одновременный доступ (чтение и/или запись) к одной и той же ячейке основной памяти упорядочивается при помощи некоторого механизма (хотя при этом порядок предоставления доступа не определяется порядком обращения процессов за доступом к памяти).

Никакой иной поддержки со стороны аппаратного обеспечения, операционной системы или языка программирования не предполагается.

Алгоритмы для реализации программного взаимоисключения:

- Алгоритм Деккера
- Алгоритм Петерсона

16.2 Алгоритм Деккера

У нас должна быть возможность следить за состоянием обоих процессов, что обеспечивается массивом flag. Мы должны навязать определенный порядок действий двум процессам, чтобы избежать проблемы "взаимной вежливости". С этой целью можно использовать переменную turn. В нашем случае эта переменная указывает, какой из процессов имеет право на вход в критический участок.

```
boolean flag[2];
int turn;
void P0()
{
    while(true)
    {
        flag[0] = true;
        while(flag[1])
            if (turn == 1)
            {
                flag[0] = false;
                while(turn == 1) /* Ничего не делать */;
                flag[0] = true;
            }
        /* Критический участок */;
        turn = 1;
        flag[0] = false;
        /* Остальной код */;
    }
}
```

```
void P1()
{
    while(true)
    {
        flag[1] = true;
        while(flag[0])
            if (turn == 0)
            {
                flag[1] = false;
                while(turn == 0) /* Ничего не делать
                flag[1] = true;
            }
        /* Критический участок */;
        turn = 0;
        flag[1] = false;
        /* Остальной код */;
    }
}

void main()
{
    flag[0] = false;
    flag[1] = false;
    turn = 1;
    parbegin(P0,P1);
}
```

Рис. 7 – Алгоритм Деккера

16.3 Алгоритм Петерсона

Алгоритм Деккера решает задачу взаимных исключений, но достаточно сложным путем, корректность которого не так легко доказать. Петерсон предложил простое и элегантное решение. Как и ранее, глобальная переменная flag указывает положение каждого процесса по отношению к взаимоисключению, а глобальная переменная turn разрешает конфликты одновременности.

```
boolean flag [2];
int turn;
void P0()
{
    while (true)
    {
        flag [0] = true;
        turn = 1;
        while (flag [1] && turn == 1) /* Ничего не делать */;
        /* Критический раздел */;
        flag [0] = false;
        /* Остальной код */;
    }
}
void P1()
{
    while (true)
    {
        flag [1] = true;
        turn = 0;
        while (flag [0] && turn == 0) /* Ничего не делать */;
        /* Критический раздел */;
        flag [1] = false;
        /* Остальной код */
    }
}
```

```
void main()
{
    flag [0] = false;
    flag [1] = false;
    parbegin(P0, P1);
}
```

Рис. 8 – Алгоритм Петерсона

1. P1 не намерен входить в критический участок. Такой случай невозможен, поскольку при этом выполнялось бы условие flag[1] = false.
2. P1 ожидает входа в критический участок. Такой случай также невозможен, поскольку если turn = 1, то P1 способен войти в критический участок.
3. P1 циклически использует критический участок, монополизировав доступ к нему. Этого не может произойти, поскольку P1 вынужден перед каждой попыткой входа в критический участок дать возможность входа процессу P0, устанавливая значение turn равным 0.

17 Организация взаимоисключения процессов и потоков. Критическая секция. Блокирующая переменная.

17.1 Взаимоисключение

Взаимоисключение — требование, чтобы, когда один процесс находится в критическом участке, который получает доступ к общим ресурсам, никакой другой процесс не мог находиться в критическом участке, который обращается к любому из этих общих ресурсов. То есть, в программе существует участок кода, который одновременно может выполнять только один процесс. Такой участок также называется "критическая секция".

17.2 Критическая секция

Критическая секция — объект синхронизации потоков, позволяющий предотвратить одновременное выполнение некоторого набора операций несколькими потоками. Критическая секция выполняет те же задачи, что и мьютекс.

Мьютекс — объект ядра, над которым можно произвести 2 операции: захватить и освободить. Обе операции атомарны, то есть, только один поток может захватить и освободить мьютекс. Когда процесс захватывает мьютекс, состояние мьютекса меняется так, что другой процесс, который попытается захватить мьютекс, уснёт. Однако, он проснётся после того, как первый процесс освободит мьютекс.

Между мьютексом и критической секцией есть терминологические различия: так процедура, аналогичная «захвату» мьютекса, называется входом в критическую секцию, а аналогичная снятию блокировки мьютекса — выходом из критической секции.

Процедура входа и выхода из критических секций обычно занимает меньшее время, нежели аналогичные операции мьютекса, что связано с отсутствием необходимости обращаться к ядру ОС.

В операционных системах семейства Microsoft Windows разница между мьютексом и критической секцией в том, что мьютекс является объектом ядра и может быть использован несколькими процессами одновременно, критическая секция же принадлежит процессу и служит для синхронизации только его потоков.

17.3 Блокирующие переменные

Для синхронизации процессов могут также использоваться глобальные блокирующие переменные.

С этими переменными, к которым все процессы имеют прямой доступ, можно работать, не обращаясь к системным вызовам ОС. Суть их такова:

- Каждому набору критических данных ставится в соответствие двоичная переменная, которой процесс присваивает значение 0, когда он входит в критическую секцию, и значение 1, когда он ее покидает.
- Если переменная установлена в 0, то данные заняты и запрашивающий процесс продолжает запрос.
- Если же данные свободны, то значение переменной устанавливается в 0 и процесс входит в критическую секцию.
- После того как он выполнит все действия с данными, значение переменной снова устанавливается равным 1.

18 Организация взаимоисключения процессов и потоков. События.

18.1 Взаимоисключение

Взаимоисключение — требование, чтобы, когда один процесс находится в критическом участке, который получает доступ к общим ресурсам, никакой другой процесс не мог находиться в критическом участке, который обращается к любому из этих общих ресурсов. То есть, в программе существует участок кода, который одновременно может выполнять только один процесс. Такой участок также называется "критическая секция".

В общем случае для реализации взаимоисключения используются мьютексы. Мьютекс — объект ядра, над которым можно произвести 2 операции: захватить и освободить. Обе операции атомарны, то есть, только один поток может захватить и освободить мьютекс. Когда процесс захватывает мьютекс, состояние мьютекса меняется так, что другой процесс, который попытается захватить мьютекс, уснёт. Однако, он проснётся после того, как первый процесс освободит мьютекс.

18.2 События

Флаги событий - слово памяти, используемое как механизм синхронизации. Код приложения может связать с каждым битом флага свое событие. Поток может ждать либо одного события, либо сочетания событий путем проверки одного или нескольких битов в соответствующем флаге. Поток блокируется до тех пор, пока все необходимые биты не будут установлены (И) или пока не будет установлен хотя бы ОДИН из битов (ИЛИ).

19 Организация взаимоисключения процессов и потоков. Семафор.

19.1 Взаимоисключение

Взаимоисключение — требование, чтобы, когда один процесс находится в критическом участке, который получает доступ к общим ресурсам, никакой другой процесс не мог находиться в критическом участке, который обращается к любому из этих общих ресурсов. То есть, в программе существует участок кода, который одновременно может выполнять только один процесс. Такой участок также называется "критическая секция".

19.2 Семафор

Семафор - целочисленное значение, используемое для передачи сигналов между процессами.

Семафор может быть инициализирован неотрицательным целочисленным значением.

Над семафором могут быть выполнены только три операции (все они являются атомарными):

1. инициализация,
2. уменьшение (декремент) значения,
3. увеличение (инкремент) значения

Операция уменьшения может привести к блокировке процесса, а операция увеличения - к разблокированию. Известен также как семафор со счетчиком или обобщенный семафор.

В начале работы семафор имеет значение нуль или некоторое положительное значение.

- Если значение положительное, то оно равно количеству процессов, которые могут вызвать операцию получения сигнала и немедленно продолжить выполнение.
- Если же значение равно нулю (полученное либо при инициализации, либо потому, что количество процессов, равное первоначальному значению семафора, вызвало операцию ожидания), очередной ожидающий процесс блокируется, а значение семафора становится отрицательным.

Каждое последующее ожидание уменьшает значение семафора, так что оно имеет отрицательное значение, по модулю равное числу процессов, ожидающих разблокирования.

Когда значение семафора отрицательное, каждый сигнал разблокирует один из ожидающих процессов.

Существует также и бинарный семафор - это семафор, который может принимать только два значения - 0 и 1.

Бинарный семафор может быть инициализирован значением 0 или 1.

- Операция semWait проверяет значение семафора. Если это значение нулевое, процесс, выполняющий semWait, блокируется. Если значение равно 1, оно изменяется, становясь равным 0, и выполнение процесса продолжается.
- Операция semSignal проверяет, имеется ли процесс, блокированный этим семафором (значение семафора равно 0). Если есть, то процесс, блокированный операцией semWait, разблокируется. Если заблокированных процессов нет, значение семафора устанавливается равным 1.

20 Организация взаимоисключения процессов и потоков. Монитор.

20.1 Взаимоисключение

Взаимоисключение — требование, чтобы, когда один процесс находится в критическом участке, который получает доступ к общим ресурсам, никакой другой процесс не мог находиться в критическом участке, который обращается к любому из этих общих ресурсов. То есть, в программе существует участок кода, который одновременно может выполнять только один процесс. Такой участок также называется "критическая секция".

20.2 Монитор

Монитор — представляет собой конструкцию языка программирования, которая обеспечивает функциональность, эквивалентную функциональности семафоров, но более легкую в управлении. Мониторы также реализуются как программные библиотеки. Это позволяет использовать мониторы, блокирующие любые объекты. В частности, например, для связанного списка можно заблокировать все связанные списки одной блокировкой либо иметь отдельные блокировки для каждого списка, а возможно - и для каждого элемента списка.

Монитор представляет собой программный модуль, состоящий из инициализирующей последовательности, одной или нескольких процедур и локальных данных. **Основные характеристики:**

1. Локальные переменные монитора доступны только его процедурам; внешние процедуры доступа к локальным данным монитора не имеют.
2. Процесс входит в монитор путем вызова одной из его процедур.
3. В мониторе в определенный момент времени может выполняться только один процесс; любой другой процесс, вызвавший монитор, будет приостановлен в ожидании доступности монитора.

Для широкого применения в параллельных вычислениях мониторы должны включать инструменты синхронизации. Предположим, например, что процесс вызывает монитор и, находясь в мониторе, должен быть приостановлен до выполнения некоторого условия. При этом нам требуется некий механизм, который не только приостанавливает процесс, но и освобождает монитор, позволяя войти в него другому процессу. Позже, когда условие окажется выполненным, а монитор доступным, приостановленный процесс сможет продолжить свою работу с того места, где он был приостановлен.

Монитор поддерживает синхронизацию при помощи условных переменных, содержащихся в мониторе и доступных только в нем. Работать с этими переменными могут две функции.

- **cwait (c)**. Приостанавливает выполнение вызывающего процесса по условию c. Монитор при этом доступен для использования другим процессом.
- **csignal (c)**. Возобновляет выполнение некоторого процесса, приостановленного вызовом cwait с тем же условием. Если имеется несколько таких процессов, выбирается один из них; если таких процессов нет, функция не делает ничего.

Обратите внимание на то, что операции **wait/signal** монитора отличаются от соответствующих операций семафора. Если процесс в мониторе передает сигнал, но при этом нет ни одного ожидающего его процесса, то сигнал просто теряется.

20.3 Разница между монитором и семафором

Отличие Монитора от Семафора:

- Основное различие между семафором и монитором состоит в том, что семафор является целочисленной переменной S, которая указывает на количество ресурсов, доступных в системе, тогда как монитор это абстрактный тип данных, который позволяет одновременно выполнять только один процесс в критической секции.
- Значение семафора может быть изменено на wait(), а также signal() операция. С другой стороны, монитор имеет общие переменные и процедуры, с помощью которых процессы могут получить доступ к общим переменным.
- В семафоре, когда процесс хочет получить доступ к общим ресурсам, процесс выполняет wait() и блокирует ресурсы, а когда освобождает ресурсы, выполняет signal() операцию. В мониторах, когда процессу требуется доступ к общим ресурсам, он должен получать к ним доступ с помощью процедур в мониторе.
- Тип монитора имеет переменные условия, каких семафор не имеет.

Мониторы проще в реализации, чем семафор, и по сравнению с семафорами вероятность ошибки в мониторе ниже.

21 Понятие тупика. Проблема тупиков.

Под потоком имеется в виду не объект ядра, а последовательность команд. То есть, нить выполнения, а не частный случай процесса

Тупик может возникнуть в ситуации, когда два или более параллельно выполняемых потока конкурируют за обладание двумя или более общими ресурсами. При клинче каждый из потоков успевает захватить один из общих ресурсов. Для окончания работы каждому потоку необходимы другие ресурсы, захваченные другими потоками. В результате, никто из потоков не может завершить свою работу, все стоят в очередях, которые не двигаются, - работа замирает - приложение "зависает". Это худшее, что может случиться с приложением.

Тупиковые ситуации надо отличать от простых очередей, хотя те и другие возникают при совместном использовании ресурсов и внешне выглядят похоже: поток приостанавливается и ждет освобождения ресурса. Однако очередь — это нормальное явление, неотъемлемый признак высокого коэффициента использования ресурсов при случайном поступлении запросов. Очередь появляется тогда, когда ресурс недоступен в данный момент, но освободится через некоторое время, позволив потоку продолжить выполнение. Тупик же, что видно из его названия, является в некотором роде неразрешимой ситуацией. Необходимым условием возникновения тупика является потребность потока сразу в нескольких ресурсах.

Невозможность потоков завершить начатую работу из-за возникновения взаимных блокировок снижает производительность вычислительной системы. Поэтому проблеме предотвращения тупиков уделяется большое внимание. На тот случай, когда взаимная блокировка все же возникает, система должна предоставить администратору-оператору средства, с помощью которых он смог бы распознать тупик, отличить его от обычной блокировки из-за временной недоступности ресурсов. И наконец, если тупик диагностирован, то нужны средства для снятия взаимных блокировок и восстановления нормального вычислительного процесса.

22 Принципы взаимного блокирования.

Взаимное блокирование (deadlock) можно определить как перманентное блокирование множества процессов, которые либо конкурируют в борьбе за системные ресурсы, либо сообщаются один с другим. Множество процессов оказывается взаимно блокированным, если каждый процесс множества заблокирован в ожидании события (обычно - освобождения некоторого запрашиваемого ресурса), которое может быть вызвано только другим блокированным процессом множества. В отличие от других проблем, возникающих в процессе управления параллельными вычислениями, данная проблема в общем случае эффективного решения не имеет.

- Автомобилю 1, движущемуся на север, нужны а и б.
- Автомобилю 2, движущемуся на запад, нужны б и в.
- Автомобилю 3, движущемуся на юг, нужны в и г.
- Автомобилю 4, движущемуся на восток, нужны г и а.

Обычное правило пересечения перекрестка состоит в том, что автомобиль должен уступить дорогу движущемуся справа. Это правило работает, когда перекресток пересекают два или три автомобиля. Но если перекресток пересекают одновременно четыре автомобиля, каждый из которых согласно правилу воздержится от въезда на перекресток, возникнет взаимоблокировка.

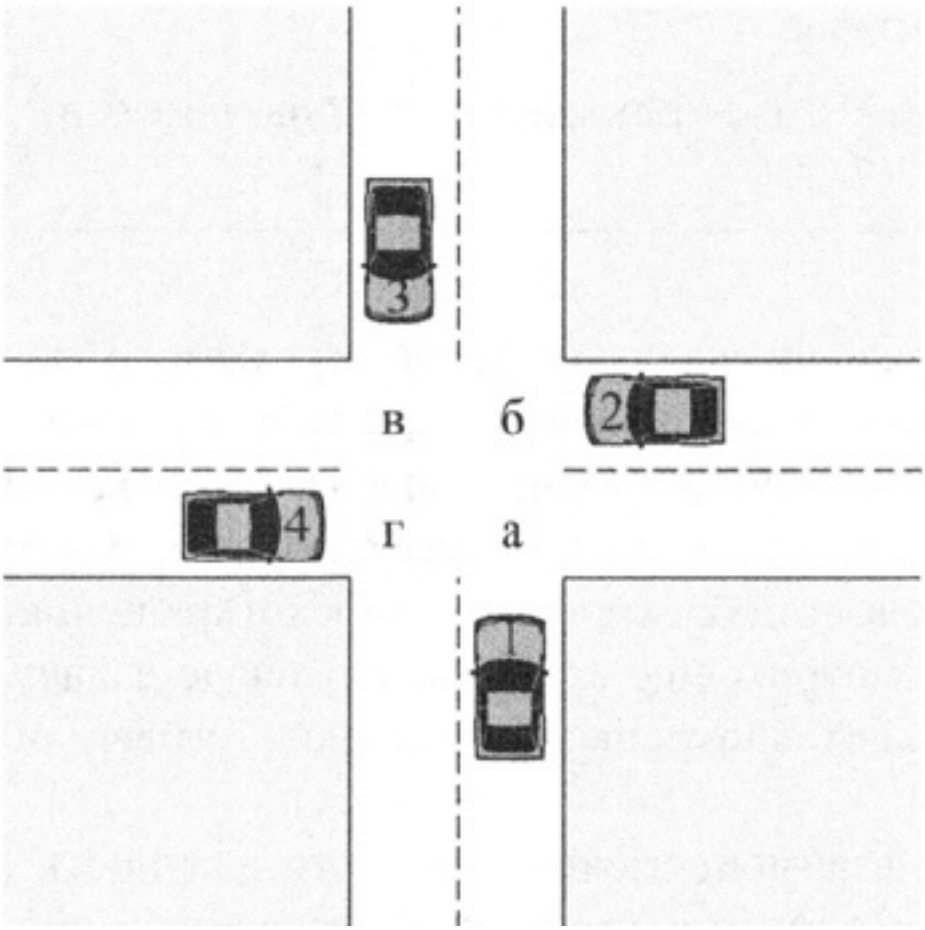

Рис. 9

Она возникнет и в том случае, если все четыре машины проигнорируют правило и осторожно въедут на перекресток, поскольку при этом каждый автомобиль захватит один ресурс (один квадрант) и останется на вечной стоянке в ожидании, когда другой автомобиль освободит следующий требующийся для пересечения перекрестка квадрант. Итак, мы опять получили взаимоблокировку.

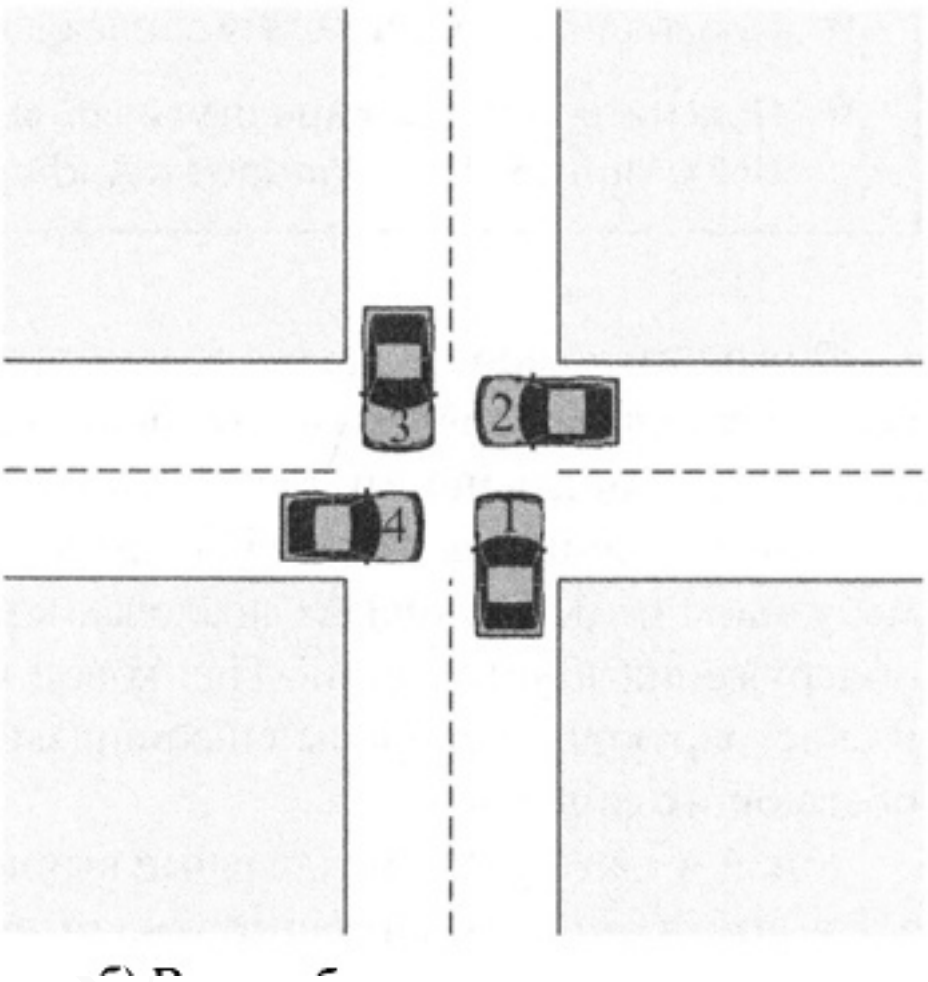

Рис. 10

23 Предотвращение и устранение взаимоблокировок.

23.1 Условия возникновения взаимных блокировок

Условия возникновения взаимных блокировок:

1. **Взаимные исключения**. Одновременно использовать ресурс может только один процесс.
2. **Удержание и ожидание**. Процесс может удерживать выделенные ресурсы во время ожидания других ресурсов.
3. **Отсутствие перераспределения**. Ресурс не может быть принудительно отобран у удерживающего его процесса. Эти условия выполняются довольно часто. Например, взаимоисключения необходимы для гарантии согласованности результатов и целостности базы данных. Аналогично невозможно произвольное применение перераспределения, в особенности при работе с данными, когда требуется обеспечить механизм отката. Для реального осуществления взаимоблокировки, требуется выполнение четвертого условия.
4. **Циклическое ожидание**. Существует замкнутая цепь процессов, каждый из которых удерживает как минимум один ресурс, необходимый процессу, следующему в цепи после данного.

23.2 Восстановление

После того как взаимоблокировка обнаружена, требуется некоторая стратегия для восстановления нормальной работоспособности системы. Вот несколько возможных подходов к решению этой проблемы, перечисленные в порядке возрастания сложности.

- **Прекратить выполнение всех заблокированных процессов**. Самый распространенный подходов.
- **Вернуть каждый из заблокированных процессов в некоторую ранее определенную точку и перезапустить все процессы**. Для этого в систему должны быть встроены механизмы отката и перезапуска. Самый большой риск при таком подходе заключается в том, что взаимоблокировка может проявиться вновь. Однако неопределенность относительных скоростей выполнения параллельных вычислений обычно позволяет этого избежать.
- **Последовательно прекращать выполнение заблокированных процессов по одному до тех пор, пока взаимоблокировка не прекратится**. Порядок выбора уничтожаемых процессов должен базироваться на некотором критерии минимальной стоимости. После каждого уничтожения процесса должен быть вызван алгоритм обнаружения взаимоблокировок для проверки, не устранены ли они.
- **Последовательно перераспределять ресурсы до тех пор, пока взаимоблокировка не прекратится**. Как и в предыдущем случае, выбор процесса должен осуществляться в соответствии с некоторым критерием минимальной стоимости, а после осуществления перераспределения должен вызываться алгоритм обнаружения взаимоблокировок. Процесс, ресурсы которого перераспределяются, должен быть возвращен к состоянию, в котором он находился до получения этого ресурса.

23.3 Интегрированные стратегии разрешения взаимоблокировок

У каждой стратегии разрешения взаимоблокировок есть свои преимущества и недостатки, а потому наиболее эффективным путем может оказаться применение разных подходов в различных ситуациях. Предлагается следующий подход к данной проблеме.

- Сгруппировать ресурсы в несколько различных классов.
- Для предотвращения циклического ожидания во избежание взаимоблокировок между классами ресурсов использовать описанный ранее метод линейного упорядочения типов ресурсов.
- В пределах одного класса ресурсов использовать наиболее подходящий для данного типа ресурсов алгоритм.

Классы ресурсов:

- **Пространство подкачки**. (Блоки памяти на вторичных устройствах хранения информации, используемые при свопинге процессов.) Предотвращение взаимоблокировок с помощью требования, чтобы все ресурсы распределялись одновременно. Такая стратегия вполне применима, если известны максимальные требования (что зачастую выполняется на практике). Можно также использовать стратегию устранения взаимоблокировок.
- **Ресурсы процесса**(Назначаемые устройства, такие как стримеры или файлы.) В этой категории ресурсов зачастую наиболее эффективным является использование стратегии устранения взаимоблокировок, поскольку можно ожидать, что процесс заранее объявит о требуемых ему ресурсах этого типа. Кроме того, можно использовать предотвращение взаимоблокировок путем упорядочения ресурсов в пределах данного класса.
- **Основная память**(Страницы или сегменты, назначаемые процессу.) Пожалуй, наиболее подходящим методом предотвращения взаимоблокировок в этом случае может оказаться перераспределение ресурсов. Процесс, ресурсы которого перераспределяются, просто сбрасывается на вторичные устройства хранения информации, освобождая основную память для разрешения взаимоблокировки.
- **Внутренние ресурсы**(Такие ресурсы, как, например, каналы ввода-вывода.)Можно использовать предотвращение взаимоблокировок путем упорядочения ресурсов в пределах данного класса.

24 Обнаружение взаимоблокировок

(было бы неплохо дополнить, но гуглится плохо. Есть в Таненбауме глава 6.4)

Тупиковые ситуации надо отличать от простых очередей, хотя те и другие возникают при совместном использовании ресурсов и внешне выглядят похоже: поток приостанавливается и ждет освобождения ресурса. Однако очередь — это нормальное явление, неотъемлемый признак высокого коэффициента использования ресурсов при случайном поступлении запросов. Очередь появляется тогда, когда ресурс недоступен в данный момент, но освободится через некоторое время, позволив потоку продолжить выполнение. Тупик же, что видно из его названия, является в некотором роде неразрешимой ситуацией. Необходимым условием возникновения тупика является потребность потока сразу в нескольких ресурсах.

Невозможность потоков завершить начатую работу из-за возникновения взаимных блокировок снижает производительность вычислительной системы. Поэтому проблеме предотвращения тупиков уделяется большое внимание. На тот случай, когда взаимная блокировка все же возникает, система должна предоставить администратору-оператору средства, с помощью которых он смог бы распознать тупик, отличить его от обычной блокировки из-за временной недоступности ресурсов. И наконец, если тупик диагностирован, то нужны средства для снятия взаимных блокировок и восстановления нормального вычислительного процесса.

Простейшим инструментом для обнаружения взаимоблокировок является менеджер процессов (top, htop в linux). В утилитах данного рода можно наблюдать за использованием ресурсов отдельного потока. Мы можем сравнить фактические значения с ожидаемыми и сделать вывод о том, правильное ли поведение у системы в данный момент.

25 Задача об обедающих философах.

25.1 Формулировка

Итак, в некотором царстве, в некотором государстве жили вместе пять философов. Жизнь каждого из них проходила в основном в размышлениях, прерываемых приемом пищи. Философы давно сошлись во мнении, что только спагетти в состоянии восстанавливать их подточенные непрерывными размышлениями силы. Питались они за одним круглым столом, на который помещались большое блюдо со спагетти, пять тарелок, по одной для каждого философа, и пять вилок. Проголодавшийся философ садится на свое место за столом и, пользуясь двумя вилками, приступает к еде. Задача состоит в том, чтобы разработать ритуал (алгоритм) обеда, который обеспечивает взаимоисключения (два философа не могут одновременно пользоваться одной вилкой) и не допускает взаимоблокировок и голодания (обратите внимание, насколько уместным оказался этот термин в данной задаче!).

25.2 Не придумывайте себе проблем

Чтобы избежать риска взаимоблокировки, можно купить еще пять вилок (кстати, самое подходящее решение задачи с точки зрения гигиены!) или научить философов есть спагетти одной вилкой. Еще один подход состоит в том, чтобы нанять вышибалу, который не позволит пяти философам садиться за стол одновременно. Если же за столом соберутся не более четырех философов, то по крайней мере один из них сможет воспользоваться двумя вилками.

25.3 Вариант с семафорами:

```
semaphore fork[5] = {1};
semaphore room = {4};
int i;
void philosopher(int i) {
while (true){
think();
wait(fork[i]);
wait(fork [(i + 1) mod 5]);
eat();
signal(fork[ (i + 1) mod 5]);
signal(fork[i]);
signal(room)}
}
void main() {
        parbegin(
        philosopher(0),
        philosopher(1),
        philosopher(2),
        philosopher(3),
        philosopher(4)
        );
 }
```

Философ берёт две вилки и кушает. Когда покушал - кладёт вилки и пробуждает соседних философов, сам при этом засыпает. Пробуждённые философы проверяют, хватает ли у них вилок. Если философ проверил и вилки у него 2, он начинает кушать. После трапеза кладёт вилки на стол и пробуждает соседей. И так по кругу.

25.4 Вариант с мьютексом

(не лучший метод, потому что вилок хватает, чтобы одновременно кушали несколько философов. А в этом решении всегда максимум сможет покушать один, а остальные сдохнут

с голоду) В лекциях описана реализация с помощью семафора, но забавный факт: это мой вариант по второй лабе ОС, именно поэтому, как по мне там проще использовать мьютекс. Типа только один чел может воспользоваться ресурсом (то есть вилкой), они передают их покругу, и короче, когда одному из этих мужичков попадается две вилки (какая удача), то она блокируется для всех остальных и радостный мужичок сидит хавает свою пасту, а другие ждут и голодают, кайф супер, мне понравилось.

26 Задача читатель/писатель.

Предположим, у нас есть хранилище данных, с которым работают несколько потоков. Потоки могут выполнять операции чтения и записи данных.

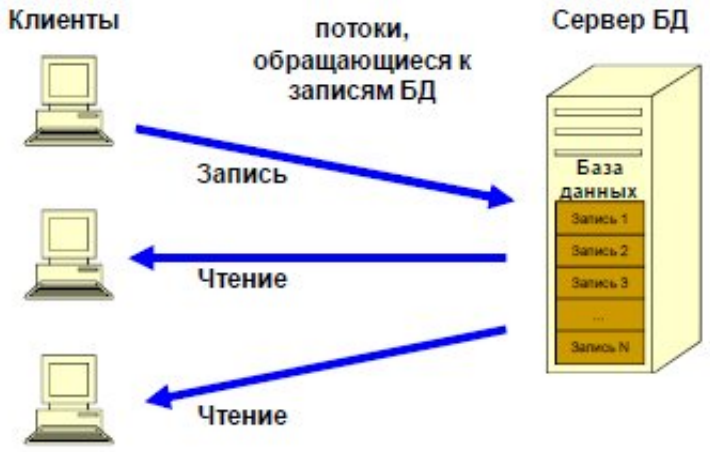

Рис. 11 – Читатель

Несколько потоков, обращающихся к базе данных Функциональность потоков читателя и писателя можно записать следующим образом.

```
Writer:
while(true) {
PrepareData();
Write(Data);
}
Reader:
while(true) {
Read(&Data);
UseData();
}
```

Если несколько потоков одновременно обращаются к разным записям базы данных, то никаких проблем не возникает. Проблема синхронизации доступа к данным возникает при обращении нескольких потоков к одной и той записи. К тому же, организация согласованного доступа для потоков-читателей и потоков писателей может отличаться, поскольку одновременное выполнение операции чтения несколькими потоками не приводит к возникновению проблем, в то время как операция записи требует эксклюзивного доступа.

Задача "Читатели-Писатели"заключается в обеспечении согласованного доступа нескольких потоков к разделяемым данным. Корректное решение должно удовлетворять следующим условиям:

- потоки выполняются параллельно;
- во время выполнения потоком операции записи, данные не могут использоваться другими потоками;
- во время выполнения потоком операции чтения, другие потоки также могут выполнять операцию чтения;
- потоки должны завершить работу в течение конечного времени.

Для защиты разделяемых критических данных используется двоичный семафор Access. Функция писателя просто получает доступ к критическим данным (уменьшая семафор Access) и использует их.

Функция читателя должна обеспечить одновременный доступ к критическим данным нескольких читателей, поэтому она использует дополнительную переменную ReadCount – количество читателей, выполняющих операцию чтения в настоящий момент. Данная переменная также требует защиты для корректного использования несколькими читателями (для этого введен двоичный семафор RC), но ее использование позволяет выполнять запрос на доступ к критическим данным только первому читателю (остальные будут заблокированы при попытке уменьшить семафор RC), а освобождение данных оставить последнему читателю, завершившему операцию чтения. Обратим внимание, что вследствие подобного подхода функция читателя содержит 3 критических секции: одну, связанную с чтением критических данных, и две, связанных с изменением переменной ReadCount и организации процесса входа и выхода из первой критической секции.

Отметим также, что если потоки-читатели постоянно входят в критическую секцию, возможно голодание (starvation) потоков-писателей. (Голодание – ситуация, когда поток не может приступить к выполнению своей задачи в течение неограниченного периода времени). Для решения этой проблемы можно использовать еще один семафор, с помощью которого организовать запрет доступа новых читателей в критическую секцию при наличии ожидающих писателей.

27 Задача производитель/потребитель.

27.1 Формулировка

Одной из типовых задач, требующих синхронизации, является задача producer-consumer (производитель-потребитель). Пусть два потока обмениваются информацией через буфер(канал) ограниченного размера. Производитель добавляет информацию в буфер, а потребитель извлекает ее оттуда.

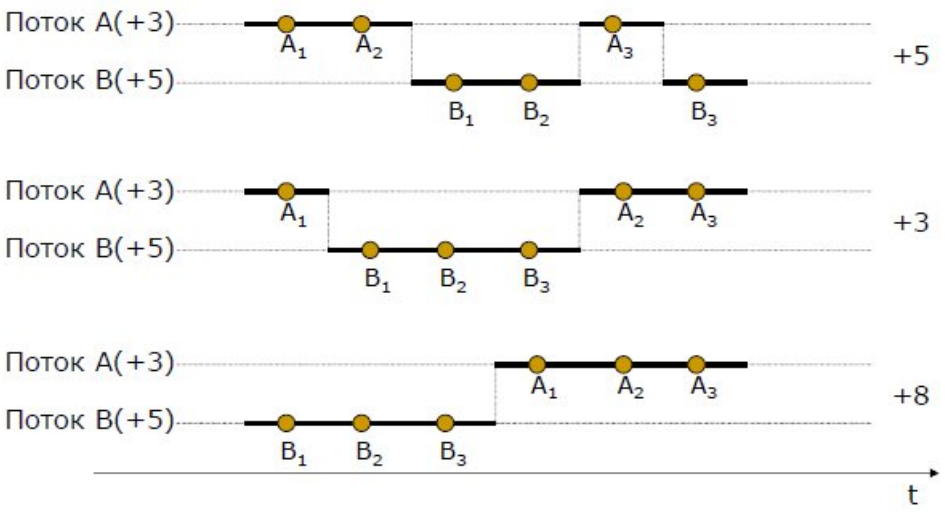

Рис. 12 – Производитель

Два потока, обменивающихся информацией через циклический буфер Функциональность потоков производителя и потребителя можно записать следующим образом.

```
Producer:
while(true) {
PrepareData();
Put(Data);
}
Consumer:
while(true) {
Get(&Data);
UseData();
}
```

Сразу необходимо отметить, что если буфер пуст, то потребитель должен ждать, пока в нем появятся данные, а если буфер полон, то производитель должен ждать появления свободного элемента. В данном случае реализация производителя и потребителя будет иметь следующий вид.

```
Producer:
while(true) {
PrepareData();
while( ! Put(Data) )
;
}
Consumer:
while(true) {
while( ! Get(&Data) )
;
UseData();
}
```

27.2 Решение

Задача "Производители-Потребители"заключается в обеспечении согласованного доступа нескольких потоков к разделяемому циклическому буферу. Корректное решение должно удовлетворять следующим условиям:

- потоки выполняются параллельно;
- одновременно в критической секции, связанной с каждым критическим ресурсом, должно находиться не более одного потока;
- потоки должны завершить работу в течение конечного времени;
- потоки должны корректно использовать операции с циклическим буфером.

Решение данной задачи должно, во-первых, обеспечивать согласованный доступ к критическим данным, во-вторых, оповещение потребителей, ожидающих поступления данных, и производителей, ожидающих появления незаполненных записей в буфере.

27.3 Реализация

```
#define N 100 /* Количество мест в буфере */
typedef int semaphore; /* Семафоры - это специальная разновидность
 целочисленной переменной */
semaphore mutex = 1; /* управляет доступом к критической области */
semaphore empty = N; /* подсчитывает пустые места в буфере */
semaphore full = 0; /* подсчитывает занятые места в буфере */
void producer(void)
{
 int item;
 while (TRUE) { /* TRUE - константа, равная 1 */
 item = produce_item( ); /* генерация чего-нибудь для помещения в
 буфер */
 down(&empty); /* уменьшение счетчика пустых мест */
 down(&mutex); /* вход в критическую область */
 insert_item(item); /* помещение новой записи в буфер */
 up(&mutex); /* покинуть критическую область */
 up(&full); /* инкремент счетчика занятых мест */
 }
}
void consumer(void)
{
 int item;
 while (TRUE) { /* бесконечный цикл */
 down(&full); /* уменьшение счетчика занятых мест */
 down(&mutex); /* вход в критическую область */
 item = remove_item( ); /* извлечение записи из буфера */
 up(&mutex); /* выход из критической области */
 up(&empty); /* увеличение счетчика пустых мест */
 consume_item(item); /* работа с записью */
 }
}
```

В этой реализации используются три семафора: один из них называется full и предназначен для подсчета количества заполненных мест в буфере, другой называется empty и предназначен для подсчета количества пустых мест в буфере, третий называется mutex, он предотвращает одновременный доступ к буферу производителя и потреби теля. Семафор full изна-

чально равен 0, семафор empty изначально равен количеству мест в буфере, семафор mutex изначально равен 1

28 Подсистема коммуникации. Базовые примитивы передачи сообщений.

28.1 Примитивы:

Работа этих механизмов не требует внесения изменений в записи, соответствующие высокоуровневым программам в очереди диспетчера, поскольку остальные программы могут продолжить свое выполнение лишь тогда, когда работа примитивов полностью завершена.

Основой этого взаимодействия может служить только передача по сети сообщений. В самом простом случае системные средства обеспечения связи могут быть сведены к двум основным системным вызовам (примитивам), один - для посылки сообщения, другой - для получения сообщения. В дальнейшем на их базе могут быть построены более мощные средства сетевых коммуникаций, такие как распределенная файловая система или вызов удаленных процедур, которые, в свою очередь, также могут служить основой для построения других сетевых сервисов.

В частности, эффективность коммуникаций в сети зависит от способа задания адреса, от того, является ли системный вызов блокирующим или неблокирующим, какие выбраны способы буферизации сообщений и насколько надежным является протокол обмена сообщениями.

28.2 Пример

В централизованных системах связь между процессами, как правило, предполагает наличие разделяемой памяти. Типичный пример - проблема "поставщик-потребитель в этом случае один процесс пишет в разделяемый буфер, а другой - читает из него. Даже наиболее простая форма синхронизации - семафор - требует, чтобы хотя бы одно слово (переменная самого семафора) было разделяемым. В распределенных системах нет какой бы то ни было разделяемой памяти. Основой этого взаимодействия может служить только передача по сети сообщений. В самом простом случае системные средства обеспечения связи могут быть сведены к двум основным системным вызовам (для посылки сообщения и для получения). В дальнейшем на их основе могут быть построены более мощные средства сетевых коммуникаций, такие как распределенная файловая система или вызов удаленных процедур.

Несмотря на простоту этих системных вызовов - ПОСЛАТЬ и ПОЛУЧИТЬ - существуют различные варианты их реализации, от правильного выбора которых зависит эффективность работы сети. В частности, эффективность коммуникаций в сети зависит от способа задания адреса, от того, является ли системный вызов блокирующим или неблокирующим, какие выбраны способы буферизации сообщений и насколько надежным является протокол обмена сообщениями.

29 Подсистема коммуникации. Блокирующие и неблокирующие примитивы.

29.1 Блокирующие и неблокирующие примитивы.

При использовании **блокирующего** примитива, процесс, выдавший запрос на его выполнение, приостанавливается до полного завершения примитива. При использовании **неблокирующего** примитива управление возвращается вызывающему процессу немедленно, еще до того, как требуемая работа будет выполнена. Преимуществом этой схемы является параллельное выполнение вызывающего процесса и процесса передачи сообщения. Однако выигрыш в производительности при использовании неблокирующих примитивов компенсируется серьезным недостатком: отправитель не может модифицировать буфер сообщения, пока сообщение не отправлено, а узнать, отправлено ли сообщение, отправитель не может. Первое решение - это заставить ядро копировать сообщение в свой внутренний буфер, а затем разрешить процессу продолжить выполнение. Второе решение заключается в прерывании процесса-отправителя после отправки сообщения, чтобы проинформировать его, что буфер снова доступен.

30 Подсистема коммуникации. Буферизируемые и небуферизируемые примитивы передачи сообщений.

30.1 Небуферизируемый

Небуферизируемый вызов ПОЛУЧИТЬ сообщает ядру машины, на которой он выполняется, адрес буфера, в который следует поместить пребывающее для него сообщение.

Эта схема работает тогда, когда вызов ПОСЛАТЬ сделан раньше ПОЛУЧИТЬ. Один из вариантов - просто отказаться от сообщения. Второй подход к этой проблеме заключается в том, чтобы хранить хотя бы некоторое время, поступающие сообщения в ядре получателя на тот случай, что вскоре будет выполнен соответствующий вызов ПОЛУЧИТЬ. Каждый раз, когда поступает такое "неожидаемое"сообщение, включается таймер. Если заданный временной интервал истекает раньше, чем происходит соответствующий вызов ПОЛУЧИТЬ, то сообщение теряется. Концептуально простым способом управления буферами является определение новой структуры данных, называемой почтовым ящиком.

30.2 Буферизированный

Процесс, который заинтересован в получении сообщений, обращается к ядру с запросом о создании для него почтового ящика и сообщает адрес, после все сообщения с данным адресом будут в ящике. Такой способ часто называют **буферизуемым** примитивом.

Можно сказать, что средство связи является буферизованным, если посланное сообщение дойдет до адресата даже если посылающий процесс прекратил существование или адресат еще не существовал (или не был активным) в момент посылки сообщения. Должен существовать буферный объект, который сохраняет сообщение между посылкой и отправкой.

Буферизация сообщений имеет отношение еще к одному термину: сохранность. Эти термины имеют много общего и в ряде применений эквивалентны. Термин буферизация показывает больше техническую сторону вопроса, термин сохранность — логическую сторону. Буферизация, т. е. сохранение сообщений в буферном объекте, в распределенных системах является просто одним из способов обеспечения сохранности сообщений. Обеспечивать сохранность сообщений можно и без использования буферизованных средств связи.

31 Подсистема коммуникации. Надежные и ненадежные примитивы передачи сообщений.

31.1 Проблема надёжности соединения

Ранее подразумевалось, что когда отправитель посылает сообщение, адресат его обязательно получает. Но на практике сообщения могут теряться. Предположим, что для обмена сообщениями используются блокирующие примитивы. Когда отправитель посылает сообщение, то он приостанавливает свою работу до тех пор, пока сообщение не будет послано. Однако нет никаких гарантий, что после того, как он возобновит свою работу, сообщение будет доставлено адресату.

31.2 Подходы к решению

Для решения этой проблемы существуют **три подхода**.

1. **Первый** заключается в том, что система не берет на себя никаких обязательств по поводу доставки сообщений. Такой способ доставки сообщений обычно называют **дейтаграммным (datagram)**. Реализация надежного взаимодействия при его применении целиком становится заботой прикладного программиста.
2. **Второй** подход заключается в том, что ядро принимающей машины посылает квитанцию-подтверждение ядру отправляющей машины на каждое сообщение или на группу последовательных сообщений. Посылающее ядро разблокирует пользовательский процесс только после получения такого подтверждения. Обработкой подтверждений занимается подсистема обмена сообщениями ОС, ни процесс-отправитель, ни процесс-получатель их не видят.
3. **Третий** подход заключается в использовании ответа в качестве подтверждения в тех системах, в которых запрос всегда сопровождается ответом, что характерно для клиент-серверных служб. В этом случае служебные сообщения-подтверждения не используются, так как в их роли выступают пользовательские сообщения-ответы. Процесс-отправитель остается заблокированным до получения ответа. Если же ответа нет слишком долго, то после истечения тайм-аута ОС отправителя повторно посылает запрос.

Надежная передача сообщений может подразумевать не только гарантию доставки отдельных сообщений, но и упорядоченность этих сообщений, при которой процесс-получатель извлекает из системного буфера сообщения в том же порядке, в котором они были отправлены. Для надежной и упорядоченной доставки чаще всего используется обмен с предварительным установлением соединения, причем на стадии установления соединения (называемого также сеансом) стороны обмениваются начальными номерами сообщений, чтобы можно было в процессе обмена отслеживать как факт доставки отдельных сообщений последовательности, так и упорядочивать их (сами сетевые технологии не всегда гарантируют, что порядок доставки сообщений будет совпадать с порядком их отправки, например из-за того, что разные сообщения могут доставляться адресату по разным маршрутам).

В хорошей подсистеме обмена сообщения должны поддерживаться как ненадежные примитивы, так и надежные. Это позволяет прикладному программисту использовать тот тип примитивов, который в наибольшей степени подходит для организации взаимодействия в той или иной ситуации. Например, для передачи данных большого объема, транспортируемых по сети в нескольких сообщениях (в сетях обычно существует ограничение на максимальный размер поля данных, из-за чего данные приходится пересылать в нескольких сообщениях), больше подходит надежный вид обмена с упорядочиванием сообщений. А вот для взаимодействия типа «короткий запрос — короткий ответ» предпочтительны ненадежные примитивы.

32 Алгоритмы взаимного исключения. Распределенный алгоритмы.

32.1 Понятие распределённого алгоритма

Распределенный алгоритм есть алгоритм, предназначенный для работы на компьютерном оборудовании, состоящем из взаимосвязанных процессоров. Распределенные алгоритмы используются во многих разнообразных областях применения распределенных вычислений, таких как телекоммуникации, научные вычисления, распределенная обработка информации и реальные-время управление процессом. Стандартные задачи, решаемые распределенными алгоритмами, включают выборы лидера, консенсус, распределенный поиск , создание остовного дерева, взаимное исключение и распределение ресурсов .

Распределенные алгоритмы являются подтипом параллельного алгоритма , обычно выполняемого одновременно, при этом отдельные части алгоритма запускаются одновременно на независимых процессорах и имея ограниченную информацию о том, что делают другие части алгоритма. Одна из основных проблем при разработке и реализации распределенных алгоритмов — это успешная координация поведения независимых частей алгоритма перед лицом сбоев процессора и ненадежных каналов связи. Выбор подходящего распределенного алгоритма для решения данной проблемы зависит как от характеристик проблемы, так и от характеристик системы, в которой будет работать алгоритм, таких как тип и вероятность сбоев процессора или канала, вид межпроцессного взаимодействия. которые могут быть выполнены, и уровень синхронизации времени между отдельными процессами.

32.2 Принцип работы

Когда процесс хочет войти в критическую секцию, он формирует сообщение, содержащее:

- имя нужной ему критической секции;
- номер процесса;
- текущее значение времени.

Затем он посылает это сообщение всем другим процессам.

Предполагается, что передача сообщения надежна, то есть получение каждого сообщения сопровождается подтверждением. Когда процесс получает сообщение такого рода, его действия зависят от того, в каком состоянии по отношению к указанной в сообщении критической секции он находится. Имеют место три ситуации:

1. Если получатель не находится и не собирается входить в критическую секцию в данный момент, то он отсылает назад процессу-отправителю сообщение с разрешением.
2. Если получатель уже находится в критической секции, то он не отправляет никакого ответа, а ставит запрос в очередь.
3. Если получатель хочет войти в критическую секцию, но еще не сделал этого, то он сравнивает временную отметку поступившего сообщения со значением времени, которое содержится в его собственном сообщении, разосланном всем другим процессам.Если время в поступившем к нему сообщении меньше, то есть его собственный запрос возник позже, то он посылает сообщение-разрешение, в обратном случае он не посылает ничего и ставит поступившее сообщение-запрос в очередь.

Процесс может войти в критическую секцию только в том случае, если он получил ответные сообщения-разрешения от всех остальных процессов. Когда процесс покидает критическую секцию, он посылает разрешение всем процессам из своей очереди и исключает их из очереди.

33 Алгоритмы взаимного исключения. Алгоритм Token Ring.

33.1 Определение

Token Ring — протокол передачи данных в локальной вычислительной сети (LAN) с топологией кольца и «маркерным доступом».

33.2 Принцип работы

Все процессы системы образуют логическое кольцо, т.е. каждый процесс знает номер своей позиции в кольце, а также номер ближайшего к нему следующего процесса.

1. Когда кольцо инициализируется, процессу 0 передается так называемый токен.
2. Токен циркулирует по кольцу. Он переходит от процесса n к процессу n+1 путем передачи сообщения по типу "точка-точка".
3. Когда процесс получает токен от своего соседа, он анализирует, не требуется ли ему самому войти в критическую секцию. Если да, то процесс входит в критическую секцию. После того, как процесс выйдет из критической секции, он передает токен дальше по кольцу.
4. Если же процесс, принявший токен от своего соседа, не заинтересован во вхождении в критическую секцию, то он сразу отправляет токен в кольцо.

Следовательно, если ни один из процессов не желает входить в критическую секцию, то в этом случае токен просто циркулирует по кольцу с высокой скоростью

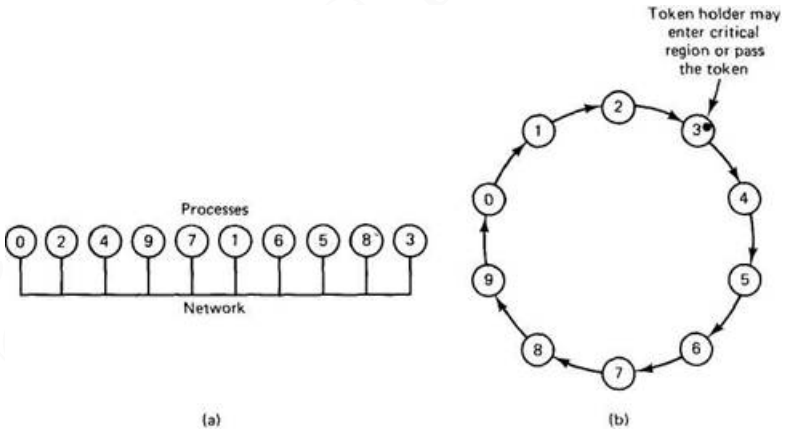

Рис. 13 – Token Ring

34 Алгоритмы взаимного исключения. Централизованный алгоритм.

Наиболее очевидный и простой путь реализации взаимного исключения в распределенных системах - это применение тех же методов, которые используются в однопроцессорных системах.

1. Один из процессов выбирается в качестве координатора(например, процесс, выполняющийся на машине, имеющей наибольшее значение сетевого адреса).
2. Если в этот момент ни один из процессов не находится в критической секции, то координатор посылает ответ с разрешением.
3. Если же некоторый процесс уже выполняет критическую секцию, связанную с данным ресурсом, то никакой ответ не посылается; запрашивавший процесс ставится в очередь, и после освобождения критической секции ему отправляется ответ-разрешение.

Этот алгоритм гарантирует взаимное исключение, но вследствие своей централизованной природы обладает низкой отказоустойчивостью.

Централизованные vs рапределенные алгоритмы

Централилизованные:

- основная часть выполняется одним процессом, остальные процессы играют незначительную роль в достижении общего результата;
- обычно это — получение от основного процесса или предоставление ему нужных данных;
- ассиметричный: различные процессы исполняются логически разные операции;
- модель клиент-сервер.

Распределенные алгоритмы:

- каждый процесс играет одинаковую роль в достижении общего результата и разделении общей нагрузки;
- симметричный: все процессы исполняют один и тот же код (логические функции).

35 Способы адресации в распределенных системах.

35.1 Способы адресации.

- Физические адреса. Одним из вариантов адресации на верхнем уровне является использование физических адресов сетевых адаптеров. Если в получающем компьютере выполняется только один процесс, то ядро будет знать, что делать с поступившим сообщением — передать его этому процессу.
- Адрес машины-процесса. Альтернативная адресная система использует имена назначения, состоящие из двух частей, определяющих номер машины и номер процесса. Однако адресация типа «машина-процесс» далека от идеала, в частности не гибка и не прозрачна, так как пользователь должен явно задавать адрес машины-получателя. В этом случае, если в один прекрасный день машина, на которой работает сервер, отказывает, то программа, в которой жестко используется адрес сервера, не сможет работать с другим сервером, установленном на другой машине.
- Централизованный счётчик. Другим вариантом могло бы быть назначение каждому процессу уникального адреса, который никак не связан с адресом машины. Одним из способов достижения этой цели является использование централизованного механизма распределения адресов процессов, который работает просто, как счетчик. При получении запроса на выделение адреса он просто возвращает текущее значение счетчика, а затем наращивает его на единицу.
- Личные адреса. Совершенно иной подход - это использование специальной аппаратуры. Пусть процессы выбирают свои адреса случайно, а конструкция сетевых адаптеров позволяет хранить эти адреса. Теперь адреса процессов не обнаруживаются путем широковещательной передачи, а непосредственно указываются в кадрах, заменяя там адреса сетевых адаптеров.
- Символьные имена. В сети выделяют специальную машину для отображения высокоуровневых символьных имен. При применении такой системы процессы адресуются с помощью символьных строк, и в программы вставляются эти строки, а не номера машин или процессов. Каждый раз перед первой попыткой связаться, процесс должен послать запрос специальному отображающему процессу, обычно называемому сервером имен, запрашивая номер машины, на которой работает процесс-получатель.

36 Алгоритм синхронизации логических часов.

36.1 Синхронизация

Синхронизация необходима процессам для организации совместного использования ресурсов, таких как файлы или устройства, а также для обмена данными.

Синхронизация процессов — приведение двух или нескольких процессов к такому их протеканию, когда определённые стадии разных процессов совершаются в определённом порядке, либо одновременно.

36.2 Проблема

В распределенных системах процессы могут выполняются на разных машинах, что приводит к невозможности использования стандартных средств синхронизации.

В распределенной системе, где каждый процессор имеет собственные часы со своей точностью хода, программы, использующие время становятся зависимыми от того, часами какого компьютера они пользуются.

Синхронизация физических часов(показывающих реальное время) является сложной проблемой, однако процессам не нужно, чтобы во всех машинах было правильное время, для них важно, чтобы оно было везде одинаковое.

Для некоторых процессов важен только правильный порядок событий. В этом случае необходима **синхронизация логических часов**.

36.3 Решение

Каждый поток поддерживает очередь запросов на вход в критическую секцию. Приоритет – <временная метка, номер потока> (т.е при равенстве временных меток берем тот поток, чей номер меньше, иначе возможна взаимная блокировка, если процессы будут общаться идеально симметрично).

Когда поток хочет войти в критическую секцию, он:

1. Добавляет свой запрос в свою очередь (т.е временную метку и номер потока)
2. Посылает всем потокам запрос (req)
3. Ждет от них ответа (ok)
4. Получив все ответы, ждет, когда он станет первым в своей очереди, и входит в критическую секцию
5. Выйдя из критической секции, удаляет свой запрос из своей очереди, посылает всем сообщение о том, что вышел (rel)

Рис. 14 – Логические часы

37 Задачи подсистемы управления памятью ОС.

37.1 Задача управления памятью

В многозадачных системах «пользовательская» часть памяти должна быть распределена для размещения нескольких процессов. Эта задача распределения выполняется операционной системой динамически и известна под названием управление памятью.

Главной операцией управления памятью является размещение программы в основной памяти для ее выполнения процессором. Практически во всех современных многозадачных системах эта задача предполагает использование сложной схемы, известной как виртуальная память. Виртуальная память, в свою очередь, основана на использовании одной или обеих базовых технологий - сегментации и страничной организации памяти. Перед тем как перейти к рассмотрению этих методов организации виртуальной памяти, мы должны познакомиться с более простыми методами, не связанными с виртуальной памятью

Эффективное управление памятью жизненно важно для многозадачных систем. Если в памяти располагается только небольшое число процессов, то большую часть времени все эти процессы будут находиться в состоянии ожидания выполнения операций ввода-вывода, и загрузка процессора будет низкой. Таким образом, желательно эффективное распределение памяти, позволяющее разместить в ней как можно больше процессов.

37.2 Требования к механизму управления памятью

При рассмотрении различных механизмов и стратегий, связанных с управлением памятью, полезно помнить требования, которым они должны удовлетворять. Эти требования включают следующее.

- Перемещение — программа может быть перемещена из одной области в другую (например, при свопинге). То есть, нельзя заранее знать, где в памяти будет находиться программа.
- Защита — каждый процесс должен быть защищен от нежелательного воздействия других процессов, случайного или преднамеренного. Следовательно, код других процессов не должен иметь возможности без разрешения обращаться к памяти данного процесса для чтения или записи. Данная проверка происходит аппаратно, на уровне процессора.
- Совместное использование — например, если несколько процессов выполняют один и тот же машинный код, то будет выгодно позволить каждому процессу работать с одной и той же копией этого кода, а не создавать собственную. Процессам, сотрудничающим в работе над некоторой задачей, может потребоваться совместный доступ к одним и тем же структурам данных. Система управления памятью должна, таким образом, обеспечивать управляемый доступ к разделяемым областям памяти, при этом никоим образом не ослабляя защиту памяти.
- Логическая организация — практически всегда основная память в компьютерной системе организована как линейное (одномерное) адресное пространство, состоящее из последовательности байтов или слов. Аналогично организована и вторичная память на своем физическом уровне. Хотя такая организация и отражает особенности используемого аппаратного обеспечения, она не соответствует способу, которым обычно создаются программы. Большинство программ организованы в виде модулей, одни из которых неизменны (только для чтения, только для выполнения), а другие содержат данные, которые могут быть изменены.
- Физическая организация — память компьютера разделяется как минимум на два уровня: основная и вторичная. Основная память обеспечивает быстрый доступ по относительно высокой цене; кроме того, она энергозависима, т.е. не обеспечивает долговременное хранение. Вторичная память медленнее и дешевле основной и обычно энерго-

независима. Следовательно, вторичная память большой емкости может служить для долговременного хранения программ и данных, а основная память меньшей емкости – для хранения программ и данных, использующихся в текущий момент.

38 Методы загрузки программ в память. Виды адресов.

38.1 Методы загрузки программ в память

Технология	Описание	Сильные стороны	Слабые стороны
Фиксированное распределение	Основная память разделяется на ряд статических разделов во время генерации системы. Процесс может быть загружен в раздел равного или большего размера	Простота реализации, малые системные накладные расходы	Неэффективное использование памяти из-за внутренней фрагментации фиксированное максимальное количество активны процессов
Динамическое распределение	Разделы создаются динамически; каждый процесс загружается в раздел строго необходимого размера	Отсутствует внутренняя фрагментация, более эффективное использование основной памяти	Неэффективное использование процессора из-за необходимости уплотнения для противодействия внешней фрагментации
Простая страничная организация	Основная память разделена на ряд кадров равного размера. Каждый процесс разделен на некоторое количество страниц равного размера одной и той же длины, что и кадры. Процесс загружается путем загрузки всех его страниц в доступные кадры	Отсутствует внешняя фрагментация	Наличие небольшой внутренней фрагментации
Простая сегментация	Каждый процесс распределен на ряд сегментов. Процесс загружается путем загрузки всех своих сегментов в динамические (не обязательно смежные) разделы	Отсутствует внутренняя фрагментация; по сравнению с динамическим распределением повышенная эффективность использования памяти	Внешняя фрагментация

Рис. 15 – Методы загрузки программы в память

Страничная организация виртуальной памяти	Все, как при простой страничной организации, с тем исключением, что не требуется одновременно загружать все страницы процесса. Необходимые нерезидентные страницы автоматически загружаются в память	Нет внешней фрагментации; более высокая степень многозадачности; большое виртуальное адресное пространство	Накладные расходы из-за сложности системы управления памятью
Сегментация виртуальной памяти	Все, как при простой сегментации, с тем исключением, что не требуется одновременно загружать все сегменты процесса. Необходимые нерезидентные сегменты автоматически загружаются в память	Нет внутренней фрагментации, более высокая степень многозадачности; большое виртуальное адресное пространство; поддержка защиты и совместного использования	Накладные расходы из-за сложности системы управления память

Рис. 16 – Методы загрузки программы в память

38.2 Виды адресов

38.2.1 Логический адрес

Логический адрес представляет собой ссылку на ячейку памяти, не зависящую от текущего расположения данных в памяти; перед тем как получить доступ к этой ячейке памяти, необходимо транслировать логический адрес в физический.

38.2.2 Относительный адрес

Относительный адрес представляет собой частный случай логического адреса, когда адрес определяется положением относительно некоторой известной точки (обычно - начала программы).

38.2.3 Физический адрес

Физический адрес (известный также как абсолютный) представляет собой действительное расположение интересующей нас ячейки основной памяти.

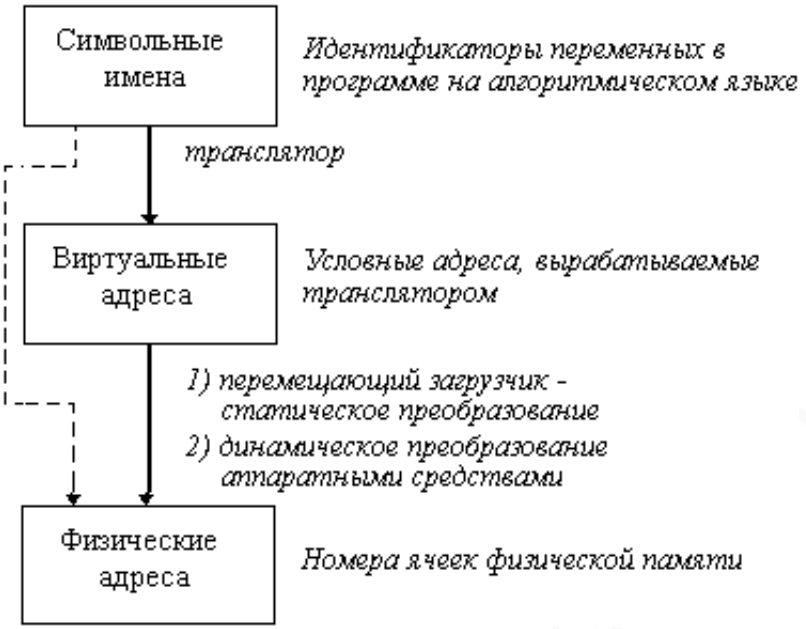

Рис. 17 – Схема работы с разными видами адресов

39 Алгоритмы распределения памяти без использования дискового пространства. Распределение памяти фиксированными разделами.

Главной операцией управления памятью является размещение программы в основной памяти для ее выполнения процессором. Существует два класса методов распределения памяти: без использования дискового пространства и с его использованием. Практически во всех современных многозадачных системах эта задача предполагает использование сложной схемы, известной как виртуальная память.

К алгоритмам распределения без использования дискового пространства относятся:

- Фиксированное распределение
- Разделение переменной величины
- Перемещение

Фиксированное распределение — В большинстве схем управления памятью мы будем полагать, что операционная система занимает некоторую фиксированную часть основной памяти и что остальная часть основной памяти доступна для использования многочисленным процессам. Простейшая схема управления этой доступной памятью - ее распределение на области с фиксированными границами.

Описание: Основная память разделяется на ряд статических разделов во время генерации системы. Процесс может быть загружен в раздел равного или большего размера

Сильные стороны:

Простота в реализации, малые системные накладные расходы.

Слабые стороны:

Неэффективное использование памяти из-за внутренней фрагментации, фиксированное максимальное количество активных процессов.

Размеры разделов:

На рисунке показаны два варианта фиксированного распределения. Одна возможность состоит в использовании разделов одинакового размера. В этом случае любой процесс, размер которого не превышает размер раздела, может быть загружен в любой доступный раздел. Если все разделы заняты и нет ни одного процесса в состоянии готовности или работы, операционная система может выгрузить процесс из любого раздела и загрузить другой процесс, обеспечивая тем самым процессор работой.

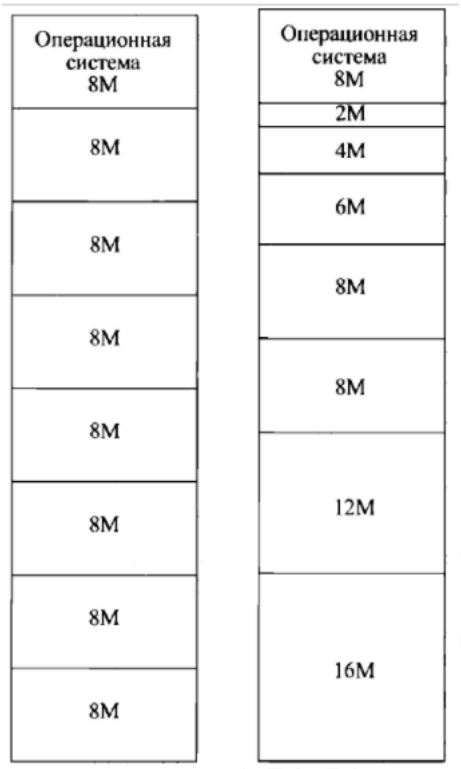

Рис. 18

При использовании разделов с одинаковым размером имеются две трудности.

- Программа может быть слишком велика для размещения в разделе. В этом случае нужно разрабатывать программу, использующую оверлеи, с тем чтобы в любой момент времени ей требовался только один раздел основной памяти. Когда требуется модуль, который в настоящий момент отсутствует в основной памяти, пользовательская программа должна сама загрузить этот модуль в раздел памяти программы (независимо от того, является ли этот модуль кодом или данными).
- Использование основной памяти при этом крайне неэффективно. Любая программа, независимо от ее размера, занимает раздел целиком. Так, в нашем примере программа размером менее мегабайта все равно будет занимать целиком раздел в 8 Мбайт; при этом остаются неиспользованными 7 Мбайт блока. Этот феномен появления неиспользованной памяти из-за того, что загружаемый блок по размеру меньше раздела, называется внутренней фрагментацией.

Бороться с этими трудностями (хотя и не устранить полностью) можно посредством использования разделов разных размеров. В этом случае программа размером 16 Мбайт может обойтись без оверлеев, а разделы малого размера позволяют уменьшить внутреннюю фрагментацию при загрузке программ малого размера.

40 Алгоритмы распределения памяти без использования дискового пространства. Распределение памяти разделами переменной величины. Фрагментация.

40.1 Алгоритмы распределения памяти

Главной операцией управления памятью является размещение программы в основной памяти для ее выполнения процессором. Существует два класса методов распределения памяти: без использования дискового пространства и с его использованием. Практически во всех современных многозадачных системах эта задача предполагает использование сложной схемы, известной как виртуальная память.

К алгоритмам распределения без использования дискового пространства относятся:

- Фиксированное распределение
- Разделение переменной величины
- Перемещение

40.2 Распределение памяти переменной величины

Когда разделы имеют разные размеры, есть два возможных подхода к назначению процессов разделам памяти. Простейший путь состоит в том, чтобы каждый процесс размещался в наименьшем разделе, способном полностью вместить данный процесс. В таком случае для каждого раздела требуется очередь планировщика, в которой хранятся выгруженные из памяти процессы, предназначенные для данного раздела памяти. Преимущество такого подхода заключается в том, что процессы могут быть распределены между разделами памяти так, чтобы минимизировать внутреннюю фрагментацию.

Таким образом, более предпочтительным подходом является использование одной очереди для всех процессов. В момент, когда требуется загрузить процесс в основную память, для этого выбирается наименьший доступный раздел, способный вместить данный процесс. Если все разделы заняты, следует принять решение об освобождении одного из них. По-видимому, следует отдать предпочтение процессу, занимающему наименьший раздел, способный вместить загружаемый процесс. Можно учесть и другие факторы, такие как приоритет процесса или его состояние (заблокирован он или активен).

Использование разделов разного размера по сравнению с использованием разделов одинакового размера придает дополнительную гибкость данному методу. Кроме того, схемы с фиксированными разделами относительно просты, предъявляют минимальные требования к операционной системе; накладные расходы работы процессора невелики.

Однако у этих схем имеются серьезные недостатки.

- Количество разделов, определенное в момент генерации системы, ограничивает количество активных (не приостановленных) процессов.
- Поскольку размеры разделов устанавливаются заранее, в момент генерации системы, небольшие процессы приводят к неэффективному использованию памяти. В средах, в которых заранее известны потребности в памяти всех задач, применение описанной схемы может быть оправдано, но в большинстве случаев эффективность этой технологии крайне низка. Фиксированное распределение в настоящее время практически не используется. Примером успешной операционной системы с использованием данной технологии может служить ранняя операционная система IBM для мейнфреймов OS/-MFT

40.3 Фрагментация

Фрагментация — процесс дробления чего-либо на множество мелких разрозненных фрагментов. Проблемы, связанные с данным термином, вызваны тем, что полезная информация занимает не весь предоставленный объём. То есть, эффективность использования памяти снижается. Например, память разбита на блоки по 64кб. То есть, наименьшей единицей объёма в данной системе будет блок 64кб. Однако, можно придумать много вариантов, в которых данные фактически будут занимать гораздо меньше памяти (храним структуру с реальным размером 1кб в таком блоке. Соответственно, 63кб не используются и не могут быть использованы).

Процесс борьбы с фрагментацией называется дефрагментация. Зачастую, данный процесс связан с жертвой производительности в целях более эффективного использования памяти. Такая ситуация хорошо видна в мире текстовых кодировок — UTF8 и UTF16. В UTF16 каждый символ представлен блоком 16 бит (2 байта). То есть, любой символ можно получить по индексу, как в обычном массиве. В UTF8 символ может быть представлен набором байт (от 1 до 4). Данное решение позволяет экономить место, ведь часто используемые символы помещаются в 1 байт. Но создаёт проблемы с доступом по индексу, ведь блоки могут быть разного размера. Для получения i-того символа нужно пройти всю цепочку символов, которые стоят перед искомым.

41 Алгоритмы распределения памяти без использования дискового пространства. Распределение памяти перемещающимися разделами.

41.1 Алгоритмы распределения памяти

Главной операцией управления памятью является размещение программы в основной памяти для ее выполнения процессором. Существует два класса методов распределения памяти: без использования дискового пространства и с его использованием. Практически во всех современных многозадачных системах эта задача предполагает использование сложной схемы, известной как виртуальная память.

К алгоритмам распределения без использования дискового пространства относятся:

- Фиксированное распределение
- Разделение переменной величины
- Перемещение

41.2 Распределение памяти перемещающимися разделами

Одним из методов борьбы с фрагментацией является перемещение всех занятых участков в сторону старших либо в сторону младших адресов, так, чтобы вся свободная память образовывала единую свободную область(Рис. 19). В дополнение к функциям, которые выполняет ОС при распределении памяти переменными разделами, в данном случае она должна еще время от времени копировать содержимое разделов из одного места памяти в другое, корректируя таблицы свободных и занятых областей. Эта процедура называется "сжатием". Сжатие может выполняться либо при каждом завершении задачи, либо только тогда, когда для вновь поступившей задачи нет свободного раздела достаточного размера. В первом случае требуется меньше вычислительной работы при корректировке таблиц, а во втором - реже выполняется процедура сжатия. Так как программы перемещаются по оперативной памяти в ходе своего выполнения, то преобразование адресов из виртуальной формы в физическую должно выполняться динамическим способом.

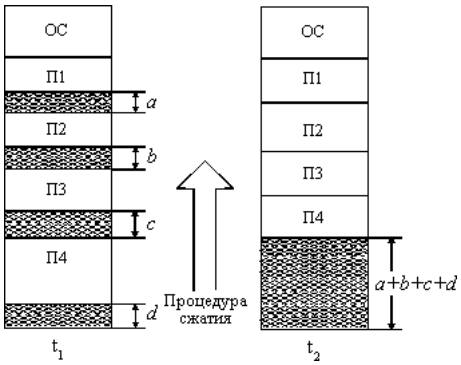

Рис. 19 – Распределение памяти перемещаемыми разделами

Хотя процедура сжатия и приводит к более эффективному использованию памяти, она может потребовать значительного времени, что часто перевешивает преимущества данного метода

42 Алгоритмы распределения памяти с использованием дискового пространства. Страничное распределение памяти.

42.1 Алгоритмы распределения памяти с использованием дискового пространства

Главной операцией управления памятью является размещение программы в основной памяти для ее выполнения процессором. Существует два класса методов распределения памяти: без использования дискового пространства и с его использованием. Практически во всех современных многозадачных системах эта задача предполагает использование сложной схемы, известной как виртуальная память.

К алгоритмам распределения с использования дискового пространства относятся:

- Страничное распределение виртуальной памяти
- Сегментное распределение виртуальной памяти
- Странично-сегментное распределение виртуальной памяти

42.2 Перемещение

Перед тем как мы рассмотрим способы, с помощью которых можно избежать недостатков распределения, следует до конца разобраться в вопросах, связанных с размещением процессов в памяти. При использовании фиксированной схемы распределения можно ожидать, что процесс всегда будет назначаться одному и тому же разделу памяти. Это означает, что какой бы раздел ни был выбран для нового процесса, для размещения этого процесса после выгрузки и последующей загрузки в память всегда будет использоваться именно этот раздел.

Если размеры разделов равны и существует единая очередь процессов для разделов разного размера, процесс по ходу работы может занимать разные разделы. При первом создании образа процесса он загружается в некоторый раздел памяти; позже, после того как он был выгружен из памяти и вновь загружен, процесс может оказаться в другом разделе (не в том, в котором размещался в последний раз). Та же ситуация возможна и при динамическом распределении. Так, к примеру процесс занимает при размещении в памяти различные места. Кроме того, при выполнении уплотнения процессы также перемещаются в основной памяти.

Таким образом, расположение команд и данных, к которым обращается процесс, не является фиксированным и изменяется всякий раз при выгрузке и загрузке (или перемещении) процесса. Для решения этой проблемы следует различать типы адресов.

Логический адрес представляет собой ссылку на ячейку памяти, не зависящую от текущего расположения данных в памяти; перед тем как получить доступ к этой ячейке памяти, необходимо транслировать логический адрес в физический.

Относительный адрес представляет собой частный случай логического адреса, когда адрес определяется положением относительно некоторой известной точки (обычно - начала программы).

Физический адрес (известный также как абсолютный) представляет собой действительное расположение интересующей нас ячейки основной памяти.

42.3 Страничное распределение памяти

Как разделы с разными фиксированными размерами, так и разделы переменного размера недостаточно эффективно используют память. Результатом работы первых становится внутренняя фрагментация, результатом работы последних - внешняя. Предположим, однако, что основная память разделена на одинаковые блоки относительно небольшого фиксированного размера. Тогда блоки процесса, известные как страницы, могут быть связаны со свободными блоками памяти, известными как **кадры или фреймы**. Каждый кадр может содержать одну страницу данных. При такой организации памяти, внешняя фрагментация отсутствует вовсе, а потери из-за внутренней фрагментации ограничены частью последней страницы процесса.

Для поддержания страничного распределения памяти одного регистра базового адреса недостаточно, и для каждого процесса операционная система должна поддерживать таблицу страниц. Таблица страниц указывает расположение кадров каждой страницы процесса. Внутри программы логический адрес состоит из номера страницы и смещения внутри нее. В случае простого распределения логический адрес представляет собой расположение слова относительно начала программы, которое процессор транслирует в физический адрес. При страничной организации преобразование логических адресов в физические также остается задачей аппаратного уровня, решаемой процессором.

Для удобства работы добавим правило, в соответствии с которым размер страницы (а следовательно, и размер кадра) должен представлять собой степень 2. При использовании такого размера страниц легко показать, что относительный адрес, который определяется относительно начала программы, и логический адрес, представляющий собой номер кадра и смещение, идентичны.

Использование страниц с размером, равным степени двойки, приводит к таким следствиям. Во-первых, схема логической адресации прозрачна для программиста, ассемблера и компоновщика. Каждый логический адрес (номер страницы и смещение) программы идентичен относительному адресу. Во-вторых, при этом относительно просто реализуется аппаратная функция преобразования адресов во время работы. Рассмотрим адрес из n+m бит, где крайние слева n бит представляют собой номер страницы, а крайние справа m бит - смещение. В данном примере n = 6 и m = 10.

Для преобразования адреса необходимо выполнить следующие шаги.

1. Выделить номер страницы, который представлен n левыми битами логического адреса
2. Используя номер страницы в качестве индекса в таблице страниц процесса, найти номер кадра k
3. Начальный физический адрес кадра - $k \cdot 2^m$, и интересующей нас физический адрес представляет собой это число плюс смещение. Такой адрес не надо вычислять - он получается в результате простого добавления номера кадра к смещению.

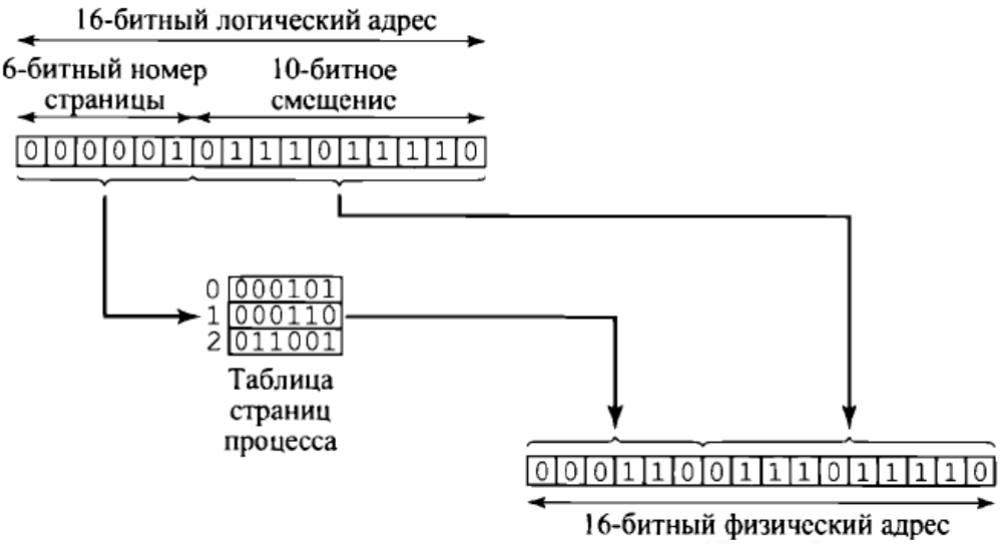

Рис. 20 – Пример преобразования адреса

Итак, в случае простой страничной организации основная память разделяется на множество небольших кадров одинакового размера. Каждый процесс разделяется на страницы того же размера, что и кадры; малые процессы требуют меньшего количества кадров, большие - большего. При загрузке процесса в память все его страницы загружаются в свободные кадры, и информация о размещении страниц заносится в соответствующую таблицу. Такой подход позволяет избежать множества присущих распределению памяти проблем.

43 Алгоритмы распределения памяти с использованием дискового пространства. Сегментное распределение памяти.

43.1 Алгоритмы распределения памяти с использованием дискового пространства

Смотри 42.1

43.2 Перемещение

Смотри 42.2

43.3 Сегментное распределение памяти

Альтернативным способом распределения пользовательской программы является сегментация. В этом случае программа и связанные с ней данные разделяются на ряд сегментов. Хотя и существует максимальный размер сегмента, на сегменты не накладывается условие равенства размеров. Как и при страничной организации, логический адрес состоит из двух частей, в данном случае - номера сегмента и смещения. Использование сегментов разного размера делает этот способ похожим на динамическое распределение памяти. Если не используются оверлеи и виртуальная память, то для выполнения программы все ее сегменты должны быть загружены в память; однако в отличие от динамического распределения в этом случае сегменты могут занимать несколько разделов, которые к тому же могут не быть смежными.

При сегментации устраняется внутренняя фрагментация, однако, как и при динамическом распределении, наблюдается внешняя фрагментация. Тем не менее ее степень снижается, в силу того что процесс разбивается на ряд небольших частей. В то время как страничная организация невидима для программиста, сегментация видима и обычно используется при размещении кода и данных в разных сегментах. При использовании принципов модульного программирования как код, так и данные могут быть дополнительно разбиты на сегменты. Главным недостатком при работе с сегментами является **необходимость заботиться о том, чтобы размер сегмента не превысил максимальный**.

Еще одно следствие того, что сегменты имеют разные размеры, состоит в отсутствии простой связи между логическими и физическими адресами. Аналогично страничной организации схема простой сегментации использует таблицу сегментов для каждого процесса и список свободных блоков основной памяти. Каждая запись таблицы сегментов должна содержать стартовый адрес сегмента в основной памяти и его длину, чтобы обезопасить систему от использования некорректных адресов. При работе процесса адрес его таблицы сегментов заносится в специальный регистр, используемый аппаратным обеспечением управления памятью. Рассмотрим адрес из n+m бит, где крайние слева n бит являются номером сегмента, а правые m бит - смещением.

Для трансляции адреса необходимо выполнить следующие действия.

1. Выделить из логического адреса n крайних слева битов, получив таким образом номер сегмента
2. Используя номер сегмента в качестве индекса в таблице сегментов процесса, найти физический адрес начала сегмента
3. Сравнить смещение, представляющее собой крайние справа т бит, с длиной сегмента. Если смещение больше длины, адрес некорректен

4. Требуемый физический адрес представляет собой сумму физического адреса начала сегмента и смещения

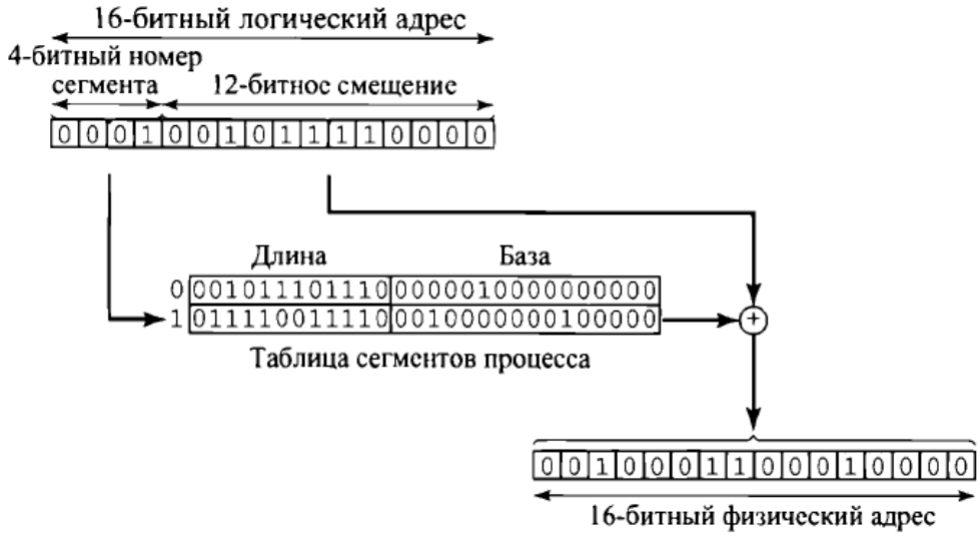

Рис. 21 – Пример преобразования адреса

Итак, в случае простой сегментации процесс разделяется на ряд сегментов, размер которых может быть разным. При загрузке процесса все его сегменты размещаются в свободных областях памяти и соответствующая информация вносится в таблицу сегментов.

44 Алгоритмы распределения памяти с использованием дискового пространства. Сегментно-страничное распределение памяти.

44.1 Алгоритмы распределения памяти с использованием дискового пространства

Смотри 42.1

44.2 Перемещение

Смотри 42.2

44.3 Сегментно-страничное распределение памяти

Как видно из названия, данный метод представляет собой комбинацию страничного и сегментного распределения памяти и, вследствие этого, сочетает в себе достоинства обоих подходов. Виртуальное пространство процесса делится на сегменты, а каждый сегмент в свою очередь делится на виртуальные страницы, которые нумеруются в пределах сегмента. Оперативная память делится на физические страницы. Загрузка процесса выполняется операционной системой постранично, при этом часть страниц размещается в оперативной памяти, а часть на диске. Для каждого сегмента создается своя таблица страниц, структура которой полностью совпадает со структурой таблицы страниц, используемой при страничном распределении. Для каждого процесса создается таблица сегментов, в которой указываются адреса таблиц страниц для всех сегментов данного процесса. Адрес таблицы сегментов загружается в специальный регистр процессора, когда активизируется соответствующий процесс. На рисунке 22 показана схема преобразования виртуального адреса в физический для данного метода.

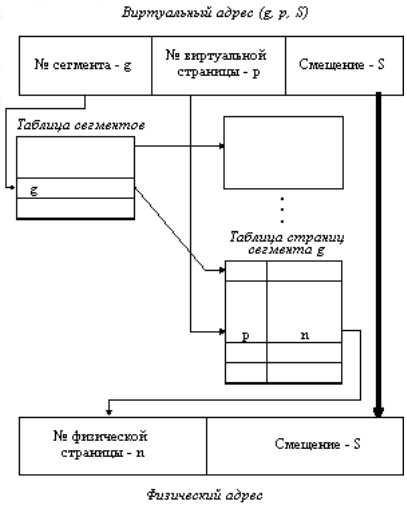

Рис. 22 – Пример преобразования адреса

45 Свопинг

Виртуальная память - это функциональная возможность, позволяющая программистам рассматривать память с логической точки зрения, не заботясь о наличии физической памяти достаточного объема.

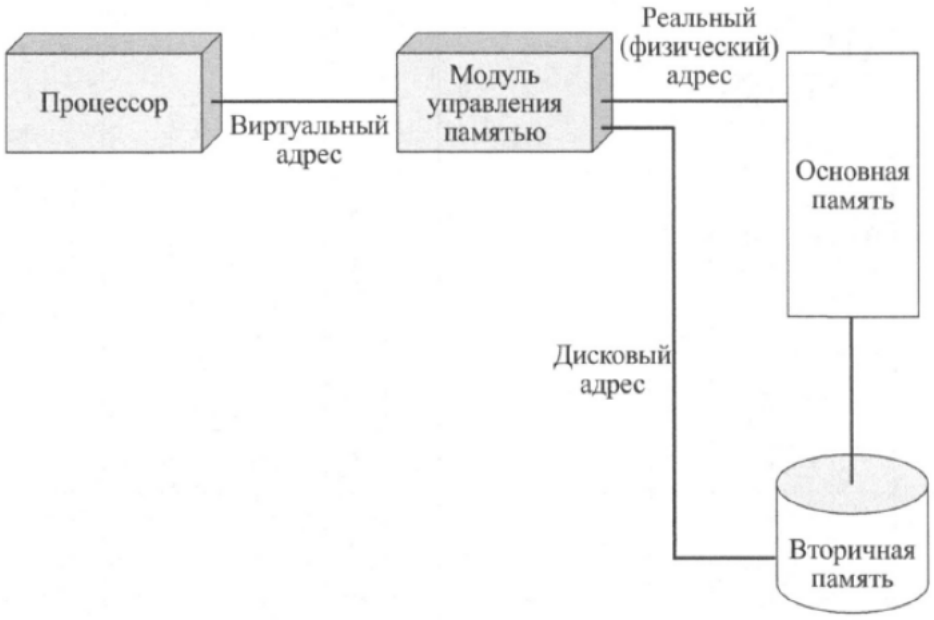

Рис. 23 – Адресация виртуальной памяти

Свопинг (подкачка страниц) — процесс освобождения части оперативной памяти и перенесение данных в заранее созданный файл подкачки на жестком диск.

Свопинг — частный случай виртуализации. Это наиболее простой способ совместного использования оперативной памяти и диска. Он позволяет освободить оперативную память для новых задач. Свопинг — весьма полезная опция, которая позволяет предотвратить тормоза и зависания компьютера или ноутбука при нехватке физической оперативной памяти.

Недостатки: очень низкая скорость работы а также дополнительный износ дискового накопителя.

45.1 Алгоритм свопинга

Описание алгоритма свопинга (swap) можно разбить на три части:

- Управление пространством на устройстве выгрузки

  Устройство выгрузки является устройством блочного типа, которое представляет собой конфигурируемый раздел диска. Тогда как обычно ядро выделяет место для файлов по одному блоку за одну операцию, на устройстве выгрузки пространство выделяется группами смежных блоков.

- Выгрузка процессов из основной памяти

  Ядро выгружает процесс, если испытывает потребность в свободной памяти. Теоретически все пространство памяти, занятое процессом, в том числе его личное адресное

пространство и стек ядра, может быть выгружено, хотя ядро и может временно заблокировать область в памяти на время выполнения критической операции. Однако практически, ядро не выгружает содержимое адресного пространства процесса, если в нем находятся таблицы преобразования адресов (адресные таблицы) процесса.

Временно выгруженные из памяти страницы могут сохраняться на внешних запоминающих устройствах как в файле, так и в специальном разделе на жёстком диске (partition), называемых соответственно swap-файл и swap-раздел. В случае необходимости выгрузить из ОЗУ страницы, соответствующие содержимому какого-либо файла на жёстком диске (например, memory-mapped files), они могут не выгружаться, а просто удаляться. При запросе такой страницы она может быть считана из оригинального файла. Когда приложение обратится к отсутствующей в ОЗУ странице, произойдет исключительная ситуация PageFault. Обработчик этого события должен проверить, была ли запрошенная страница ранее выгружена, и, если она есть в swap-файле, загрузить её обратно в память.

- Подкачка (загрузка) процессов в основную память

  Процесс подкачки просматривает все процессы, находящиеся в состоянии "готовности к выполнению, будучи выгруженными и выбирает из них один, который находится в этом состоянии дольше остальных. Если имеется достаточно свободной памяти, процесс подкачки загружает выбранный процесс, выполняя операции в последовательности, обратной выгрузке процесса. Сначала выделяется физическая память, затем с устройства выгрузки считывается нужный процесс и освобождается место на устройстве.

45.2 Пробуксовка

Пробуксовка — состояние, когда подсистема виртуальной памяти компьютера находится в состоянии постоянного свопинга, часто обменивая данные в памяти и данные на диске, в ущерб выполнению приложений. Это вызывает замедление или практическую остановку работы компьютера. Такое состояние может продолжаться неограниченно долго, пока вызвавшие его причины не будут устранены. Вероятность пробуксовки возрастает, когда сумма рабочих множеств всех процессов превышает объём оперативной памяти. Причинами этого могут быть слишком большое число запущенных процессов, слишком большой объём рабочих множеств процессов из-за нарушения в программах принципа локальности и недостаточный объём оперативной памяти.

46 Организация кэш-памяти.

Хотя кеш и невидим для операционной системы, он взаимодействует с аппаратным обеспечением, связанным с памятью.

При выполнении каждого цикла команды процессор по крайней мере один раз обращается к памяти, чтобы произвести выборку команды. Часто это происходит повторно, причем возможны случаи нескольких повторных обращений, при которых извлекаются операнды и/или сохраняются результаты. Очевидно, что частота, с которой процессор выполняет команды, ограничена временем обращения к памяти. Это ограничение было существенной проблемой из-за постоянного несоответствия между скоростью процессора и скоростью доступа к основной памяти - в течение многих лет скорость процессора возрастала быстрее, чем скорость доступа к памяти.

Наряду с относительно большой и более медленной основной памятью у нас есть кеш, обладающий меньшей емкостью, но и меньшим временем доступа. В кеше хранится копия фрагмента основной памяти. Когда процессор пытается прочесть байт или слово из памяти, выполняется проверка на наличие этого слова в кеше. Если оно там есть, этот байт или слово передается процессору. Если же его там нет, в кеш считывается блок основной памяти, состоящий из слов с определенными адресами, после чего требуемый байт или слово передается процессору. Вследствие локальности обращений при считывании в кеш блока данных, содержащего одно из требуемых слов, последующие обращения к данным с высокой вероятностью тоже будут выполняться к словам из этого блока.

Кеш предназначен для того, чтобы приблизить скорость доступа к памяти к максимально возможной и в то же время обеспечить большой объем памяти по цене более дешевых типов полупроводниковой памяти.

На рис. 24 изображен кеш, состоящий из нескольких уровней. Кеш L2 медленнее и обычно больше , чем кеш L 1, а кеш L3 медленнее и обычно больше, чем кеш L2.

Рис. 24 – Кеш, состоящий из нескольких уровней

Основная память состоит из 2n адресуемых слов, каждое из которых характеризуется своим уникальным n-битовым адресом. Для целей отображения предполагается, что вся память состоит из определенного количества блоков фиксированной длины, в каждый из которых входит K слов. Таким образом, всего имеется M=2n/K блоков. Кеш состоит из C слотов (именуемых также линиями) по K слов. При этом количество слотов намного меньше количества блоков (C на много меньше чем M). Некоторое подмножество блоков основной памяти хранится в слотах кеша. Если нужно прочесть из памяти слово из какого-то блока, которого нет в кеше, то этот блок передается в один из слотов кеша. Из-за того что блоков больше, чем слотов, нельзя закрепить за каждым блоком свой слот. Поэтому каждый слот должен содержать дескриптор, идентифицирующий хранящийся в нем блок. Количества блоков фиксированной длины, в каждый из которых входит к слов.

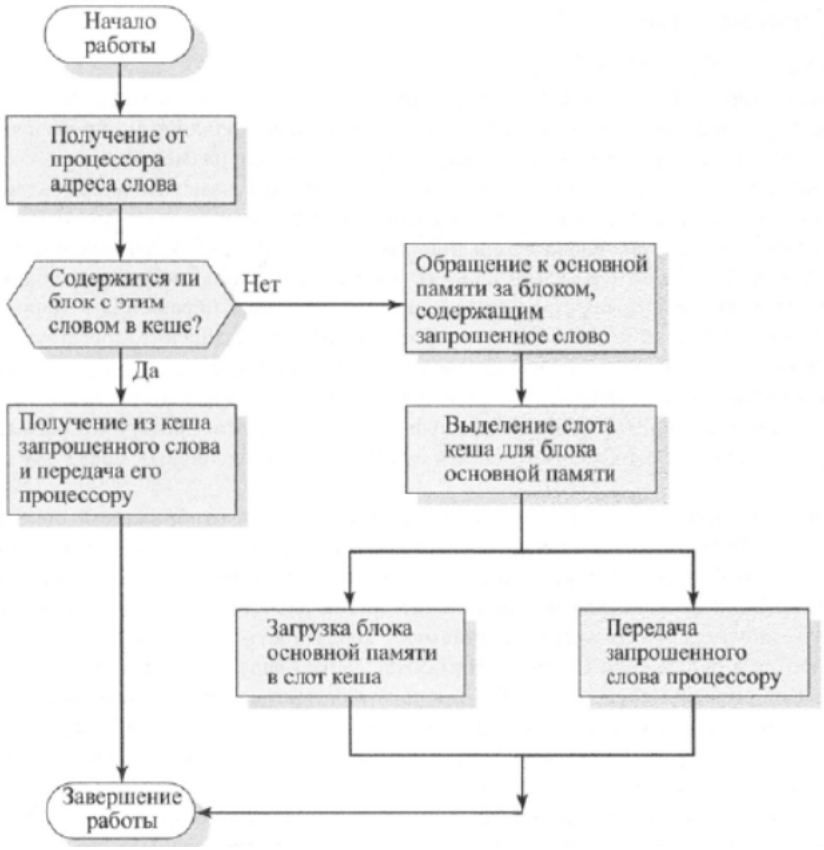

Рис. 25 – Структура кэша и основной памяти

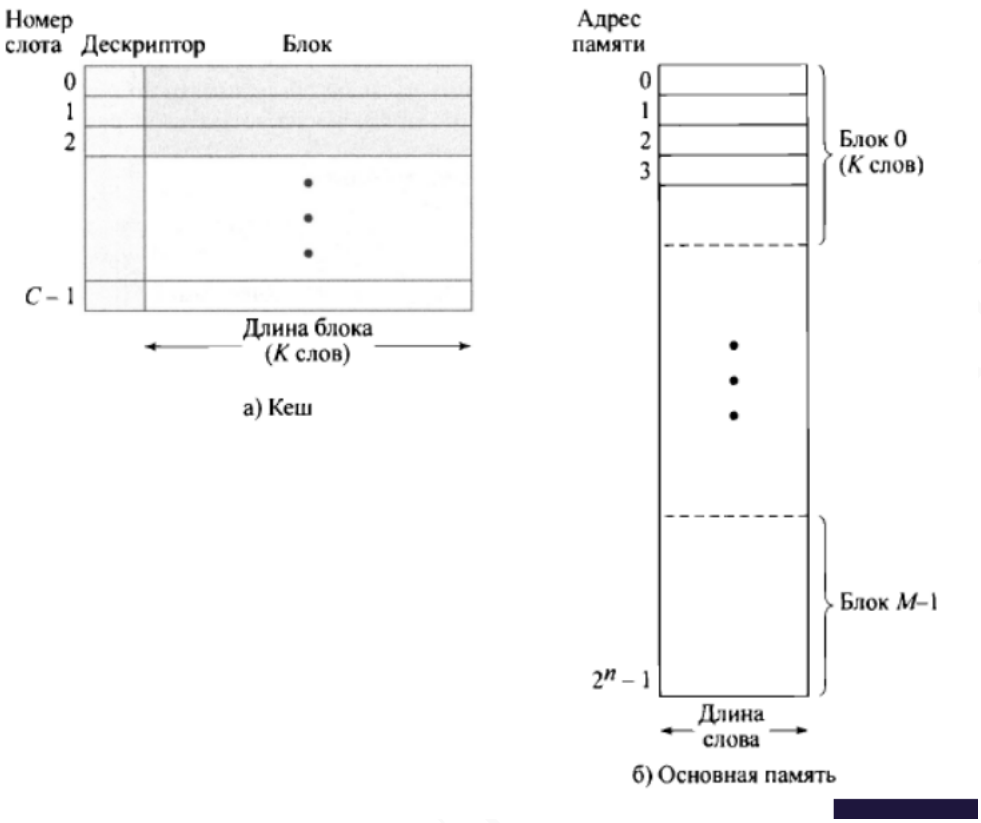

Рис. 26 – Структура кэша и основной памяти

47 Виртуальная память.

47.1 Определение

Виртуальная память (англ. virtual memory) — метод управления памятью компьютера, позволяющий выполнять программы, требующие больше оперативной памяти, чем имеется в компьютере, путём автоматического перемещения частей программы между основной памятью и вторичным хранилищем (например, жёстким диском). Для выполняющейся программы данный метод полностью прозрачен и не требует дополнительных усилий со стороны программиста, однако реализация этого метода требует как аппаратной поддержки, так и поддержки со стороны операционной системы.

В системе с виртуальной памятью используемые программами адреса, называемые виртуальными адресами, транслируются в физические адреса в памяти компьютера. Трансляцию виртуальных адресов в физические выполняет аппаратное обеспечение, называемое блоком управления памятью. Для программы основная память выглядит как доступное и непрерывное адресное пространство либо как набор непрерывных сегментов, вне зависимости от наличия у компьютера соответствующего объёма оперативной памяти. Управление виртуальными адресными пространствами, соотнесение физической и виртуальной памяти, а также перемещение фрагментов памяти между основным и вторичным хранилищами выполняет операционная система

47.2 Страничная организация виртуальной памяти

Смотри подробнее 42.3

В большинстве современных операционных систем виртуальная память организуется с помощью страничной адресации. Оперативная память делится на страницы: области памяти фиксированной длины (например, 4096 байт), которые являются минимальной единицей выделяемой памяти (то есть даже запрос на 1 байт от приложения приведёт к выделению ему страницы памяти). Исполняемый процессором пользовательский поток обращается к памяти с помощью адреса виртуальной памяти, который делится на номер страницы и смещение внутри страницы. Процессор преобразует номер виртуальной страницы в адрес соответствующей ей физической страницы при помощи буфера ассоциативной трансляции (кэш центрального процессора, который содержит записи - соответствия виртуального адреса страницы физическому адресу той же страницы)

47.3 Сегментная организация виртуальной памяти

Смотри подробнее 43.3

Механизм организации виртуальной памяти, при котором виртуальное пространство делится на части произвольного размера — сегменты. Этот механизм позволяет, к примеру, разбить данные процесса на логические блоки. Для каждого сегмента, как и для страницы, могут быть назначены права доступа к нему пользователя и его процессов. При загрузке процесса часть сегментов помещается в оперативную память, а часть сегментов размещается в дисковой памяти. Сегменты одной программы могут занимать в оперативной памяти несмежные участки.

Виртуальный адрес при сегментной организации памяти может быть представлен парой (g, s), где g — номер сегмента, а s — смещение в сегменте. Физический адрес получается путём сложения начального физического адреса сегмента, найденного в таблице сегментов по номеру g, и смещения s.

48 Дисковое планирование. Параметры производительности диска.

48.1 Дисковое планирование

Дисковым планированием называют набор правил по которым осуществляется организации доступа к дисковым накопителям.

Алгоритмы дискового планирования:

Название	Описание	Примечания
RSS	Случайное планирование	Для анализа и моделирования
FIFO	Первый вошёл — первым вышел	Наиболее простой и беспристрастный метод
PRI	Приоритет процесса	Очередь запросов к диску регулируется внешней системой
LIFO	Последним вошёл — первым вышел	Максимизация локализации использования ресурса
SSTF	выбор самого короткого времени обслуживания	Высокая степень использования, малые очереди
SCAN	Перемещение вперёд и назад по диску	Лучшее распределение обслуживания
C-SCAN	Однонаправленное перемещение с быстрым возвратом	Низкая изменчивость обслуживаания
N-step-SCAN	SCAN с N записями в пакете	Гарантия обслуживания

Рис. 27 – Алгоритмы дискового планирования

PRI - Приоритеты

Всистеме с использованием приоритета (PRI)управление планированием является внешним по отношению к программному обеспечению управления диском. Такой подход не имеет отношения к оптимизацииииспользования диска, но зато удовлетворяет некоторым другим целям операционнойсистемы. Коротким пакетным заданиям, а такжеинтерактивным заданиям часто присваивается более высокийприоритет, чем длинным заданиям, требующим более длительных вычислений. Эта схема позволяет быстро завершитьбольшое количество коротких заданий в системе обеспечивает малое время отклик

LIFO-последним вошел -первым вышел В системах обработки транзакций при предоставлении устройства для последнего пользователя должно выполняться лишь небольшое перемещение указателя последовательного файла. Использование преимуществ локализации позволяет повысить пропускную способность и уменьшить длину очереди. К сожалению, если нагрузка на диск велика, существует очевидная возможность голоданияпроцесса.

SSTF

Стратегия выбор наименьшего времени обслуживания (Shortest Service Time First — SSTF) заключается в выборе того дискового запроса, которые требует наименьшего времени перемещения головок из текущей позиции.

SCAN

Все предыдущие методы, кроме FIFO имеют недостаток — некоторые запросы в очереди могут лежать бесконечно долго. Стратегия SCAN позволяет избежать данной проблемы. При использовании алгоритма перемещение головки происходит только в одном направлении, удовлетворяя те запросы, которые соответствуют выбранному направлению. По достижению последней дорожки, направление меняется на противоположное.

CSCAN

Циклический SCAN. Работает аналогично SCAN, но при достижении последней дорожки головка перемещается в начало диска и не меняет направление.

48.2 Параметры производительности диска.

При работе диска его скорость вращения постоянна. Для того чтобы выполнить чтение или запись, головка должна находиться над искомой дорожкой, а кроме того — над началом искомого сектора на этой дорожке. Процедура выбора дорожки включает в себя перемещение головки (в системе с подвижными головками) или электронный выбор нужной головки (в системе с неподвижными головками). В системе с подвижными головками на позиционирование головки над дорожкой затрачивается время, известное как время поиска. В любом случае после выбора дорожки контроллер диска ожидает момент, когда начало искомого сектора достигнет головки. Время, необходимое для достижения головки началом сектора, известно как время задержки из-за вращения, или время ожидания вращения. Сумма времен поиска (если таковой выполняется) и времени задержки из-за вращения составляет время доступа — время, которое требуется для позиционирования для чтения или записи. Как только головка попадает в искомую позицию, выполняется операция чтения или записи, осуществляемая во время движения сектора под головкой, — это и есть непосредственная передача данных при выполнении операции ввода-вывода.

Можно выделить следующие параметры производительности диска:

- Время поиска — позиционирование головки диска.
- Задержки из-за вращения — ожидание, пока нужные данные "докрутятся"до позиции читающей головки. Зависит от скорости вращения диска.
- Время передачи данных — время, за которой с дорожки диска считается требуемое количество байт. Зависит от скорости вращения.

49 RAID. Уровни RAID.

49.1 RAID

RAID (англ. Redundant Array of Independent Disks — избыточный массив независимых дисков) — технология виртуализации данных, которая объединяет несколько дисков в логический элемент для повышения производительности. Соответственно, минимальное количество требуемых дисков — 2, но может потребоваться и больше. Всё зависит от того, какой именно массив вам нужен и для чего.

49.2 Уровни RAID

В типологии RAID выделают 5 основных уровней:

1. RAID0 — метод основан на чередовании записи/чтения данных. Для реализации необходимо взять 2 или более накопителей. Запись в такой массив будет производиться поочерёдно - первый блок данных в первый накопитель, второй блок во второй. Чтение, соответственно, будет происходить также блоками поочерёдно из каждого накопителя. Благодаря этому общая скорость работы увеличивается во столько раз, сколько накопителей используется. К тому же, общий объём системы будет равен сумме объёмов всех используемых накопителей. Однако, при выходе из строя одного накопителя, значительная часть информации может быть потеряна.
2. RAID1 — массив из двух и более более дисков, информацию в которых дублируется. В данном случае ускорения работы не происходит, как и увеличения ёмкости хранилища. Но при выходе из строя одного накопителя, информация не теряется, ведь есть полная копия на втором накопителе.
3. RAID2 — в данном варианте часть массива дисков используется для хранения информации, а часть для хранения кодов коррекции ошибок. Производительность хранилища при этом выше, чем у одиночных накопителей. Однако, превосходство над RAID1 появляется при использовании сборки из 7 накопителей (4 с данными, 3 для кодов коррекции)
4. RAID01 — при использовании этой схемы хранилище представляет массив RAID1, каждый элемент которого является массивом RAID0. То есть, итоговая ёмкость является половиной от фактической ёмкости используемых накопителей из-за дублирования информации на уровне массива RAID1. Однако, внутри самих элементов массива RAID1 информация не дублируется, из-за чего, при выходе из строя одного диска RAID1-ячейки, выходит из строя вся ячейка (так как сама ячейка использует RAID0).
5. RAID10 — схема является улучшенной версией RAID01 при которой система представляет из себя массив RAID0 ячеек, каждая ячейка которого - RAID1 массив. В данной ситуации повышение производительности происходит глобально, а дупликация локально. Что логичнее и эффективнее, чем вариант RAID01

Также существуют RAID3, RAID4, RAID5, RAID6, RAID7, RAID1E и их вариации.

50 Определение файловой системы. Именование файлов. Атрибуты файлов. Методы обеспечения доступа к файлам.

50.1 Определение

Файловая система (англ. file system) — порядок, определяющий способ организации, хранения и именования данных на носителях информации в компьютерах, а также в другом электронном оборудовании: цифровых фотоаппаратах, мобильных телефонах и т. п. Файловая система определяет формат содержимого и способ физического хранения информации, которую принято группировать в виде файлов. Конкретная файловая система определяет размер имен файлов (и каталогов), максимальный возможный размер файла и раздела, набор атрибутов файла. Некоторые файловые системы предоставляют сервисные возможности, например, разграничение доступа или шифрование файлов.

Файловая система связывает носитель информации с одной стороны и API для доступа к файлам — с другой. Когда прикладная программа обращается к файлу, она не имеет никакого представления о том, каким образом расположена информация в конкретном файле, так же как и о том, на каком физическом типе носителя (CD, жёстком диске, магнитной ленте, блоке флеш-памяти или другом) он записан. Всё, что знает программа — это имя файла, его размер и атрибуты. Именно файловая система устанавливает, где и как будет записан файл на физическом носителе (например, жёстком диске).

50.2 Именование файлов

По способам именования файлов различают “короткое” и “длинное” имя. В операционной системе MS-DOS имя файла состояло из 8 символов имени и 3 символов расширения (данная схема также известна под именем 8.3). Сейчас такое имя называется коротким. Начиная с Windows 95 было введено понятие длинного имени, которое может содержать до 256 символов.

50.3 Атрибуты файлов

Атрибут файла — метаданные, которые описывают файл. Атрибут может находиться в двух состояниях: либо установленный, либо снятый. Атрибуты рассматриваются отдельно от других метаданных, таких как даты, расширения имени файла или права доступа.

В Linux выделяют следующие атрибуты:

- Read — обозначается буквой ’r’. Сигнализирует о возможности чтения данного файла.
- Write — обозначается буквой ’w’. Сигнализирует о возможности изменения файла.
- Execute — обозначается буквой ’x’. Сигнализирует о возможности запуска файла как программы пользователями системы.

50.4 Методы доступа к файлам

Существует два подхода к определению прав доступа:

1. Избирательный доступ – ситуация, когда владелец объекта определяет допустимые операции с объектом. Этот подход называется также произвольным доступом, так как позволяет администратору и владельцам объекта определить права доступа произвольным образом, по их желанию. Однако администратор по умолчанию наделен всеми правами.

2. Мандатный доступ (от mandatory – принудительный) – подход к определению прав доступа, при котором система (администратор) наделяет пользователя или группу определенными правами по отношению к каждому разделяемому ресурсу. В этом случае группы пользователей образуют строгую иерархию, причем каждая группа пользуется всеми правами группы более низкого уровня иерархии. Мандатные системы доступа считаются более надежными, но менее гибкими. Обычно они применяются в системах с повышенными требованиями к защите информации.

51 Задачи файловой системы. Логическая организация файловой системы.

Определение файловой системы смотри в 50.1

51.1 Задачи файловой системы

Основные функции любой файловой системы нацелены на решение следующих задач:

- именование файлов
- программный интерфейс работы с файлами для приложений.
- отображения логической модели файловой системы на физическую организацию хранилища данных.
- организация устойчивости файловой системы к сбоям питания, ошибкам аппаратных и программных средств;
- содержание параметров файла, необходимых для правильного его взаимодействия с другими объектами системы (ядро, приложения и пр.).

В многопользовательских системах появляется ещё одна задача: защита файлов одного пользователя от несанкционированного доступа другого пользователя, а также обеспечение совместной работы с файлами, к примеру, при открытии файла одним из пользователей, для других этот же файл временно будет доступен в режиме «только чтение».

51.2 Логическая организация файловой системы

Предоставление пользователю комфортной работы с данными, хранящимися на дисках, является одной из основных задач операционной системы. Для этого ОС подменяет физическую структуру хранящихся данных удобной для пользователя логической структурой. Логическая структура файловой системы реализуется в виде дерева каталогов, символьными именами файлов и командами работы с файлами и каталогами. Базовым элементом этой структуры является файл.

Файл — это именованная область внешней памяти, в которую можно записывать и из которой можно считывать данные. Файлы хранятся в памяти, не зависящей от энергопитания, обычно — на магнитных или электронных дисках.

52 Логическая организация файла.

В общем случае данные, содержащиеся в файле, имеют некую логическую структуру. Эта структура является базой при разработке программы, предназначенной для обработки этих данных. Например, чтобы текст мог быть правильно выведен на экран, программа должна иметь возможность выделить отдельные слова, строки, абзацы и т. д. Признаками, отделяющими один структурный элемент от другого, могут служить определенные кодовые последовательности или просто известные программе значения смещений этих структурных элементов относительно начала файла. Поддержание структуры данных может быть либо целиком возложено на приложение, либо в той или иной степени эту работу может взять на себя файловая система.

В первом случае, когда все действия, связанные со структуризацией и интерпретацией содержимого файла целиком относятся к ведению приложения, файл представляется ФС неструктурированной последовательностью данных. Приложение формулирует запросы к файловой системе на ввод-вывод, используя общие для всех приложений системные средства, например, указывая смещение от начала файла и количество байт, которые необходимо считать или записать. Поступивший к приложению поток байт интерпретируется в соответствии с заложенной в программе логикой. Например, компилятор генерирует, а редактор связей воспринимает вполне определенный формат объектного модуля программы. При этом формат файла, в котором хранится объектный модуль, известен только этим программам. Подчеркнем, что интерпретация данных никак не связана с действительным способом их хранения в файловой системе.

Модель файла, в соответствии с которой содержимое файла представляется неструктурированной последовательностью (потоком) байт, стала популярной вместе с ОС UNIX, а теперь она широко используется в большинстве современных ОС, в том числе в MS-DOS, Windows NT/2000, NetWare. Неструктурированная модель файла позволяет легко организовать разделение файла между несколькими приложениями: разные приложения могут по-своему структурировать и интерпретировать данные, содержащиеся в файле.

Логическая запись является наименьшим элементом данных, которым может оперировать программист при организации обмена с внешним устройством. Даже если физический обмен с устройством осуществляется большими единицами, операционная система должна обеспечивать программисту доступ к отдельной логической записи.

53 Физическая организация файла.

Физическая организация файла описывает правила расположения файла на устройстве внешней памяти, в частности на диске. Файл состоит из физических записей – блоков. Блок – наименьшая единица данных, которой внешнее устройство обменивается с оперативной памятью.

Существуют следующие способы физической организации файла:

1. **Непрерывное размещение** – простейший вариант физической организации при котором файлу предоставляется последовательность блоков диске, образующих последовательность блоков диска, образующих единый сплошной участок дисковой памяти. Для задания адреса файла в этом случае достаточно указать только номер начального блока. Другое достоинство этого метода – простота. Но имеются и два существенных недостатка. Во-первых, во время создания файла не известна его длина, а значит неизвестно, сколько памяти надо зарезервировать для этого файла, во-вторых, при таком порядке размещения неизбежно возникает фрагментация, и пространство на диске используется не эффективно, так как отдельные участки маленького размера (минимально 1 блок) могут остаться не используемыми.

2. **Размещение в виде связанного списка блоков дисковой памяти**. При таком способе в начале каждого блока содержится указатель на следующий блок. В этом случае адрес файла также может быть задан одним числом – номером первого блока. В отличие от предыдущего способа, каждый блок может быть присоединен в цепочку какого-либо файла, следовательно, фрагментация отсутствует. Файл может изменяться во время своего существования, наращивая число блоков. Недостатком является сложность реализации доступа к произвольно заданному месту файла. Кроме того, при этом способе количество данных файла, содержащихся в одном блоке, не равно степени двойки (то есть, если размер блока равен 64 слова, то одно слово содержит адрес следующего блока. Следовательно, полезной информации остаётся 63 слова, что не кратно степени двойки).

3. **Использование связанного списка индексов.** С каждым блоком связывается некоторый элемент – индекс. Индексы располагаются в отдельной области диска. Если некоторый блок распределен некоторому файлу, то индекс этого блока содержит номер следующего блока данного файла. При такой физической организации сохраняются все достоинства предыдущего способа, но снимаются оба отмеченных недостатка – для доступа к произвольному месту файла достаточно прочитать только блок индексов, отсчитать нужное количество блоков файла по цепочке и определить номер нужного блока. Разновидность – хранение номеров блоков с данными в специальной структуре – индексном дескрипторе.

54 Отображаемые в память файлы. Использование файлов, проецируемых в память в разработке.

Отображаемый в память файл (Memory-mapped file) — сегмент виртуальной памяти (см. Билет 47), который воспринимается программой как обычный файл.

Абстракция в виде виртуальной памяти помогает разработчику не задумываться о фактическом местонахождении файла, а операционной системе оптимизировать скорость доступа к файлам.

Отображаемым в память файлом может называться любой ресурс, на который система может сослаться посредством файлового дескриптора. Файловый дескриптор — право чтения/записи файла или потока, выдающееся операционной системой процессам.

Отображаемый в память файл может быть:

1. Сохранённым или соответствующим файлу на диске и будет сохранён на физическом хранилище после окончания процесса.
2. Не сохранённым или представленным исключительно в виртуальной памяти и будет удалён после окончания процесса.

55 Распределенные файловые системы. Основные понятия.

Распределенная файловая система — файловая система, которая предоставляет пользователю возможность использовать несколько физических хранилищ так, как будто бы это локальное хранилище файлов. Физические хранилища соединяются друг с другом через сеть или посредством высокоскоростной шины.

Преимущества распределённых файловых систем:

- Отказоустойчивость и отсутствие единой точки отказа — если одно их хранилищ выйдет из строя, остальные будут доступны;
- Высокая производительность — запросы доступа к данным распределяются между физическими хранилищами;
- Безопасность — запросы извне можно фильтровать;
- Надёжность — можно иметь копии данных сразу на нескольких физических хранилищах;
- Расширяемость — всегда можно увеличить объём файловой системы, докупив физических хранилищ;

Недостатки распределённых файловых систем:

- Задержки, связанные с сетью;
- Дополнительные расходы на:
  - Разворачивание и поддержание сложной инфраструктуры;
  - Обеспечения безопасности;
  - Сами хранилища.

56 Семантика разделения файлов.

Во избежание непредвиденных ошибок, нужно точно определить как конкретно будут вносится изменения в файл, который читается и изменяется несколькими пользователями одновременно, и как внесённые одним пользователем изменения будут влиять на других пользователей.

Существуют разные подходы в решении проблемы одновременного чтения и изменения файла:

1. Семантика UNIX: операции чтения/записи выполняются последовательно, а все пользователи немедленно видят результат изменений. Из-за ограничений сети этот подход является неэффективным в распределённых системах — приходится слишком часто отправлять и получать обновления файлов.
2. Сеансовая семантика: до окончания внесения изменений они видны только вносящему их процессу. Жирный минус такого подхода заключается в том, что, при одновременном редактировании, изменение, вносящееся последним, или перезатрёт все предыдущие, или будет недетерменированным.
3. Семантика неизменяемых файлов: все файлы являются неизменяемыми. Это приводит к распуханию файловой системы и сложностям в использовании файлов из-за их версионирования.
4. Семантика транзакций: изменения вносятся транзакционно. В отличие от сеансов, изменения всегда вносятся осмысленными пачками — это позволяет избежать несогласованности файлов.

57 Организация устройств ввода/вывода.

57.1 Ввод/вывод. Устройства ввода/вывода.

Программные модули и устройства взаимодействуют друг с другом посредством интерпретируемых сигналов. Операции ввода/вывода — обмен такими сигналами. Устройства ввода/вывода упрощают формирование сигналов до физических действий или использования интерфейса.

Обозначение ввода/вывода зависит от перспективы: записываемый на микрофон звук одновременно является выводом для микрофона и вводом для компьютера.

Примеры таких устройств:

- Устройства ввода:
  - Клавиатура
  - Мышь
- Устройства вывода:
  - Монитор
  - Принтер
  - Наушники
- Устройства ввода/вывода:
  - Модем
  - Сетевая карта
  - Звуковая карта

Ввод/вывод может осуществляться тремя способами:

1. Программный ввод/вывод — контролируемый программой ввод/вывод. Пример: редактирование файла в текстовом редакторе.
2. Инициированный прерыванием ввод/вывод — уведомление процессора о событии посредством сигнала прерывания. Благодаря сигналу прерывания процессору не приходится регулярно опрашивать устройство и контролировать его ввод/вывод. Пример: при нажатии клавиши на клавиатуре, процессору отправляется сигнал прерывания.
3. Прямой доступ к памяти — обмен данными с запоминающим устройством, минуя процессор. Пример: сетевая карта при скачивании записывает файл на диск напрямую.

57.2 Проблемы организации ввода/вывода.

(см. Билет 59)

58 Подсистема управления вводом/выводом ОС. Физическая организация устройств ввода/вывода.

58.1 Подсистема управления вводом/выводом ОС.

Подсистема управления вводом/выводом ОС предоставляет интерфейс для выполнения операций ввода/вывода и выполняет сопутствующие ему задачи:

- Планирование ввода/вывода — определение порядка выполнения запросов;
- Буферизация — временное сохранение данных ввода/вывода между двумя устройствами или устройством и приложением. Может потребоваться если:
  - Есть разница в скорости между производителем потока данных и его потребителем;
  - Данные нужно по-другому представить;
  - Данные вычитываются или записываются порционно.
- Кэширование — временная загрузка данных в более быструю память ради увеличения скорости чтения.
- Спулинг (Spooling, от англ. Spool — катушка) — буфер, представляющий собой поток данных, предназначенный для устройства, которое временно не может корректно обработать все подаваемые ему на вход данные. Пример: принтер не может печатать несколько документов сразу — для этого нужно последовательно подавать документы на вход.
- Обработка ошибок — предотвращение прекращения работы ОС в результате ошибок ввода/вывода.
- Обеспечение безопасности — предотвращение несанкционированного доступа к вводу/-выводу.

58.2 Физическая организация устройств ввода/вывода.

Одним из главных элементов внешних устройств ввода/вывода является контроллер. Он выступает в роли интерфейса между операционной системой и самим устройством и выполняет следующие задачи:

- Выступает посредником в обмене сообщениями между операционной системой и устройством и интерпретирует их;
- Управляет устройством;
- Обрабатывает ошибки.

Пример обмена сообщениями между ОС и контроллером:

1. Процессор записывает одну из команд (READ, WRITE и т.д.) в регистр контроллера и переключается на другие задачи.
2. Контроллер обрабатывает сообщение, записывает результат в свой регистр и отправляет процессору сигнал прерывания.
3. Процессор получает сигнал прерывания и проверяет результат выполнения команды.

59 Основные проблемы организации ввода/вывода. Многослойная модель подсистемы ввода-вывода.

Основной проблемой организации ввода/вывода является многообразие устройств и интерфейсов взаимодействия с ними. Для того, чтобы изолировать операционную систему от обработки низкоуровневых вызовов, сигналов и ошибок устройств ввода/вывода и упростить их использование выделяются абстракции — слои.

На рис. 28 продемонстрированы основные абстракции ввода/вывода.

Рис. 28 – Абстракции ввода/вывода

Каждый из слоёв взаимодействует с соседним: драйвера в зависимости от полученных сигналов вызывают менеджер и, наоборот, отправляют соответствующие обращениям менеджера сигналы устройствам. Благодаря многослойной модели устройства и сам менеджер не зависят от реализаций устройств. Но архитектурное решение — это всегда компромисс: актуальные версии драйверов нужно разрабатывать и поддерживать.

Стоит понимать, что менеджер ввода/вывода направляет сигналы в конкретные драйвера и может вызывать другие модули ОС.

60 Организация программного обеспечения ввода/вывода. Драйверы устройств.

60.1 Организация программного обеспечения ввода/вывода.

(см. Билет 57)

60.2 Драйверы устройств.

Драйвер устройства — программа, представляющая собой интерфейс для взаимодействия между устройством и операционной системой, которая интерпретирует сигналы устройства и вызовы операционной системы.

Типы драйверов устройств:

- Драйвер устройства в режиме ядра — драйвер к обязательному для работы компьютера устройству. Загружается вместе с ОС и содержит в себе общие аппаратные средства. Для работы компьютеру необходимо иметь минимальный набор драйверов.
- Драйвер устройства в режиме пользователя — драйвер к пользовательскому устройству. Для корректной работы любого устройства ОС нужен совместимый драйвер. Современные операционные системы чаще всего подгружают нужные драйвера самостоятельно или уже содержат в себе подходящую версию.