ФАКУЛЬТЕТ
ВЫЧИСЛИТЕЛЬНОЙ
МАТЕМАТИКИ И
КИБЕРНЕТИКИ
МГУ ИМЕНИ
М.В. ЛОМОНОСОВА

teach-in
ЛЕКЦИИ УЧЕНЫХ МГУ

ОПЕРАЦИОННЫЕ СИСТЕМЫ

МАШЕЧКИН
ИГОРЬ ВАЛЕРЬЕВИЧ

ВМК МГУ

КОНСПЕКТ ПОДГОТОВЛЕН СТУДЕНТАМИ, НЕ ПРОХОДИЛ ПРОФ. РЕДАКТУРУ И МОЖЕТ СОДЕРЖАТЬ ОШИБКИ. СЛЕДИТЕ ЗА ОБНОВЛЕНИЯМИ НА VK.COM/TEACHINMSU.

ЕСЛИ ВЫ ОБНАРУЖИЛИ ОШИБКИ ИЛИ ОПЕЧАТКИ, ТО СООБЩИТЕ ОБ ЭТОМ, НАПИСАВ СООБЩЕСТВУ VK.COM/TEACHINMSU.

БЛАГОДАРИМ ЗА ПОДГОТОВКУ КОНСПЕКТА
СТУДЕНТКУ ФИЗИЧЕСКОГО ФАКУЛЬТЕТА МГУ
ХОТЛУБЕЙ ЕЛИЗАВЕТУ СЕРГЕЕВНУ

Содержание

1 Лекция 1. Введение 9

1.1 Введение. Компьютеры первого поколения 9
1.2 Компьютеры второго поколения 12
1.3 Компьютеры третьего поколения 14

2 Лекция 2. Поколения компьютеров 15

2.1 Компьютеры семейства IBM 360 и другие семейства 15
2.2 Четвертое поколение компьютеров. Микропроцессоры 16
2.3 Суперкомпьютерные технологии 17
2.4 Вычислительные системы и их уровни 17
2.5 Аппаратный уровень 18
2.6 Уровень управления физическими ресурсами 18
2.7 Физические ресурсы. Драйвер 19
2.8 Управление виртуальными или логическими устройствами 20

3 Лекция 3. Основы архитектуры вычислительной системы 21

3.1 Основы архитектуры вычислительной системы 21
3.2 Управление физическими ресурсами вычислительной системы 21
3.3 Управление виртуальными (логическими) ресурсами 22
3.4 Уровень систем программирования 24
3.5 Прикладные системы 26
3.6 Выводы. Список терминологии 26
3.7 Основы архитектуры компьютера. Компьютер фон Неймана 27

4 Лекция 4. Основы компьютерной архитектуры 29

4.1 Принципы фон Неймана 29
4.2 Оперативное запоминающее устройство 29
4.3 Расслоение памяти 32
4.4 Центральный процессор. Регистры 34
4.5 Рабочий цикл процессора 35
4.6 Кэш-память 36

5 Лекция 5. Иерархия памяти 38

5.1 Оперативное запоминающее устройство, центральный процессор и оперативная память 38
5.2 Центральный процессор 38

5.3 Аппарат прерывания 39
5.4 Схема обработки прерываний 40
5.5 Внешние запоминающие устройства (ВЗУ) 42
5.6 Синхронная организация обмена 44
5.7 Иерархия устройств хранения информации 45
5.8 Аппаратная поддержка ОС и систем программирования 46

6 Лекция 6. Аппаратная поддержка ОС 48
6.1 Базовая аппаратная поддержка мультипрограммного режима 48
6.2 Перемещаемость программы по ОЗУ. Фрагментация памяти 50
6.3 Виртуальная память 51
6.4 Адресация с базированием. Страничная организация памяти 52
6.5 Регистровые окна 55
6.6 Классификация архитектур многопроцессорных ассоциаций 56
6.7 Иерархия MIMD-систем 57

7 Лекция 7. Классификация архитектур многопроцессорных ассоциаций 58
7.1 Классификация архитектур многопроцессорных ассоциаций 58
7.2 Иерархия MIMD-систем 58
7.3 SMP-системы 60
7.4 Синхронизация кэша 60
7.5 NUMA-системы 62
7.6 Системы с распределенной памятью (MPP-системы) 62
7.7 COW-кластеры (кластеры рабочих станций) 63
7.8 Терминальные комплексы 64
7.9 Линии связи и каналы 66
7.10 Компьютерные сети 67

8 Лекция 8. Компьютерные сети 69
8.1 Компьютерные сети 69
8.2 Сеть коммутации каналов 69
8.3 Сеть коммутации сообщений 70
8.4 Сеть коммутации пакетов 71
8.5 Модель ISO/OSI организации взаимодействия в сети 72
8.6 Логическое взаимодействие по протоколу 73

ФАКУЛЬТЕТ ВЫЧИСЛИТЕЛЬНОЙ МАТЕМАТИКИ И КИБЕРНЕТИКИ МГУ ИМЕНИ М.В. ЛОМОНОСОВА

9 Лекция 9. Организация сетевого взаимодействия 76
9.1 Модель ISO/OSI взаимодействия в сети . . . 76
9.2 Соответствие модели ISO/osi модели семейства протоколов TCP/IP . . 77
9.3 Взаимодействия между уровнями протоколов TCP/IP . . . 79
9.4 Уровень доступа к сети . . . 80
9.5 Протокол IP. Межсетевой уровень . . . 81
9.6 Задача маршрутизации . . . 82
9.7 Транспортный уровень . . . 83
9.8 Уровень прикладных программ . . . 84
9.9 Основы архитектуры операционных систем . . . 85

10 Лекция 10. Основы архитектуры ОС 86
10.1 Основы архитектуры ОС . . . 86
10.2 Требования к ОС . . . 86
10.3 Структурная организация ОС . . . 87
10.4 Архитектура ядра . . . 88
10.5 Логические функции ОС . . . 90
10.6 Типы ОС. Пакетная ОС . . . 90
10.7 Системы разделения времени . . . 91
10.8 Системы планирования . . . 92

11 Лекция 11. Процесс в ОС 93
11.1 Разновидности операционных систем (ОС). Пакетная . . . 93
11.2 ОС реального времени . . . 94
11.3 Управление процессами . . . 95
11.4 Модельная ОС . . . 96
11.5 Типы процессов . . . 98
11.6 Процесс в Unix . . . 100

12 Лекция 12. Контекст процесса 103
12.1 Контекст процесса . . . 103
12.2 Создание нового процесса. Системный вызов fork() . . . 106
12.3 Семейство системных вызовов exec() . . . 109
12.4 Использование схемы fork-exec . . . 111
12.5 Завершение процесса _exit() . . . 112
12.6 Получение информации о завершении своего потомка wait . . . 113
12.7 Жизненный цикл процессов в UNIX . . . 115

13 Лекция 13. Параллельные процессы **116**
13.1 Жизненный цикл процесса в Unix 116
13.2 Начальная загрузка 117
13.3 Инициализация системы 118
13.4 Схема дальнейшей работы системы 119
13.5 Параллельные процессы 120
13.6 Разделение ресурсов 121
13.7 Требование мультипрограммирования 121
13.8 Тупики (deadlocks) 123
13.9 Способы реализации взаимного исключения 123

14 Лекция 14. Классические задачи синхронизации **127**
14.1 Классические задачи синхронизации процессов 127
14.2 Обедающие философы 127
14.3 Читатели и писатели 131
14.4 Спящий парикмахер 133
14.5 Реализация взаимодействия процессов 135
14.6 Сигналы 137
14.7 Работа с сигналами 138

15 Лекция 15. Взаимодействие процессов **142**
15.1 Взаимодействие процессов 142
15.2 Работа с сигналами 144
15.3 Неименованные каналы. Системный вызов pipe() 145
15.4 Конвейер 150
15.5 Совместное использование сигналов и каналов "пинг-понг 151
15.6 Именованные каналы 154

16 Лекция 16. Межпроцессорное взаимодействие **155**
16.1 Модель межпроцессорного взаимодействия "Главный-подчиненный 155
16.2 Схема установки контрольной точки 157

17 лекция 17. Операционные системы – Система IPC **159**
17.1 Система межпроцессорного взаимодействия IPC. Общая концепция . . 159
17.2 Очередь сообщений IPC 161
17.3 Доступ к очереди сообщений. 162

17.4 Отправка сообщения. 163
17.5 Получение сообщений 164
17.6 Управление очередью сообщений. 164
17.7 Примеры. 165
17.8 Пример "Клиент-сервер 167
17.9 Разделяемая память IPC 169
17.10 Массив семафоров 172

18 Лекция 18. Работа с разделяемой памятью с синхронизацией семафорами 175
18.1 Работа с разделяемой памятью с синхронизацией семафорами 175
18.2 Взаимодействие процессов посредством сокетов 177
18.3 Предварительное установление соединения 179
18.4 Сокеты без предварительного соединения 181
18.5 Управление оперативной памятью 182
18.6 Одиночное непрерывное распределение 183
18.7 Распределение неперемещаемыми разделами 184
18.8 Распределение перемещаемыми разделами 186

19 Лекция 19. Управление оперативной памятью 187
19.1 Обзор прошлой лекции 187
19.2 Страничное распределение 190
19.3 Буфер быстрого преобразования адресов (TLB) 191
19.4 Иерархическая организация таблицы страниц 192

20 Лекция 20. Страничное распределение памяти 194
20.1 Использование хэш-таблицы 194
20.2 Инвертированные таблицы страниц 195
20.3 Сегментная организация памяти 198
20.4 Сегментно-страничная организация памяти 200
20.5 Файловые системы 200
20.6 Основные сценарии работы с файлом 203
20.7 Типовые программные интерфейсы работы с файлами 204

21 Лекция 21. Файловые системы. 205
21.1 Файловые системы. Структурная организация. 205
21.2 Атрибуты файла и основные сценарии работы 206

21.3 Типовые программные интерфейсы работы с файлами 208
21.4 Модельная организация 208
21.5 Структура системного диска 210
21.6 Модели реализации файлов. Непрерывные файлы. 211
21.7 Файлы, имеющие организацию связанного списка 212
21.8 Таблица размещения файловой системы FAT 214

22 Лекция 22. Основные концепции файловых систем. 216
22.1 Использование индексных дескрипторов (узлов) 216
22.2 Организация каталогов 217
22.3 Соотношение имени и содержимого файла 217
22.4 Координация использования пространства внешней памяти 218
22.5 Ресурсы системы 219
22.6 Квотирование пространства внешней памяти 219
22.7 Надежность файловой системы 220
22.8 Проверка целостности системной информации 222

23 Лекция 23. Файловая система Unix. 224
23.1 Организация файловой системы Unix 224
23.2 Права доступа 225
23.3 Логическая структура каталогов 226
23.4 Модель версии system V 228
23.5 Работа с массивами номеров свободных индексных дескрипторов 229
23.6 Индексные дескрипторы 230
23.7 Файл каталог 231
23.8 Достоинства и недостатки файловой системы System V 232

24 Лекция 24. Управление внешними устройствами. 234
24.1 Управление внешними устройствами 234
24.2 Программное управление внешними устройствами 236
24.3 Планирование дисковых обменов 236
24.4 RAID системы 237
24.5 Работа с внешними устройствами в операционной системе UNIX 239
24.6 Файлы устройств 240
24.7 Организация обмена данными с файлами 241

Лекция 1. Введение

Введение. Компьютеры первого поколения

Рассмотрим этапы развития вычислительной техники с точки зрения развития вычислительной системы. В истории развития вычислительной техники можно выделить четыре поколения компьютеров.

Компьютеры первого поколения появились в первой половине - середине 1940-х годов. Это были устройства, разработанные в германии и США. Аналогичные работы у нас проводились чуть позже. ENIAC (Electronic Numerical Integrator and Computer), разработанный Пенсильванским университетом США, считается первым компьютером. Его назначение - решение военных задач. В том числе баллистических и энергетических задач. Элементная база компьютера - электронно-вакуумные лампы ($\sim$ 20 тысяч). В то же время были идентичные по характериситикам отечественные разработки. Компьютеры первого поколения занимались расчетными задачами.
Проблемы :

- Большое устройство с высоким энергопотреблением и высокой энергоотдачей. Из-за перепадов температуры лампы выходили из строя, а в местах подключения ламп возникали неконтакты. Ежемесячно заменяли до 2000 ламп.
- За счет больших габаритов устройство имело маленькую производительность порядка тысяч операций в секунду

Программирование компьютера представляло инженерно сложный процесс. Компьютеры первого поколения имели центральную часть (процессор, оперативная память) и внешнее устройство, которое позволяло сохранить информацию долговременно. Информация либо выводилась на бумажный носитель (перфокарты, перфоленты), либо хранилась на магнитном запоминающем устройстве (магнитной ленте). Информация загружалась в память компьютера посредством инженерных пультов. Можно было задавать адреса оперативной памяти и содержимое с помощью тумблеров.

Самые ранние компьютеры не имели средств автоматизации программирования. Программировали в кодах машины. Для того, чтобы внести программу с перфоленты, необходимо было снача вручную с помощью тумблеров занести в память программу, которая обеспечит запуск и считывание информации со считывателя перфоленты. В результате, в некоторой области оперативного запоминающего устройства появлялась программа, которая была на перфоленте. Затем на регистрах задавался

адрес входа в эту программу. При запуске программы могли произойти некоторые непредвиденные ситуации, что вызывало остановку машины. В этом случае инженерный пульт использовался как отладчик.

Основные трудности в работе с такой машиной:

- программисту не просто было необходимо знать все системные особенности компьютера, но и вводить данные со специального пульта в двоичном (машинном) коде;
- необходимо было владеть математическими методами, позволяющими строить математические модели и алгоритмы, досконально знать систему команд компьютера;
- программист должен был быть инженером-электроником
- в случае аварийной ситуации компьютер останавливал работу и необходимо было искать ошибку в двоичном коде;
- трудно было изменять программу, т.к. использовалась безусловная адресация
- возникали проблемы в работе с внешними устройствами.

В итоге работа была неэффективной и дорогостоящей. Появилась задача снижения стоимости и повышение эффективности работы.

1) Это можно было достигнуть путем повышения надежности компьютера с точки зрения конструкции. В первую очередь путем более эффективного охлаждения с помощью кондиционирования, водяных систем и т.д.

2) Также нужно было решить проблема надежности с точки зрения сервисных функций. С появлением компьютеров первого поколения появились **сервисные управляющие программы**. Это класс специальных программ, которые создавались и поставлялись вместе с компьютером. Они предназначались для решения сервисных функций. В частности, решали функции загрузки информации с внешних носителей. Уже на уровне компьютеров первого поколения появилась часть оперативной памяти, называющаяся "Постоянное запоминающее устройство"или ROM (Read Only Memory). Так как эта часть памяти была энергонезависима, в ней можно было зафиксировать программу, которая сохранялась после выключения. Запуск устройства считывания с перфокарт был упрощен до установки точки входа на программу в ПЗУ и нажатия кнопки "Старт".

$\Rightarrow$ Такие специальные программы могли вводить информацию с разных устройств и, соответственно, выводить на разные устройства . Они могли запускать какие-то специальные функции тестирования устройства и компонентов компьютера.

3) Появились средства автоматизации программирования. Так как раньше программа представляла собой колонки цифр в которых найти ошибку было сложно, исправить ошибку можно было только с помощью переадресации, что делало программу слишком массивной. Появление *автокодов* или *языков ассемблера* позволило уйти от этой проблемы. Это средство нотации программы в виде мнемонических обозначений машинных команд и операндов. Что вызвало необходимость в сервисной программе, которая осуществляла бы перевод из нотации ассемблера в нотацию машинного кора. Такие программы назывались "трансляторы "компиляторы"или "ассемблеры". Соответственно, в классе сервисных управляющих программ появилась программа "компилятор ассемблера". Новый алгоритм ввода информации: на носителе была программа в нотации ассемблера, посредством загрузчика она размещалась в оперативной памяти. В оперативной памяти появлялся мнемонический код программы в виде ассемблера. Затем вызывалась сервисная программа "компилятор который транслировал информацию с asm в машинный код. После этого можно было запустить программу. $\Rightarrow$ На этом же этапе человеку все еще необходимо было работать у инженерного пульта, но у него уже появилось средство программирования в виде ассемблера.

Большой минус: ассемблер - язык машинно-зависимый. Жестко связан с системой команд конкретного компьютера. Программа на ассемблере одного компьютера не заработает на другом компьютере.

С компьютерами первого поколения связано появление первых языков высокого уровня. То есть языков, элементарными конструкциями которого были некоторые высокоуровневые конструкции, такие как цикл, условие, функция. Одним из первых языков, который стал активно использоваться, был язык Fortran (начало 50-х).
Первые компьютеры были ориентированы на решение расчетных задач.Был некоторый алгоритм, который связывал последовательность вычислений по некоторым формулам. Для языка Fortran появились некоторые компиляторы. Для него было создано множество сервисных функций, которые реализовывались в виде библиотек программ.

Режим использование компьютеров первого поколения. Это однопользовательский персональный режим, как сейчас. Первый режим работы – однопрограммный режим, то есть в памяти компьютера могла находиться и исполняться только одна обрабатываемая программа.

Компьютеры второго поколения

Появление компьютеров второго поколения связано с прорывом в области элементной базы, изобретением *транзистора* (конец 50-х – вторая половина 60-х гг..). Элементной базой компьютеров второго поколения стали диодно-транзисторные схемы. Соответственно, появились элементы, функционально идентичные лампам.
Преимущества:

- размер меньше на порядки;
- элементы впаивались в базу.

⇒ уменьшились габариты компьютера;
⇒ диоды и транзисторы потребляли меньше энергии;
⇒ ушла проблема контактных ошибок.

Соответственно, появление диодно-транзисторных технологий позволило создавать более компактные машины, существенно более функционально наполненные и более надежные, что повлияло на развитие программного обеспечения компьютера и режимов его использования.

С появлением компьютеров второго поколения связано развитие программного обеспечения, что позволило кардинально изменить применение и использование компьютера.

Пакетная обработка программ. В памяти компьютера постоянно находилась некоторая специальная программа "монитор пакета работающая циклически. Монитор пакета считывал последовательно перфокарты из колоды до карты, которая означала конец, а затем запускал внесенную программу. В случае успешного завершения управление возвращалось к монитору пакета, что снова запускало считывающее устройство. Каждая программа последовательно выполнялась. В колоде были специальные карты, которые интерпретировались пакетом как начало программы, задание для выполнения и конец. Появился язык управления заданиями.

⇒ С появлением пакетных систем у программистов пропала необходимость пользоваться инженерным пультом. Появилась должность "оператор компьютора".

В компьютерах второго поколения появилась аппаратная функция обработки ошибок. Впоследствии она стала называться "аппарат прерываний". Программа монитор получала информацию об ошибке, а затем выдавала ее пользователю.

Пакетный однопрограммный режим имел

плюсы

- программист больше не обслуживал компьютер
- не обязательно постоянное присутствие человека рядом с машиной

минусы

- все еще однопрограммный режим
- во время обмена с внешним устройством процессор простаивал, что снижало эффективность

Появление режима *мультипрограммирования* дало более эффективную загрузку процессора. Этот режим позволял единовременно находиться в обработке монитору и группе программ пользователя. Одна из этих программ могла исполняется центральным процессором, часть ожидала завершения обмена, остальные, готовые к исполнению, ждали своей очереди обработки программой "монитор".

Виртуализация. Появились средства создания и использования *виртуальных или логических ресурсов.*

Определение. Виртуальный или логический ресурс – некий объект, часть эксплуатационных характеристик которого реализуются программно. Пример – файловая система.

Виртуализация доступа к данным. Первый элемент такого типа был связан с организацией данных. Стали появляться первые системы организации данных, которые позволили унифицировать процесс хранения данных и доступа к ним. На компьютерах второго поколения появились файловые системы. Т.е. файл как именованный набор данных, имеющих унифицированный интерфейс доступа. Теперь не было необходимости знать, где находится нужный файл, на каком из запоминающих устройств. Доступ к содержимому можно было получить посредством унифицированного файлового интерфейса.

Операционные системы. На этапе компьютеров второго поколения появились первые операционные системы как комплекс программ, обеспечивающий обработку программ пользователей и распределение между ними ресурсов компьютера и вычислительной системы. Также развитие получили средства обработки ошибок. Аппарат прерывания – аппаратно-программные средства компьютера, предполагающие некоторую предопределенную реакцию на возникновение в компьютере или системе одного из заранее определенных событий(прерываний). Эти события могут инициироваться как внутренними схемами контроля процессора, например деление на ноль

или переполнение, так и внешними схемам, например возникновение ошибки при обмене с внешним устройством. В результате управление передается на некоторую определенную точку оперативной памяти, в которой находится программа, которая обработает это прерывание.

Один из лучших компьютеров второго поколения – БЭСМ-6, разработанный в институте точной механики и вычислительной техники под руководством академика Лебедева.

Компьютеры третьего поколения

Период развития пришелся на конец 60-х – 70-х гг. Элементная база основана на *интегральных схемах малой интеграции.* Появились такие элементы как сумматор или инвертор, которые могли выполнять более сложные функии.

С появлением интегральных схем малой интеграции функциональное наполнение элементной базы существенно возросло, что позволило создать более компактные, менее энергопотребляющие и более надежные компьютеры.

В период компьютеров третьего поколения можно отметить следующие качества.

- аппаратная унификация узлов и устройств совместимости различных моделей;
- появление «семейств» компьютеров, для преемственности программ компьютерами различных моделей снизу вверх. (любая программа, которая работает на компьютере младшей модели, обязана работать на компьютере более старшей модели, обратное необязательно);

⇒ Появилась возможность оптимизировать в рамках предприятия расходы на вычислительную технику.

Лекция 2. Поколения компьютеров

Компьютеры семейства IBM 360 и другие семейства

Наиболее ярким представителем семейств компьютеров было семейство IBM 360. Это 32-х разрядные компьютеры, обладавшие развитыми унифицированными внешними устройствами и развитым системным программным обеспечением. IBM 360 для своего времени обладал наиболее полным комплектом систем программного обеспечения: операционные системы, системы программирования, системы управления данными. Были ориентированы на решение задач программирования и экономических задач, бизнес-задач.

Другим примером семейств ЭВМ были компьютеры PDP 11. Это семейство 16-ти разрядных компьютеров, архитектура и программное обеспечение которых были ориентированы на автоматизацию научных исследований. К ним можно было подключать экспериментальное лабораторное оборудование. Аналогом семейства PDP 11 было Honeywell. Эта компания выпускала семейства компьютеров, которые предназначались для решения задач "реального времени". Они могли подключаться к производственному технологическому оборудованию.

⇒ С появлением новых поколений существенно расширяется круг применения компьютеров. Если компьютеры первого поколения были ориентированы на решение военных задач, с компьютерами второго поколения появились средства автоматизации и решение гражданских задач, то третье поколение было ориентировано на *решение задач из конкретных предметных областей.*

Унификация компонентов программного обеспечения позволила решить проблему переносимости программ. Если язык Fortran – попытка создания средств машинно-независимого программирования. На компьютерах третьего поколения вновь появились проблемы унификации. В этот период времени началась *стандартизация языков программирования.* Появились стандарты, которые оговаривали все нюансы реализации компиляторов для этих языков.

Революция в области операционных систем. Появилась операционная система UNIX.

1. Появление языка Си как языка, на котором написана операционная система UNIX. Впервые был разработан язык, не являвшийся машинно-ориентированным, средствами которого можно было эффективно программировать системное программное обеспечение. До этого программа, написанная на языке высокого

уровня, была на порядки менее производительной, чем программа на ассемблере. После появления языка Си ассемблер начал отмирать.

2. Впервые была разработана операционная система, которая была написана на машинно-независимом языке – операционная система UNIX.

3. UNIX стал стандартом в области интерфейсов, которые предоставляет операционная система. значительная часть стандартов POSIX (Portable Operating System Interface), стандарт программных интерфейсов для операционных систем, строится на интерфейсах UNIX.

4. Данная система получилась не просто экспериментальной, а стала широко распространенной операционной системой.

⇒ Решение проблемы унификации программного обеспечения.

В отечественной вычислительной технике, не смотря на наличие высококлассных специалистов в области архитектуры компьютеров и в области программного обеспечения, из-за проблем с элементной базой было принято решение закрыть отечественные проекты.

Четвертое поколение компьютеров. Микропроцессоры

Появились первые микропроцессоры,когда в одном корпусе могло быть реализовано процессорное устройство (интеграция). Первый микропроцессор Intel-4004 (1971г.) содержал 2250 элементарных единиц. Производство таких процессоров развивалось так, что в 1981г. в уже 32-битном микропроцессоре было 450 тыс. элементов.

В 1974 году в США появилась игрушка Altair-8800 – компьютер в разобранном виде. В 1975 году Бил Гейтс для этого компьютера разработал интерпретатор для языка Basic. Это был первый персональный компьютер четвертого поколения.

В 1976 г. Стив Ждобс и Стив Возник с использованием 8-разрядного микропроцессора сделали персональный компьютер Apple-1.

Первый ноутбук – компьютер Osborne 1 (1980г.).

Современные компьютеры относятся к компьютерам четвертого поколения. Основываются на использовании микропроцессорных технологий и элементной базе большой и сверхбольшой интеграции.

Итоги появления компьютеров четвертого поколения:

- Расширение областей практического применения компьютера. Расширилась область встроенных компьютеров, область управления технологическими процессами в режимах реального времени;

- Компьютер стал бытовым устройством, что произошло благодаря компаниям Microsoft и Apple.

⇒ На сегодняшний день имеется повсеместная компьютеризация технологических процессов, которая идет в сторону роботизации.

Суперкомпьютерные технологии

Сверхвысокая интеграция позволила по-другому посмотреть на суперкомпьютеры. Благодаря микропроцессорной элементной базе возможно строить компьютеры *сверхглубокого параллелизма.* Упрощаются проблемы энергопитания, теплоотвода и т.д.

В настоящее время наблюдается тенденция подключать систему охлаждения компьютера к системе отопления в суперкомпьютерных центрах.

Квантовые технологии используются для безопасного и надежного обеспечения передачи данных.

Вычислительные системы и их уровни

Определение. *Компьютер* – набор аппаратуры без программного обеспечения.

Определение. *Вычислительная система* – интеграция аппаратуры компьютера и программного обеспечения, функционирующая в единой системы и предназначенная для решения задач из конкретных предметных областей.

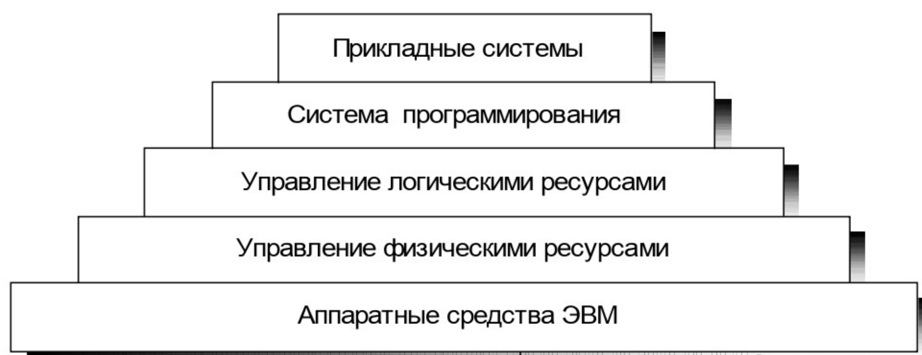

Рис. 2.1. Уровни вычислительной системы

Вычислительную систему можно представить в виде некоторой иерархии уровней. Самый нижний уровень – аппаратура компьютера, самый высокий уровень – прикладные системы, которые реализованы и функционируют в рамках данного компьютера.

ФАКУЛЬТЕТ ВЫЧИСЛИТЕЛЬНОЙ МАТЕМАТИКИ И КИБЕРНЕТИКИ МГУ ИМЕНИ М.В. ЛОМОНОСОВА

Эксплуатационные качества вычислительной системы определяются как свойствами аппаратуры, так и программных компонентов.

Каждый из уровней предоставляет для вышестоящих уровней некоторые интерфейсы.

Аппаратный уровень

Уровень аппаратуры компьютера представляется для всех вышестоящих уровней иерархии в виде совокупности двух типов сущностей:

1. система команд компьютера;
2. совокупность *физических ресурсов*, которые предоставляются на вышестоящие уровни.

Определение. Физический ресурс – аппаратный компонент компьютера, который обладает следующими характеристиками:

- правила программного использования, т.е. правила, по которым посредством машинных команд можно обратиться к этому устройству и выполнить с этим устройством определенные действие (пример - экран);
- производительностью или емкостью;
- степенью занятости или используемости и .тд.

Главный признак – правила программного использования.

⇒ Аппаратный уровень предоставляет на вышестоящие уровни вычислительной системы систему команд и аппаратные интерфейсы программного взаимодействия с физическими ресурсами.

Уровень управления физическими ресурсами

Данный уровень уже является программным. Имея первый аппаратный уровень и совокупность аппаратных интерфейсов управления физическими ресурсами, можно программировать эти ресурсы, но для этого необходимо знать все детали процесса управления.

⇒ Задача уровня управления физическими ресурсами заключается в систематизации и стандартизации правил программного использования физических ресурсов.

На этом уровне для каждого физического ресурса может присутствовать специализированная программа или программы, которая называется *драйвером физического ресурса.*

Физические ресурсы. Драйвер

Функция драйвера – предоставление на вышестоящие уровни унифицированных программных интерфейсов работы с каждым конкретным физическим устройством.

Определение. Драйвер физического устройства – программа, основанная на использовании команд управления конкретного физического устройства и предназначенная для организации работы с данным устройством.

Т.е. на данном уровне иерархии вычислительной системы обеспечивается корректное функционирование и использование физических ресурсов/устройств. Управление остается аппаратно-ориентированным.

Драйвер физического устройства скрывает от пользователя детальные элементы управления конкретным физическим устройством, а также он ориентирован на конкретные свойства устройства.

Например, внутри драйвера реализуется свой сценарий предобработки и диагностирования ошибки, а также принятие решения, является ли она фатальной.

Для одного физического устройства может быть два и более физических драйверов. Пример – магнитная лента. На ленте информация может размещаться в виде записей, это порция данных произвольного размера, ограниченная маркером начала и маркером конца, а может храниться в виде записей ограниченного размера. Соответственно один драйвер может работать с записями произвольного размера, а другой с записями фиксированного размера. Каждый из драйверов будет предоставлять физический аппаратно-ориентированный интерфейс управления устройством.

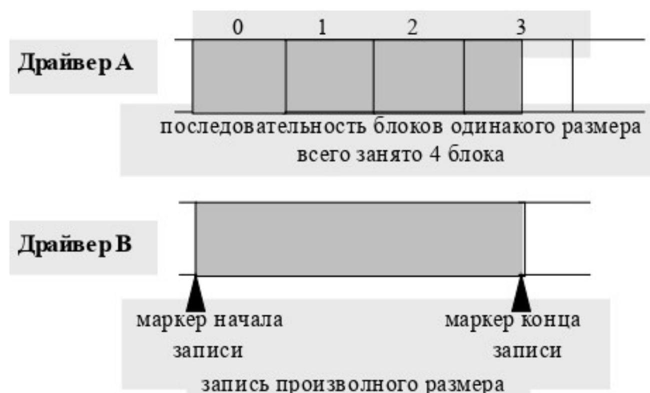

Рис. 2.2. Пример различных драйверов для магнитной ленты

Большой минус – должна использоваться модификация для перехода от одного физического устройства к другому.

ФАКУЛЬТЕТ ВЫЧИСЛИТЕЛЬНОЙ МАТЕМАТИКИ И КИБЕРНЕТИКИ МГУ ИМЕНИ М.В. ЛОМОНОСОВА

Управление виртуальными или логическими устройствами

Определение. *Виртуальное или логическое устройство/ресурс* – устройство вычислительной системы, некоторые эксплуатационные характеристики которого (возможно все) реализованы программно.
Одной из причин появления виртуальных устройств является создание унифицированного единого интерфейса взаимодействия с однотипными устройствами. То есть можно создать некоторый "драйвер виртуального диска определить для него программные интерфейсы, а далее все программное обеспечение будет обращаться к дискам через драйвер виртуального диска. Внутри этого драйвера виртуального диска связывается машинно-независимый запрос с запросом к конкретному физическому драйверу. ⇒ Больше нет необходимости знать, с каким именно диском мы работаем.

Определение. Драйвер виртуального ресурса – это программа, обеспечивающая существование и использование соответствующего ресурса.

Виртуальные ресурсы могут быть

- ресурсами, обобщающими какие-то характеристики какого-то класса реальных устройств (пример: драйвер диска, драйвер клавиатуры и т.д.);
- ресурсами, для которых не существует аппаратной реализации (пример: файловая система).

Лекция 3. Основы архитектуры вычислительной системы

Основы архитектуры вычислительной системы

Вычислительная система – совокупность аппаратных и программных средств, функционирующих как единая система и предназначенных для решения задач из конкретных областей.

Основа вычислительной системы – аппаратные средства ЭВМ, над этим уровнем расположены программные уровни, которые наполняют вычислительную систему возможностями и функциональным смыслом (см. Рис. 2.1).

Каждый уровень поставляет на вышестоящие уровни некоторые интерфейсы.

Любую вычислительную систему можно рассмотреть независимо с позиции каждого из уровней.

Аппаратный уровень представляется совокупностью физических ресурсов, которые имеются у данного компьютера, и системой команд. Физический ресурс – некий аппаратный компонент компьютера, обладающий правилами использования, т.е. это то, что можно программно использовать.

Управление физическими ресурсами вычислительной системы

Управление физическими ресурсами – первый программный уровень вычислительной системы. Представляется в виде совокупности драйверов физических ресурсов.

Драйвер физического устройства -– программа, основанная на использовании команд и средств управления конкретным физическим устройством и предназначенная для организации работы с данным устройством.

Цель появления драйвера – систематизация и стандартизация организации работы с физическими ресурсами в вычислительной системе. Каждый драйвер предоставляет для использования некоторый программный интерфейс. Он семантически или функционально во многом идентичен функциям, которые предоставляет аппаратные средства управления конкретным устройством (программный интерфейс).

Каждый драйвер физического устройства предоставляет программный интерфейс доступа к физическому ресурсу, с другой стороны скрывает некоторые детали этих физических ресурсов внутри себя от пользователей.

Драйвер физического устройства ориентирован на управление конкретным физическим устройством, следовательно программный интерфейс тоже ориентирован

на управление именно этим устройством. Например, для струйного и матричного принтера различаются те фрагменты программы, которые переводят "прикладные"обращения в обращения к физическому устройству посредством драйвера.

На прошлой лекции было сказано, что для одного физического устройства может быть два и более физических драйвера.

Управление виртуальными (логическими) ресурсами

Создание виртуальных устройств было направлено на решение проблемы разнообразия внешних устройств. Упрощение возможности использования в программе различных устройств одного типа, обобщение характеристик и свойств существующих устройств.

Определение. Виртуальный ресурс – это такой ресурс вычислительной системы, в котором часть эксплуатационных характеристик или возможностей (возможно все) реализованы программным образом.

Основная функция – унификация доступа к устройствам одного типа. Пример: виртуальный принтер, виртуальный диск.

Виртуальный диск предоставляет на пользовательский уровень некоторый унифицированный, машинно-независимый интерфейс работы с диском, а внутри этого драйвера виртуального диска программа разбирается, к какому реальному физическому устройству идет обращение. Происходит перевод запроса к виртуальному диску в запрос к конкретному физическому диску, с которым будет осуществляться работа.

Определение. Драйвер виртуального (логического) ресурса – программа, обеспечивающая существование и использование соответствующего ресурса.

Виртуальные ресурсы, для которых нет аппаратного аналога

Следующий шаг развития виртуальных ресурсов – появление класса виртуальных ресурсов, для которых нет прямых аппаратных аналогов. Одним из первых ресурсов такого типа являются *файлы* и *файловая система*.

Определение. Файл – именованный набор данных, который предоставляет возможность доступа к данным посредством имени файла и использования программных интерфейсов доступа.

Рассмотрим схематичный пример организации работы с внешними устройствами. Есть файловая система и интерфейс взаимодействия с файловой системой. При обращении к файлу, на самом деле обращение осуществляется к драйверу файловой системы. Драйвер файловой системы преобразует запрос к драйверу виртуального

диска. Затем драйвер виртуального диска осуществит трансляцию полученного запроса в виде обращения к тому или иному драйверу физического ресурса, который ассоциирован с этим запросом. На уровне управления физических устройств, запрос идет либо к конкретному дисковому устройству, либо к оперативной памяти.

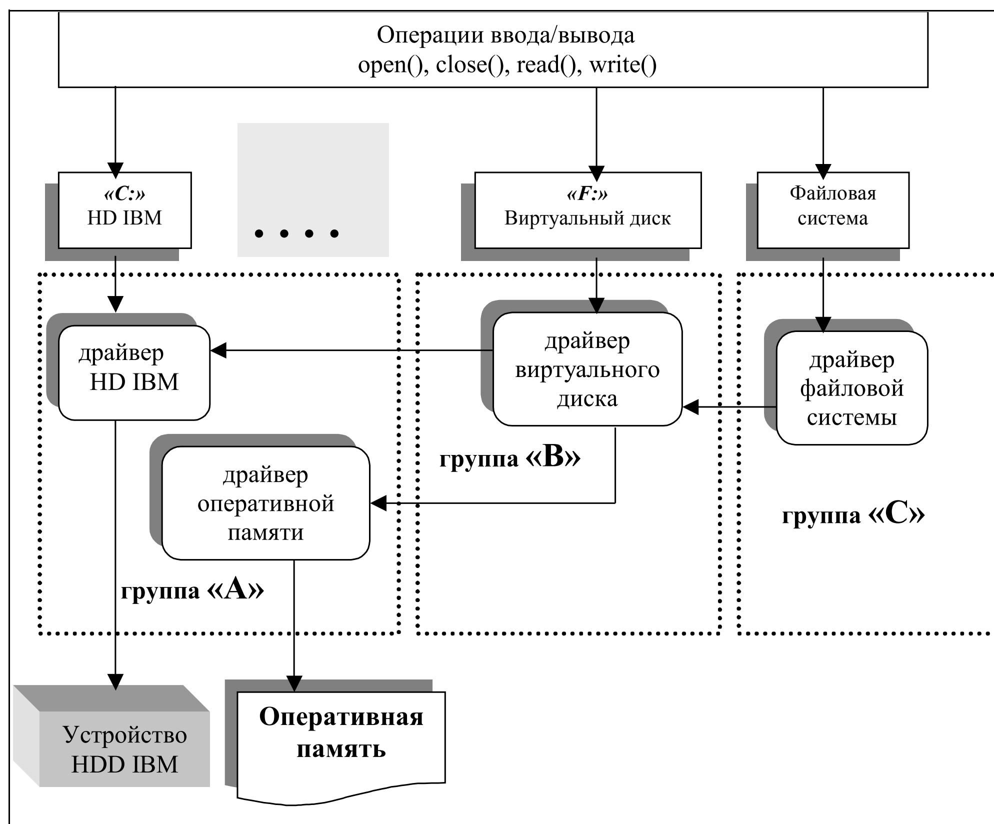

Рис. 3.1. Пример организации работы с внешними устройствами

1. Система драйверов на уровне управления физическими устройствами и на уровне управления виртуальными ресурсами может быть многоуровневой. На каждом из уровней драйвера могут быть построены иерархически.

2. Запрос, который идет к виртуальному устройству, не обязательно будет реализован в последовательности обращений к драйверу физического устройства (например, принтеру), а к драйверу файловой системы, и затем к конкретному физическому устройству (диску).

Итоги:

- Уровень управления физическими ресурсами и уровень управления виртуальными ресурсами являются программными и составляют операционную систему.

ФАКУЛЬТЕТ ВЫЧИСЛИТЕЛЬНОЙ МАТЕМАТИКИ И КИБЕРНЕТИКИ МГУ ИМЕНИ М.В. ЛОМОНОСОВА

- Под ресурсами вычислительной системы понимают совокупность физических и виртуальных ресурсов.

⇒ Определение. *Операционная система* – комплекс программ, обеспечивающий существование, доступ и управление ресурсами вычислительной системы. Это то, что обеспечивает существование всей совокупности драйверов.

Уровень систем программирования

Определение *Система программирование* – комплекс программ, предназначенный для поддержания и реализации жизненного цикла создаваемой программы.

Уровень системы программирования обеспечивает поддержание этапов жизни программы.

Жизненный цикл программы можно представить в виде совокупности шагов. Эти шаги могут как итерационные, так и последовательные.

1. *Проектирование программного комплекса.* Один из самых ответственных этапов. На этом этапе происходит Формализация и уточнение постановки задачи; оценивается круг задач, их ресурсоемкость, выбирается аппаратной платформы; выбор методов решения задачи, использование математических методов для решения задачи; выбор инструментальных средств программирования, затем проектируется реализация и выбирается спецификации архитектуры. Архитектура системы – это то, из каких модулей будет состоять система, какие интерфейсы будут предоставлять эти модули, спецификация этого взаимодействия.

2. *Кодирование.* Имеем некоторую инструментальную систему. На этом этапе важна строгость инструментальных средств программирования.

3. *Тестирование и отладка.*

   - Тестирование – это проверка соответствия функционирования разрабатываемого программного обеспечения спецификациям проекта на некоторых наборах тестовых нагрузок. Тестирование подтверждает работоспособность программы на некотором тестовом наборе.
   - Отладка – это поиск и исправление ошибок, выявленных при тестировании.

4. *Ввод программной системы в эксплуатацию (внедрение) и сопровождение.* Подготовка эксплуатационной документации, организация методики сопровождения и эксплуатации системы и т.д..

Современный подход – подход, основанный на том, что на каждом шагу жизненного цикла, где возможно внести ошибку, ищутся решения, позволяющие автоматизировать соответствующие действие и осуществлять переход от одного этапа жизненного цикла к другому "непрерывно". Схема уточняется и детализируется до тех пор, пока она не превратится в программу. В этим случае мы уйдем от проблем появления ошибок на каждом из шагов и различной интерпретации проектированного.

Временная иерархия интерпретации понятия "системы программирования":

- Начало 50-х годов XX – века.

  Система программирования или система автоматизации программирования включала в себя ассемблер (или автокод) и загрузчик, позже появились библиотеки стандартных программ и макрогенераторов.

- Середина 50-х – начало 60-х годов XX – века.

  Появление и распространение языков программирования высокого уровня (Фортран, Алгол-60, Кобол и др.). Переход от команд-цифр к высокоуровневым программам. Формирование концепций модульного программирования.

- Середина 60-х годов – начало 90-х XX – века.

  Развитие интерактивных и персональных систем, появление и развитие языков объектно-ориентированного программирования. Появление средств, которые позволяли создавать программные объекты, скрывающие организацию данных и их обработку от пользователя. Появление первых промышленных систем программирования, основанных на языках высокого уровня. Появился язык Си, правда, это скорее машинно-независимый ассемблер.

- 90-е XX – века – настоящее время.

  Появление систем программирования, в которых есть комплексные системы. Последовательное взаимозаменение программ. Появление промышленных средств автоматизации проектирования программного обеспечения, CASE-средств (Computer-Aided Software/System Engineering), унифицированного языка моделирования UML.

  В настоящее время это CASE-системы. Системы, в которых делается попытка интеграции шагов жизненного цикла и попытка представления движения по жизненным циклам как единого целого. Проект переходит в программу.

На уровне систем программирования пользователю доступны средства программирования на уровне операционной системы и программные средства поддержки жизненного цикла программ, такие как компилятор, редакторы связей, загрузчики, библиотеки и прочее.

Прикладные системы

Определение. *Прикладная система система* – программная система, ориентированная на решение или автоматизацию решения задач из конкретной предметной области.

Уровень прикладных систем – это целевой уровень, ради которого строится вычислительная система. На этом уровне реализуются средства формализации взаимодействия с программными системами, используемыми для решения конкретных прикладных задач.

Выводы. Список терминологии

Представление уровней вычислительной системы для пользователя:

Прикладные системы	+ набор функциональных средств прикладной системы.
Системы программирования	+ трансляторы языков высокого уровня, библиотеки...
Управление виртуальными устройствами	+ интерфейсы драйверов виртуальных устройств.
Управление физическими устройствами	+ интерфейсы драйверов физических ресурсов.
Аппаратные средства	Система команд, аппаратные интерфейсы программного управления физическими устройствами

Рис. 3.2. Пользователь и уровни структурной организации ВС

К данному моменту мы определили следующие базовые понятия:

- Вычислительная система
- Физические ресурсы (устройства), его основные характеристики
- Драйвер физического устройства

- Логические или виртуальные ресурсы (устройства)
- Драйвер логического/виртуального ресурса
- Ресурсы вычислительной системы
- Операционная система
- Жизненный цикл программы в вычислительной системе
- Система программирования
- Прикладная система

Основы архитектуры компьютера. Компьютер фон Неймана

Рассмотрим компьютер в контексте системных взаимосвязей и организации вычислительной системы на уровне операционной системы.

Одним из первых компьютеров можно называть компьютер ENIAC, разработанный в Пенсильванском университете под руководством двух инженеров Мокли и Эккерта.

Следующим шагом развития концепции компьютера ENIAC стал проект EDVAC (Electronic Discrete Variable Automatic Computer – Электронный Компьютер Дискретных Переменных). Фон Нейман по результатам работы Мокли и Эккерта подготовил доклад "Предварительный доклад о компьютере EDVAC в котором были сформулированы некоторые базовые концепции.

Принципы построения компьютера фон Неймана:

1. *Принцип двоичного кодирования.* Вся информация, поступающая и обрабатываемая внутри компьютера, представляется в виде двоичной системы.

2. *Принцип программного управления.* Декларировалось, что программа представляется в виде совокупности команд, в которых указываются операнды и операции над операндами, и данных, с которыми связываются операнды. Выполнение программы – это последовательное выполнение команд, составляющих программу. Декларировалось, что в компьютере есть процессор, в функции которого входит выполнение команд программы в заданной последовательноси.

   ⇒ То есть это принцип последовательного однопроцессорного функционирования компьютера.

ФАКУЛЬТЕТ
ВЫЧИСЛИТЕЛЬНОЙ МАТЕМАТИКИ И
КИБЕРНЕТИКИ МГУ ИМЕНИ М.В.
ЛОМОНОСОВА

3. *Принцип хранимой программы.* Программа и все данные, которые используются в программе, хранятся единообразно в устройстве памяти. Хранение команд и данных осуществляется единообразно, и идентификация того, является ли информация, поступающая в процессор, командой или данными, определяется в момент выбора и исполнения каждой команды.

Принцип фон Неймана предполагает, что компьютер состоит минимум из трех функциональных узлов: оперативная память, процессор, внешнее устройство.

Определение. *Оперативная память* – устройство в памяти, в котором может располагаться исполняемая программа.

Определение. *Процессор* – аппаратный модуль компьютера, который обеспечивает выполнение команд программы и контроль за исполнением этих программ.

Процессор содержит два логических модуля:

- модуль устройства управления, обеспечивающий последовательность выбора команд на исполнение, контроль корректности команд (код операции из диапазона реализованных на данном процессоре, операнды имеют ссылки на существующие операнды), в устройстве управление осуществляется выборка операндов;
- модуль арифметико-логического устройства, осуществляющий реализацию команд, предполагающих арифметическую или логическую обработку содержимого операндов.

Внешним устройством на момент создания компьютера EDVAC являлись магнитная лента, устройство считывания с перфокарт и перфолент, устройство печати, устройство вывода информации на перфокарты/перфоленту.

Лекция 4. Основы компьютерной архитектуры

Принципы фон Неймана

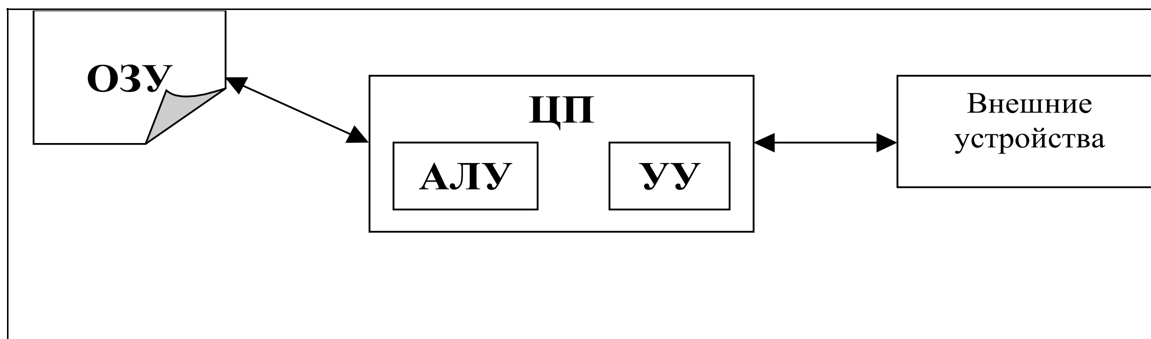

Рис. 4.1. Структура, основные компоненты компьютера фон Неймана

Принципы построения компьютера фон Неймана:

1. Принцип двоичного кодирования
2. Принцип программного управления
3. Принцип хранимой программы

Рассмотрим отличия современных компьютеров от компьютера фон Неймана. Принцип программного управления – фактически последовательное выполнение программы, в настоящее время компьютеры параллельные, единовременно обрабатывают две и более команды. База для представления данных в компьютере не обязательно мо;ет быть двоичной. Был разработан принцип троичного кодирования инженером Н.П. Брусенцовым. Принцип хранимой программы тоже не соблюдается, т.к. в современных компьютерах команды и операнды представляются по-разному для того, чтобы нельзя было несанкционированно исполнять данные как команды. ⇒ на сегодняшний день не используются компьютеры фон Неймана.

Далее рассмотрим особенности и системные аспекты каждого из компонент компьютера фон Неймана.

Оперативное запоминающее устройство

ОЗУ – это устройство памяти компьютера в котором может располагаться исполняемая программа.

Оперативное запоминающее устройство структурно состоит из последовательности ячеек памяти.

Определение. *Ячейка памяти* – устройство для хранения информации, имеющее предопределенное фиксированное количество разрядов, предопределенную длину в двоичных разрядах.

Все ячейки памяти имеют уникальные имена, которые называются *адресами*. Обычно адресация памяти представляет собой нумерацию ячеек памяти. Адрес является некоторой функцией от номера ячейки.

Системы адресации памяти:

- если адресуется вся ячейка или все машинное слово, тогда адресом является номер ячейки;
- если адресуется каждый байт, тогда машинные слова или ячейки адресуются кратны длине машинного слова в байтах.

Структура ячейки памяти. Ячейка памяти состоит из двух полей.

1. Поле машинного слова – это поле, в котором может размещаться программно-изменяемая информация. В машинном слове размещаются машинные команды и операнды, данные. Машинное слово имеет фиксированную длину, кратную количеству байтов, которые помещаются в машинном слове. Длина машинного слова является ключевой характеристикой оперативного запоминающего устройства.

2. Поле тэга – это поле, которое используется для решения функций контроля при работе с оперативной памяти. Может иметь произвольную длину или отсутствовать, если нет контроля.

Использование полей тэга

1. Контроль целостности данных. Например, контроль четности. Предполагается, что есть машинное слово длиной 16 бит. Поле тэга состоит из 1 бита. При записи информации в соответствующее машинное слово, аппаратная система контроля оперативной памяти подсчитывает количество двоичных единиц в записываемом коде. Если это количество нечетное, в разряд тэга добавляется единица, иначе добавляется нолик. Таким образом, в ячейке сохраняется четное количество единичек. При считывании информации из машинного слова происходит обратное действие. Осуществляется подсчет количества единичек, считанных в машинном слове и сопоставляется разряд из тэга. Если логика нарушена(второй пример), то схема контроля фиксирует ошибку.

Более сложные схемы контролируют содержимое кода, исправляют одинарные ошибки и обнаруживают двойные.

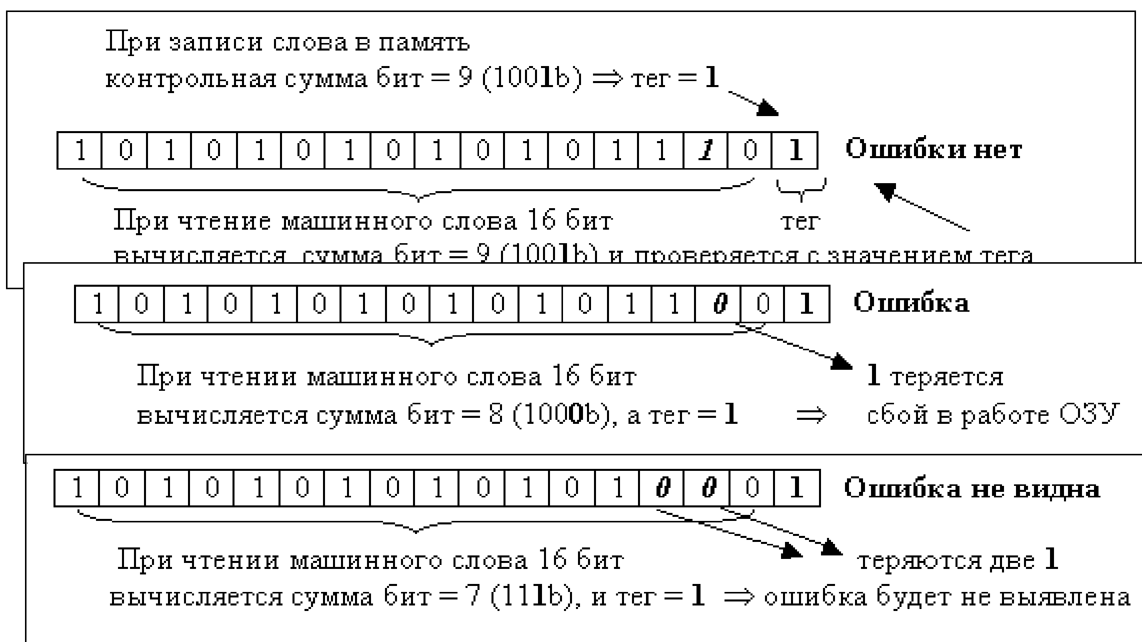

Рис. 4.2. Пример контроля за целостностью данных по четности

2. Контроль доступа к командам/данным. Тэг может использоваться для осуществления контроля за тем, обращаемся мы к команде или к данным. Добавляется еще один разряд в поле тэга. Если в нем единичка, значит, что в машинном слове находится команда.

3. Контроль доступа к машинным типам данных. Более развитые архитектуры могут осуществлять контроль доступа к машинным типам данным. При выполнении команды осуществляется контроль, чтобы тэги у операндов совпадали.

Вычислительная система объединяет в себе компоненты различной производительности. Скорость процессора на порядки превышает скорость доступа к оперативной памяти. В аппаратной части компьютера есть совокупность решений, которые позволяют сгладить дисбаланс в производительности этих компонентов.

Определение. Производительность оперативной памяти – скорость доступа процессора к данным, размещенным в ОЗУ. Производительность оперативной памяти характеризуют два показателя:

1. время доступа (access time, t_{access}) - время между запросом на чтение информации из оперативной памяти и получением запрошенной информации;

2. длительность цикла памяти (cycle time - t_{cycle}) - минимальное время между двумя запросами к оперативной памяти, между началом текущего запроса и началом обработки последующего обращения.

$$t_{cycle} > t_{access}$$

Время доступа больше, потому что операция чтения состоит из чтения и записи. Разница между временем доступа и времени цикла идет на регенерацию, восстановление прочитанной информации. Получается, что характеристика регенерации снижает системную производительность.

Расслоение памяти

Одним из решений, которые предназначены для организации сглаживания проблемы длительной регенерации – это использование памяти с расслоением.

Все физическое адресное пространство представляется в виде K независимых банков памяти. Нулевой адрес находится в нулевом банке, первый – в первом банке, (K-1)-й находится адрес в (K-1)ом банке, а K-й адрес находится в нулевом банке. Получается, что все адреса распределены спиралью между банками.

Пусть $K = 2^L$. Возьмем представление адреса. Крайние правые L разрядов будут содержать номер банка (см. рис. 4.3).

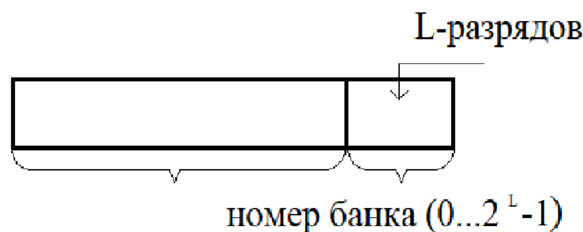

Рис. 4.3. Принцип расслоения памяти

Каждый банк может работать независимо. Расслоение памяти может быть реализовано в разных моделях.

- Рассмотрим модель с общим контроллером доступа к памяти. Одно устройство управляет всеми K банками (см. рис. 4.4). Если идет последовательное обращение к адресам, мы уйдем от времени цикла. Считали ячейку A, отдали ячейку A, после этого банк, в котором находится ячейка A, блокируется на регенерацию. При считывании ячейки A+1 мы попадаем в следующий банк, который не заблокирован. То есть регенерация идет в каждом банке.

- Если каждый банк имеет свой контроллер, тогда последовательные запросы длиной не более K обрабатываются практически параллельно (см. рис. 4.5).

ФАКУЛЬТЕТ ВЫЧИСЛИТЕЛЬНОЙ МАТЕМАТИКИ И КИБЕРНЕТИКИ МГУ ИМЕНИ М.В. ЛОМОНОСОВА

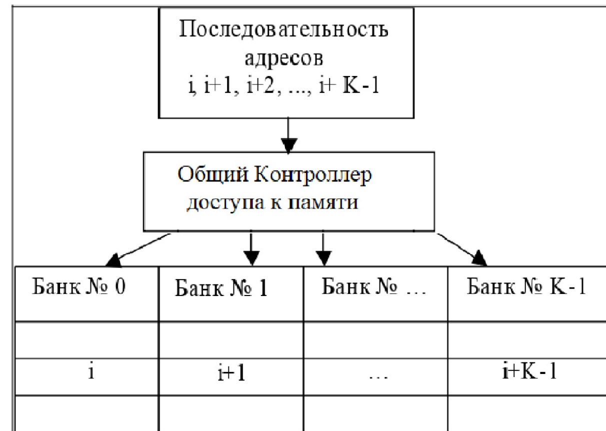

Рис. 4.4. Модель с общим контроллером доступа к памяти

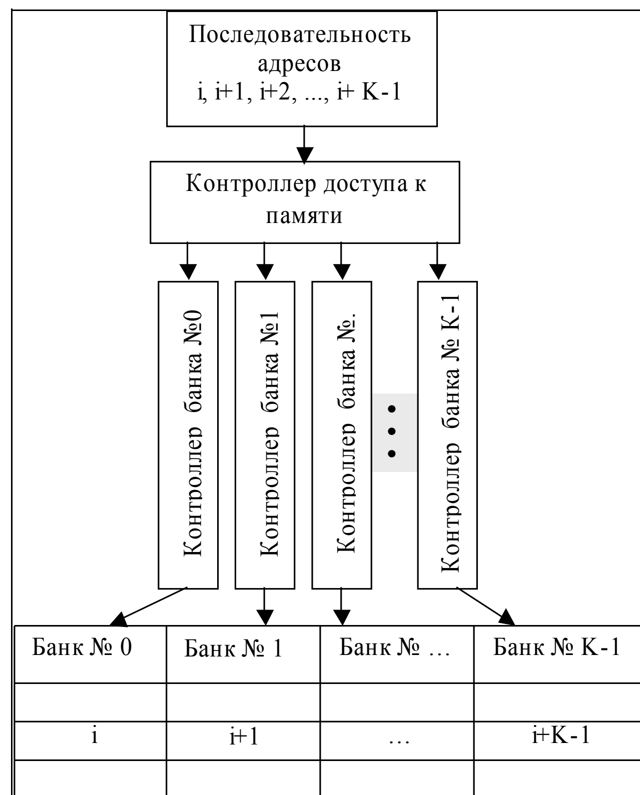

Рис. 4.5. Модель с K-1 контроллерами доступа к памяти

Сравним время доступа для считывания трех адресов.

Для случая без расслоения затраченное время будет равно $3{*}t_{cycle}$. Для случая общего управление затраченное время будет $3{*}t_{access}$.

Для случая, когда каждый банк имеет свой контроллер, время будет порядка времени доступа t_{access}, т.к. банки могут быть запущены параллельно.

ФАКУЛЬТЕТ ВЫЧИСЛИТЕЛЬНОЙ МАТЕМАТИКИ И КИБЕРНЕТИКИ МГУ ИМЕНИ М.В. ЛОМОНОСОВА

Центральный процессор. Регистры

Логическая структура процессора состоит из устройства управления, арифметико-логическое и два устройства памяти: регистровая память и кэш первого уровня.

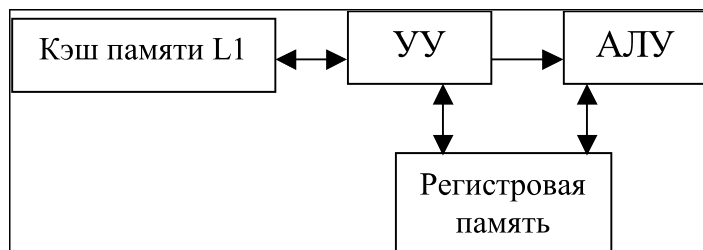

Рис. 4.6. Логическая структура процессора

Устройство управления (control unit)– координирует выполнение команд программы процессором. Арифметико-логическое устройство (arithmetic/logic unit) – обеспечивает выполнение команд, предусматривающих арифметическую или логическую обработку операндов.

Регистровая память – это память, которая реализована в процессоре. Рассмотрим два типа.

1. *Регистры общего назначения (РОН)*, те регистры, которые используются в командах программ пользователя. Регистры общего назначения бывают типизированные или целочисленные. РОН появились как средства сглаживания дисбаланса между скоростью оперативной памяти и скоростью процессора за счет того, что наиболее часто используемые операнды можно разместить на регистрах. Таким образом сокращается количество обращений в медленную оперативную память.

2. *Специальные регистры* используются схемами контроля и управления работой процессора.

   - Указатель команд является наиболее известным. Для реализации команд безусловного перехода, в него устройство управления записывает исполнительный адрес команды безусловной передачи управления.
   - Указатель стека, если есть стековые операции. Указатель стека устанавливается адрес, где находится стек.
   - и т.д.

Рабочий цикл процессора

Рассмотрим модельный пример того, как организуется работа процессора. Пусть в оперативной памяти используется пословная адресация. Пусть имеется счетчик команд, в котором имеется какое-то содержимое. Рассмотрим действия процессора.

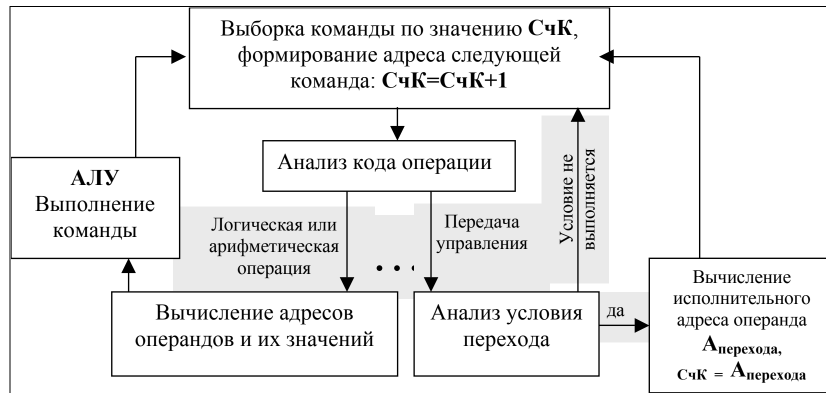

Рис. 4.7. Упрощенный алгоритм рабочего цикла процессора

1. Осуществляется выборка машинного слова из оперативной памяти, по адресу, который находится в счетчике команд. Это слово помещается на некоторый специальный регистр процессора. Счетчик команд автоматически увеличивается на единичку.

2. Начинает работать устройство управления. Анализ кода операции, которую считали. Если код некорректный, аппаратура фиксирует ошибку. Если код операции – арифметическая или логическая операция, происходит вычисление исполнительных адресов операндов, осуществляется получение значения этих операндов, которые помещаются на специальные регистры процессора. Управление передается в арифметико-логическое устройство, в котором эта команда исполняется. Если не фиксируется ошибка, возвращается на начальный этап, где снова считываем команду.

3. Снова анализ кода операции. Если это команда передачи управления, проводится анализ условия перехода. Если условие не выполняется, возвращается на самое начало. При выполнении условия перехода, вычисляется исполнительный адрес операнда и счетчику команд присваивается данный исполнительный адрес. Затем возвращается на первый этап

Кэш-память

Кэш-память первого уровня – это специализированное устройство памяти внутри процессора. Она реализована как регистровая память. Регистры и кэш первого уровня работают в темпе процессора.

Определение. *Кэш* – это специализированная память, размещенная в процессоре, которая организована в виде последовательности блоков фиксированного размера. Размер каждого блока кэша является степенью двух.

Между кэшем и оперативной памятью может осуществляться обмен. Обмен происходит порциями данных размером равным размеру блока кэша.

С каждым блоком кэша ассоциирован *тэг* – поле служебной информации. В тэге может быть указана информация о соответствии содержимого блока кэша виртуальному блоку оперативной памяти. В тэге может быть информация о том, свободен ли блок, производилась в блоке кэша модификация или нет, и т.д.

Когда процессору нужно считать некоторое машинное слово из оперативной памяти, подается команда чтения, вычисляется исполнительный адрес, затем по тэгам блоков кэша происходит параллельный поиск, если ли блок кэша, который содержит этот адрес или нет.

Если находится такой блок, фиксируется ситуация попадания. То есть необходимая информация находится без обращения в оперативную память.

Если же фиксируется ситуация промаха, начинаются действия с содержимым кэша. Обновляется содержимое кэша, т.е. в какой-то блок будет помещен виртуальный блок оперативной памяти, в котором есть искомый адрес. Для этого нужно вытеснить какой-то блок кэша. Выбирается такой блок для вытеснения либо случайно, либо выбирается наименее популярный блок.

Вытеснение кэша данных:

- сквозное кэширование характеризуется тем, что после изменения информации в блоке кэша, блок скидывается в оперативную память, т.е. соблюдается соответствие содержимого оперативной памяти кэшу.

- кэширование с обратной связью устроено так, что если был смодифицирован блок кэша, в тэге устанавливается признак того, что блок модифицировался. Если блок, выбранный для вытеснения модифицировался, он сбрасывается в оперативную память, а затем освобождается.

Основная концепция кэширования. Кэш не будет работать, если поток адресов,

который формируется для обращения в память как для получения команд, так и для операндов, является случайным. Если он случайный, на каждый адрес с большой вероятностью будет проведен процесс вытеснения.

Кэш работает за счет того, что действует принцип локализации формирования потока запросов к адресам.

Так как взаимодействие между памятью и кэшем происходит блоками, появляется параллельность. При наличии одновременно кэша и расслоения памяти обмен блоками происходит параллельно с максимальной скоростью.

⇒ Два средства сглаживания дисбаланса работы медленной оперативной памяти и быстрого процессора – это расслоение памяти, кэш и их функционирование в системе.

Лекция 5. Иерархия памяти

Оперативное запоминающее устройство, центральный процессор и оперативная память

Одной из проблем, в контексте которой обсуждалось оперативное запоминающее устройство, было различие в производительностях процессора и оперативной памяти. Средствами сглаживания этого дисбаланса являются

1. *Регистры общего назначения.* В процессоре есть сверхоперативная память, которая работает в темпе процессора, и те операнды, которые часто используются, можно разместить на регистрах общего назначения. Таким образом сокращается количество реальных обращений процессора в медленную оперативную память.

2. *Расслоение памяти* дает возможность единовременного обращения группы запросов к оперативной память. Физическая память разделяется на некоторое количество независимых устройств, которые могут работать практически параллельно. При обработке последовательных обращений к памяти, можно использовать независимость банков памяти. В случае использования модели расслоения памяти с K контроллерами, производительность памяти увеличивается в K раз.

Центральный процессор

Центральный процессор может иметь внутри себя некоторую специально организованную регистровую память, которая называется кэш первого уровня. Определение
Кэш – это некоторый буфер фиксированного размера, который разделен на блоки, где каждый блок тоже фиксированного размера. Информация между оперативной памятью и кэшем может перемещаться порциями данных размером в блок кэша.

Для организации обмена между оперативной памятью и кэшем (процессором) вся физическая оперативная память компьютера разделяется на порции данных, равных размеру этого кэша. По любому адресу физической памяти можно определить номер виртуального блока памяти, на которые она поделена.

Каждый блок имеет тэг с информацией о блоке.

⇒ Основная задача кэша – снизить количество реальных обращений от быстрого процессора в медленную оперативную память за счет того, что адреса команд и операнды зависают в кэше.

Поток обращений в оперативную память, поток адресов имеет свойство некоторой локализации. Они и помещаются в кэш.

Взаимодействие между процессором и оперативной памятью, если процессор имеет кэш первого уровня, осуществляется кэшевыми блоками. В случае наличия расслоения памяти взаимодействие идет параллельно. То есть наличие одновременно кэша и расслоения памяти решает две проблемы:

1. параллельность, банки расслоенной памяти могут работать параллельно;

2. сокращение количества реальных обращений в оперативную память.

Аппарат прерывания

Аппаратура компьютера должна уметь фиксировать программные ошибки, иначе результат работы программы будет недетерминированный. Схема контроля компьютера может фиксировать аппаратные ошибки.

Первые компьютеры останавливали свою работу при обнаружении ошибки. Программист должен был искать ошибку, пользуясь инженерным пультом. На сегодняшний день вычислительная система работает автоматически, что означает, что возникновение ошибок должно обрабатываться автоматически без участия оператора. Для решения этой задачи на современных компьютерах есть *аппарат прерывания.*

При выполнении программы может произойти возникновение некоторых исключительных или критических событий (сломалось устройство, произошло деление на ноль). При разработке компьютера фиксируется некоторый набор событий, при возникновении которых в вычислительной системе предусмотрена некоторая предопределенная последовательность действий из двух этапов.

- *Аппаратный этап.* Аппаратура компьютера реагирует на возникновение определенного события и автоматически, без участия программы и оператора, выполняет некоторую предопределенную последовательность действий.

- *Программный этап.* После предварительных действий аппаратура передает управление на некоторую предопределенную точку оперативной памяти своего оперативного запоминающего устройства в предположении, что с этой точки находится программа, которая будет обрабатывать это событие.

Определение. Прерывание – событие в компьютере, при возникновении которого в процессоре происходит предопределенная последовательность действий. Определение. Обработка прерывания – реакция вычислительной системы на возникновение прерывания.

ФАКУЛЬТЕТ
ВЫЧИСЛИТЕЛЬНОЙ МАТЕМАТИКИ И
КИБЕРНЕТИКИ МГУ ИМЕНИ М.В.
ЛОМОНОСОВА

Все прерывания можно разделить на две группы:

1. внутренние прерывания – возникают в схемах контроля работы процессора. Примеры внутренних прерываний: произошло переполнение (overflow), деление на ноль, обращение в несуществующую область памяти.

2. внешние прерывания – инициируются в результате взаимодействия процессора с внешними устройствами. Прерывания не всегда связаны с ошибкой. Сигнал о прерывании может быть послан контроллером в результате успешного обмена с жестким диском для того, чтобы уведомить центральный процессор об успешном завершении обмена.

Схема обработки прерываний

Рассмотрим модельную упрощенную схему обработки прерывания.

При возникновении прерывания первая реакция в вычислительной системе – реакция аппаратуры. Начинается аппаратный этап обработки прерывания.

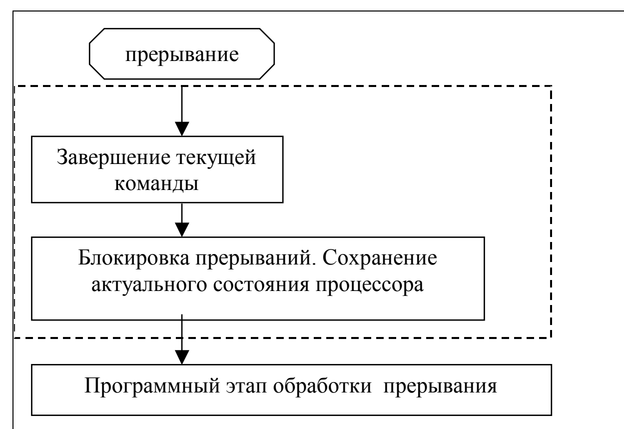

Рис. 5.1. Схема аппаратного этапа обработки прерываний

Рассмотрим алгоритм действий в случае внешнего прерывания.

1. Завершается выполнение текущей команды. Процессор завершает машинную команду, которая начала обрабатываться. Свойство того, что команда не может быть прервана внешним прерыванием – атомарность команды.

2. Процессор переходит в режим блокировки прерывания. При этом режиме обработка других возникающих в системе прерываний будет приостановлена. Запросы на такие прерывания могут буферизоваться. Это делается для того, чтобы точка прерывания не потерялась.

3. Аппаратура процессора осуществляет сохранение актуального состояния процессора. Аппаратно сохраняется некоторая совокупность параметров работы процессора, характеризующая точку возникновения прерывания. Эта информация помещается в некоторый специальный регистровый буфер. Точку возникновения прерывания в программе характеризует счетчик команд, состояние части регистров общего назначения, режим работы процессора. Эта минимальная информация нужна для того, чтобы после запуска программного этапа обработки прерывания и после обработки вернуться в точку прерывания и запустить программу с прерванного места.

4. Управление передается на некоторую предопределенную точку оперативной памяти. В счетчик команд автоматически записывается адрес. Это адрес зашивают в аппаратуру при производстве процессора. Предполагается, что по этому адресу находится программа, которая обработает возникшее прерывания.

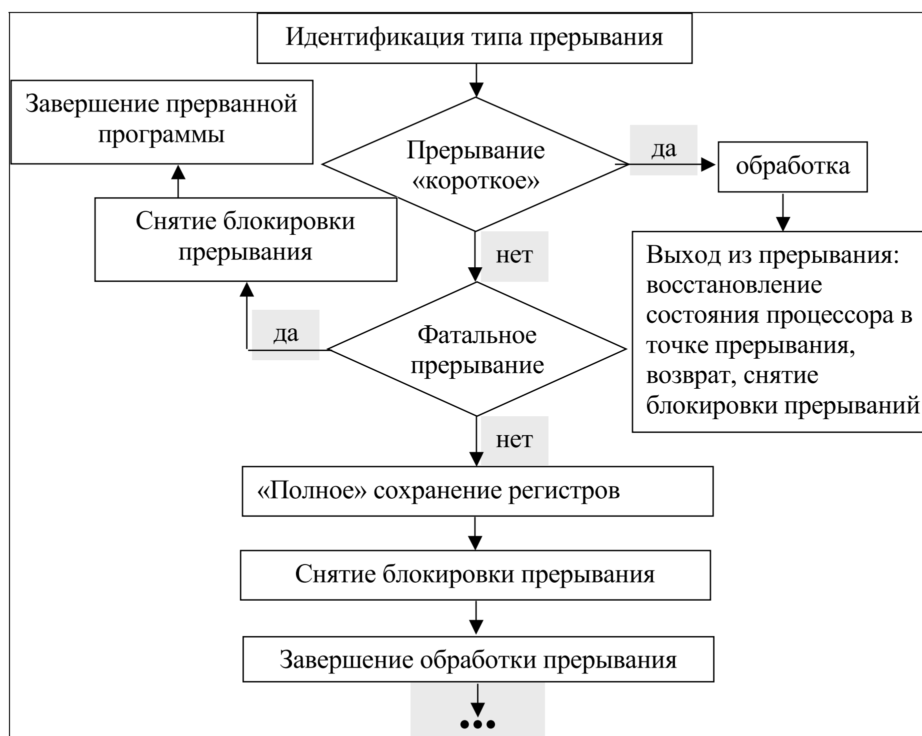

Рис. 5.2. Схема программного этапа обработки прерывания

5. После попадания на точку входа в программу обработки прерывания происходит анализ этого прерывания, оценивается тип прерывания. Запускается программа операционной системы.

   - Если прерывание является коротким, оно обрабатывается, затем осуществ-

ФАКУЛЬТЕТ ВЫЧИСЛИТЕЛЬНОЙ МАТЕМАТИКИ И КИБЕРНЕТИКИ МГУ ИМЕНИ М.В. ЛОМОНОСОВА

ляется выход из прерывания. По команде выхода из прерывания восстанавливается состояние процессора в точке прихода прерывания, снимается блокировка прерывания, управление передается в точку прерывания.

- Если прерывание не является коротким, продолжается определение типа прерывания.
  - Если прерывание является длинным фатальным (деление на ноль, обращение в несуществующую память), снимается блокировка, запускается процесс завершения прерванной программы, выдача данных, связанных с аварийным завершением программы.
  - Если длинное прерывание не является фатальным, после обработки прерывания произойдет возвращение в точку прерывания. Для обработки такого прерывания потребуются все ресурсы, может быть быть подан запрос на обмен с жестким диском или магнитной лентой. Происходит полное сохранение регистров, информация из регистрового буфера записывается в программную таблицу операционной системой. Ресурсы, что не были сохранены аппаратно(регистры общего назначения), тоже записываются в эту таблицу. Таким образом формируется точка контекста прерывания, которая сохраняется программно в таблице операционной системы. После этого снимается блокировка прерывания и запускаются операции завершения обработки прерывания.

Аппарат прерывания решает две задачи:

1. обнаружение и обработка возникающих в системе ошибок;
2. организация взаимодействия с внешними устройствами.

Внешние запоминающие устройства (ВЗУ)

Третьим компонентом схемы фон Неймана является внешнее устройство. Если в начале самым дорогим в компьютере был процессор и оперативная память, то сейчас стоимость компьютера определяют внешние устройства (монитор, клавиатура и т.д.).

Внешние запоминающие устройства используются в системе для оперативного хранения информации, с возможностью доступа и обработки этой информации. Принципиальным вопросом является скорость работы ВЗУ в рамках системы.

Систематизируем свойства внешних запоминающих устройств по разным признакам.

1) Обмен данными может осуществляться

- записями фиксированного размера – *блоками.* Пример блок-ориентированного устройства – это диски.
- записями произвольного размера. Такие записи предполагают наличие на устройстве маркеров, разделяющих записи.

2) Доступ к данным: • операции чтения и записи (жесткий диск, CDRW). • только операции чтения (CDROM, DVDROM, ...).

По характеристикам доступа к данным различают устройства:

- *Последовательного доступа.* В устройствах последовательного доступа для чтения i-ого блока памяти необходимо пройти по предыдущим i-1 блокам. (Аналогия – книга без оглавления. Чтобы найти необходимую информацию, надо прочитать всю книгу до нужного момента.) Пример такого устройства – магнитная лента. Устройства последовательного доступа в вычислительных системах используются для организации архивирования информации или долговременного хранения информации. Для этих целей не столько важно время доступа, сколько объем и надежность хранения информации.
- *Прямого доступа.* В таких устройствах не нужен просмотр предыдущих записей при организации обмена с какой-то из записей, размещенных на этом устройстве. (Аналогия – книга с оглавлением) То есть в устройствах прямого доступа для того, чтобы прочесть i-ый блок данных, не нужно читать первые i-1 блоков данных. устройства прямого доступа в основном являются блок-ориентированными устройствами. Пример – диск.

  В каждом из блок-ориентированного устройства прямого доступа можно выделить этап механической обработки заказа на обмен.

  - Для жесткого диска это введение блока головок на нужную позицию под дисками, их фиксация, проворот диска до точки начала информации (см. рис 5.3).
  - Магнитный барабан организован так, что в нем есть некоторая фиксированная штанга, на которой размещено фиксированное количество головок обмена (см. рис 5.4). Для совершения обмена необходимо, чтобы барабан прокрутился до позиции, где находятся данные. Когда эта позиция окажется под головками, электронным образом включается головка.

  Чем меньше совершается механических действий, тем быстрее устройство.

ФАКУЛЬТЕТ ВЫЧИСЛИТЕЛЬНОЙ МАТЕМАТИКИ И КИБЕРНЕТИКИ МГУ ИМЕНИ М.В. ЛОМОНОСОВА

Рис. 5.3. Схема магнитного диска

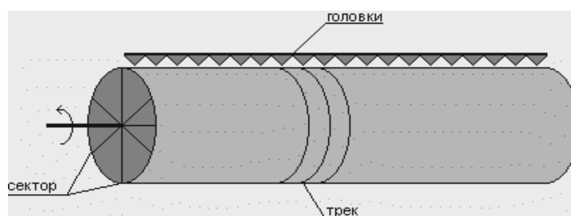

Рис. 5.4. Схема магнитного барабана

Синхронная организация обмена

При управлении внешними запоминающими устройствами, важной характеристикой является *модель синхронизации.*

Есть две модели синхронизации обмена:

1. синхронная организация обмена. Допустим, в определенный момент времени программа обращается к магнитной ленте с заказом на чтение информации из какой-то записи, т.е. после подачи заказа на обмен драйвер устанавливает соответствующие параметры обмена и запускает устройство, лента начинает проматываться. Драйвер ожидает информацию о завершении обмена. Эта информация может быть передана драйверу за счет размещения кода завершения обмена в каком-то специальном регистре. В зависимости от результата обмена процесс, который обратился к магнитной ленте, может быть разблокирован. При синхронной работе все компоненты системы будут приостановлены.

2. асинхронная организация обмена. Допустим, в определенный момент времени программа обращается к магнитной ленте с запросом на обмен с какой-то записью. Идет обращение к драйверу устройства, драйвер передает информацию на аппаратуру управления магнитной лентой о том, что ему следует перемотать ленту на самое начало. После этого управление возвращается в драйвер, а затем управление возвращается в программу. Программа может продолжить выпол-

няться. Факт выполнения заказа будет показан для системы возникновением прерывания. То есть при завершении перемотки ленты на начало аппаратура управления магнитной лентой передаст в центральный процессор прерывание. После этого драйвер передает подает заказ на перемотку этой ленты к началу записи, с которой нужно совершить обмен, затем снова возвращается в программу. Система продолжает выполнение. Факт перемотки снова будет обозначен прерыванием. Затем драйвер передаст третью команду обмена. После завершения этой операции снова будет прерывания.

⇒ Асинхронная модель предполагает возможность параллельной обработки взаимодействия с внешними устройствами с информированием вычислительной системы о результатах запросов по управлению внешним устройством посредством прерываний. То есть для организации асинхронного обмена является наличие аппарата прерываний.

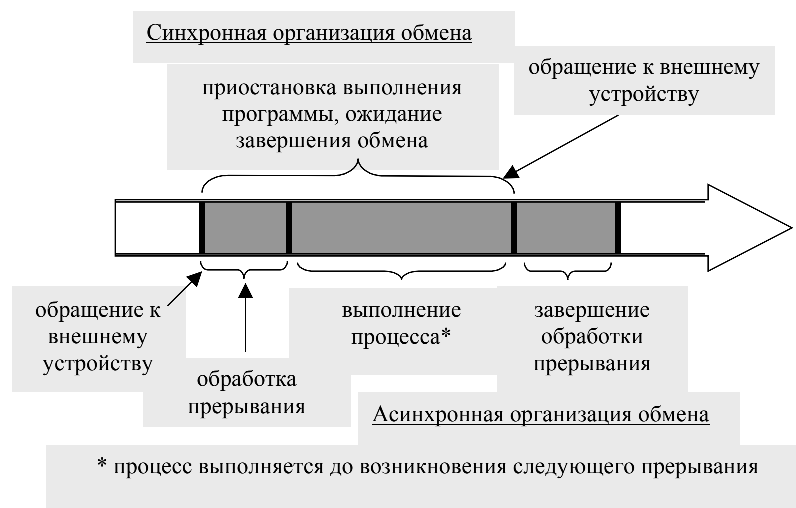

Рис. 5.5. Схема организации обмена

Иерархия устройств хранения информации

Можно разместить устройства хранения информации в некоторой иерархии (см. рис. 5.6).

- На вершине размещены самые быстрые и дорогостоящие устройства. К ним можно отнести регистровую память, регистры общего назначения, регистры специального назначения, кэш первого уровня.

- Далее идет кэш второго уровня. Это дорогостоящая сверхпроизводительная память, которая размещена все процессора. Кэш второго уровня взаимодействует с процессором посредством проводников, которые забирают время.
- Оперативное запоминающее устройство.
- Внешнее запоминающее устройство. Из ВЗУ прямого доступа самыми производительными считаются те, что имеют кэш. Такие устройства, у которых аппаратура управления, контроллеры, имеют внутри себя буфер, который буферизует обращения к соответствующему диску. Это приводит к сокращению реальных обменов из оперативной памяти к более медленному дисковому устройству, а также продлению жизни устройства.
- Внешнее запоминающее устройство долговременного хранения данных.

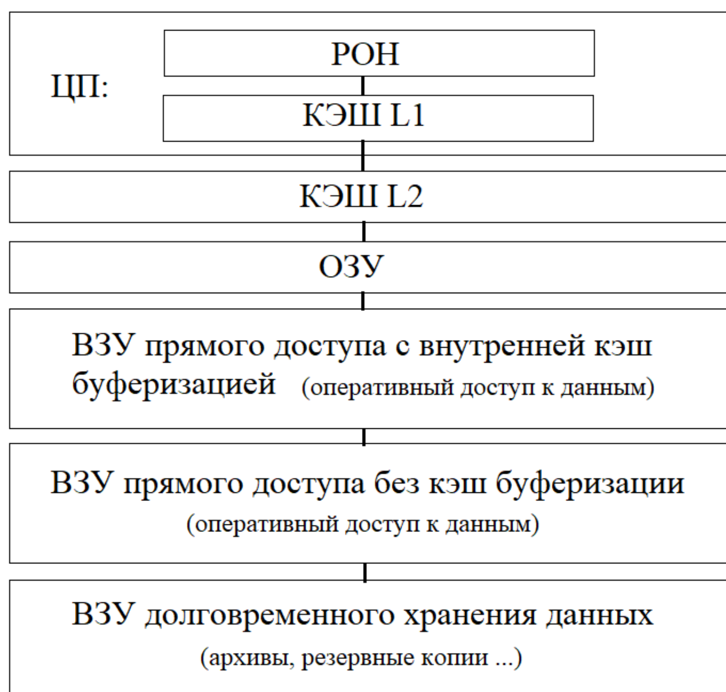

Рис. 5.6. Схема организации обмена

Аппаратная поддержка ОС и систем программирования

Существуют аппаратные средства компьютера, которые ориентированы на оптимизацию функционирования системного программного обеспечения, операционных систем и систем программирования.

На сегодняшний день любая вычислительная система предполагает функционирование в рамках своих ресурсов минимум двух программ. Это программа, которая выполняет функции операционной системы и пользовательской программа. Эти две программы функционировали еще тогда, когда система называлась однопрограммной. Когда в рамках вычислительной системы происходила обработка только одной пользовательской программы.

Когда вычислительная система работает в однопрограммном режиме, корректность функционирования системы является проблемой пользователя.

Мультипрограммный режим предполагает функционирование вычислительной системы и обработку единовременно двух и более программ пользователей. Для мультипрограммных вычислительных систем появляются особые требования к корректности функционирования этой вычислительной системы. Каждая программа, которая завершилась, имеет свой детерминированный результат. Необходимо, чтобы в получение этого результата не было никакого несанкционированного вмешательства. Для однопроцессорной системы в мультипрограммном режиме одна программа выполняется, какие то из других процессов будут заблокированы, потому что они подали запрос на обмен и ждут завершения обмена, оставшиеся программы уже завершили обмен и ожидают, когда им будет предоставлен процессор.

С этим связан ряд проблем.

- В общем случае исполняемая программа может обратиться в память любой другой программы, прочесть информацию и изменить ее. Если такая программа есть, то говорить о мультипрограммировании нельзя.
- Если при работе системы в мультипрограммном режиме какая-то из программ, находясь в состоянии исполнения, зациклилась, то система зависнет, перестанет работать продуктивно.

Лекция 6. Аппаратная поддержка ОС

Базовая аппаратная поддержка мультипрограммного режима

Определение. *Мультипрограммный режим* – режим, при котором возможна организация переключения выполнения с одной программы на другую.

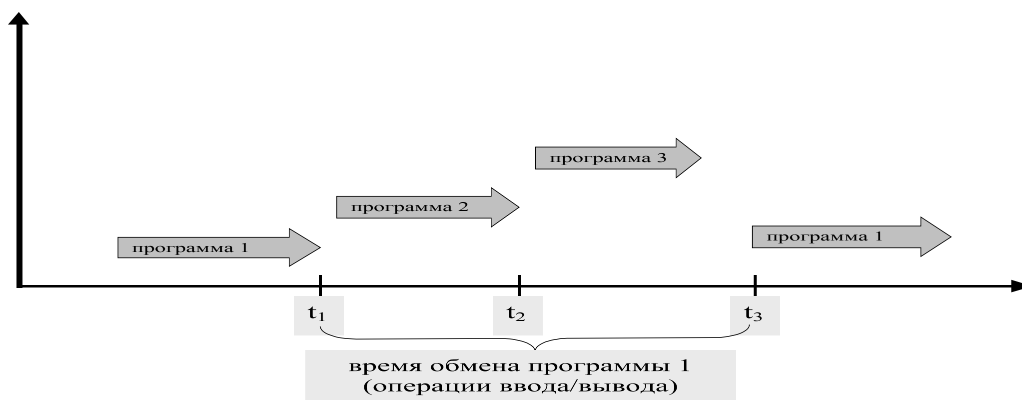

Рис. 6.1. Мультипрограммный ражим

Для мультипрограммной системы появилось особое требование к корректности функционирования. Система должна обеспечить, чтобы каждая из выполняемых программ в мультипрограммном режиме не оказывала нелегального действия на другие обрабатываетс=мые программы или процессы.

Чтобы можно было работать в мультипрограммном режиме на каком-то компьютере, этот компьютер должен обеспечить *аппаратную поддержку корректного мультипрограммирования.*

Аппаратная поддержка корректного мультипрограммирования направлена на борьбу со следующими проблемами:

1. Если в мультипрограммном/мультипроцессном режиме одна программа/процесс может обратиться к оперативной памяти, веделенной для другой программы/процесса, и считать или записать туда информацию, эта ситуация будет некорректрной.

   Вычислительная система должна обеспечить защиту памяти. Эта защита должна быть реализована аппаратно. То есть в случае попытки обращения по адресу, который не принадлежит исполняемой программе, должно произойти прерывание.

   ⇒ Решение – *Аппарат защиты памяти.*

   Операционная система, прежде чем запустить программу, передать этой программе процессор, выполнит специальные команды, которые перенастроют кон-

ФАКУЛЬТЕТ ВЫЧИСЛИТЕЛЬНОЙ МАТЕМАТИКИ И КИБЕРНЕТИКИ МГУ ИМЕНИ М.В. ЛОМОНОСОВА

троль аппарата защиты памяти.

2. Доступ к специальным командам должен быть ограничен. В системе должна быть возможность запрета исполнения части машинных команд для процессов/программ пользователей. Получается, в системе должны быть реализованы два режима:

   (a) режим работы операционной системы, в котором процессору доступны все возможные команды;

   (b) режим пользователя, в котором часть команд запрещена.

   Пусть в мультипрограммном режиме обрабатывается несколько программ. Все программы в определенные моменты времени обращаются к устройству печати посредством обращения к определенному драйверу, который будет осуществлять печать. Операционная система будет принимать запросы на обмен, аккумулировать эту информацию и выдавать на принтер информацию, когда предыдущий процесс будет закончен.

   Обращение из программы пользователя к принтеру может осуществляться посредством запрещенной команды. В результате произойдет прерывание, управление перейдет к операционной системе, которая увидит, что процесс пытается обратиться к печатающему устройству.

   ⇒ Решение – *Специальный режим: режим операционной системы и пользовательский режим.*

3. Проблема зацикливания процесса может быть решена с помощью прерывания по таймеру. Периодически управление от исполняемого процесса передается операционной системе, которая может передать управление другому процессу, если решит, что прошлый процесс зациклился.

   ⇒ Решение – *Аппарат прерываний.*

В итоге, аппаратные средства компьютера, необходимые для поддержания мультипрограммного режима:

1. Аппарат защиты памяти.

2. Специальный режим операционной системы. (привилегированный режим или режим супервизора)

3. Аппарат прерываний (как минимум, прерывание по таймеру).

Перемещаемость программы по ОЗУ. Фрагментация памяти

Рассмотрим другие проблемы, связанные с функционированием системы в мультипрограммном режиме.

Если программу, написанную на ассемблере, компилировать с помощью компилятора ассемблера, получится объектный модуль. Объектный модель – программа, еще не распределенная по адресам, внешние связи еще не решены. Поле получения объектного модуля, его можно передать на редактор внешней связи и/или загрузчик, который свяжет программу по адресам оперативной памяти, в которые затем будет загружена эта программа. Получаем исполняемый модуль. Момент подготовки исполняемого модуля и момент его исполнения могут быть разнесены во времени.

Проблемы мультипрограммирования в контексте использования оперативной памяти:

1. *проблема перемещаемости программы, проблема перемещаемости кода.* В мультипрограммном режиме может сложиться такая ситуация, когда адреса, на которые должна быть загружена программа, заняты другой программой. Чтобы система эффективно использовалась, можно было бы взять исполняемый модуль и поместить его не в тот диапазон адресов, на который он настроен, а переместить по оперативной памяти в любой диапазон оперативной памяти, достаточный по объему, но не связанный с той адресацией, которая используется в программе.

2. *Проблема фрагментации программы.* Пусть в мультипрограммной системе решена проблема перемещения программы по памяти. После запуска системы, операционная система загрузила последовательно программы для мульпрограммной обработки и начала их обрабатывать. Попеременно какая-то программа становится исполняемой, какие-то ждут завершения обмена, а какие-то ждут, когда операционная система им предоставит процессор.В некоторые произвольные моменты времени программмы будут завершаться и будут образовываться свободные участки памяти.

   Планировщик операционной системы будет забираться в свой буфер программ, которые ждут обработки, и по какой-то стратегии будет выбирать программы для обработки, которые поместятся в освободившийся фрагмент. После помещения программы может остатья небольшой фрагмент свободной памяти, Таких фрагментов свободной памяти, которыми не может воспользоваться никакая другая программа, может быть много. Таким образом получается, ресурс используется неэффективно. Эта проблема и называется проблемой фрагмента-

ции программы.

Виртуальная память

Решить проблемы перемещаемости и фрагментации можно с помощью *виртуальной памяти.*

Для программы/исполняемого кода всегда существует некоторая модель оганизации и использования адресного пространства, которое используется в данной программе.

Определение. Логическое(виртуальное) адресное пространство – адресное пространство, которое используется в программе.

Каждая готовая к исполнению программа имеет свое адресное пространство, свое витруальное пространство. То есть в исполняемом модуле имеется вируальное адресное пространство. С другой стороны, мы имеем физическую память.

Решение проблем перемещаемости и фрагментации программы связано с возможностью отображения виртуального пространства, используемого внутри процессора или внутри программы, на физическую память.

Аппарат виртуальной памяти – модель, связанная с понятием базирования адресов.

Определение. *Аппарат виртуальной памяти* – аппаратные средства компьютера, обеспечивающие преобразование (установление соответствия) программных адресов, используемых в программе адресам физической памяти в которой размещена программа при выполнении.

Идея модели заключается в том, что в процессоре выделяется специальный регистр базы. Исполнительный программный адрес может быть двух типов:

1. *абсолютный программный адрес*, который ссылается на фиксированную точку физической памяти;

2. *относительный адрес*, к которому после получения исполнительного программного адреса добавляется содержимое регистра базы, после чего получаем исполнительный физический адрес.

Пусть есть физическое адресное пространство, в котором размещена операционная система, имеем исполнительный модуль, который основывается на базировании адресов. Все адреса этого исполнительного модуля построены не как адреса по физической памяти, а как смещения от начала этого исполнительного модуля. То есть относительный исполнительный программный адрес является расстоянием от начала

модуля до конкретной точки. Вся программа строится на использовании таких относительных исполнительных программных адресов. Эту программу можем поместить в любую область физической памяти. Затем адрес начала загрузки записывается в регистр базы. После этого автоматически все относительные исполнительные адреса будут увеличиваться на значение регистра базы.

⇒ Таким образом из относительных адресов можео получить адреса конкретных физических точек.

То есть аппарат виртуальной памяти – такая аппаратно-программная функция, которая обеспечивает преобрахование виртуального адреса, который используется внутри процесса/программы, в физический адрес того места в физической оперативной памяти, в которое загружена исполняемая программа

Адресация с базированием. Страничная организация памяти

Рассмотрим модели виртуальной памяти.

1. *Адресация с базированием.* Базирование адресов – реализация одной из моделей аппарата виртуальной памяти. Базирование решает проблему перемещаемости программы. При базировании происходит отображение фрагмент виртуального адресного пространства в физическую память один в один. То есть должен быть найден фрагмент в физической памяти такого же размера. Базирование памяти решает проблему перемещения, но не решает проблему фрагментации. Для решения проблемы фрагментации используются более развитые механизмы организации ОЗУ и виртуальной памяти.

2. *Страничная организация памяти.* Все адресное пространство представляется как совокупность страниц. Страница – некоторая область памяти фиксированного размера 2^k. Физическое адресное пространство разделяется на физические страницы, соответственно виртуальное адресное пространство, которое может использоваться в программе/процессе, разделяется на страницы такого же размера.

   Определение. *Виртуальное адресное пространство* – множество виртуальных страниц, доступных для использования в программе. Количество виртуальных страниц определяется размером поля «номер виртуальной страницы» в адресе.

   Определение. *Физическое адресное пространство* – оперативная память, подключенная к данному компьютеру. Физическая память может иметь произвольный размер (число физических страниц может быть меньше, больше или равно числу виртуальных страниц).

ФАКУЛЬТЕТ ВЫЧИСЛИТЕЛЬНОЙ МАТЕМАТИКИ И КИБЕРНЕТИКИ МГУ ИМЕНИ М.В. ЛОМОНОСОВА

Так как размер страницы – степень двойки, верно следуюее устверждение: если из любого адреса вырезать k крайних левых(младших) разрядов, получим, что старшие разряды с k-го провее являются номером страницы, а младшие k разрядов – это номер в странице(см. рис. 6.2). Номер в странице фактически является смещением относительно начала страницы.

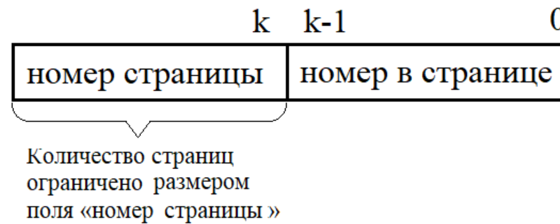

Рис. 6.2. Структура адреса

Модельная реализация.

Пусть в процессоре есть аппаратная таблица страниц фиксированного размера. Количество строк в этой таблице равно предельному количеству страниц, которые могут быть в виртуальном адресном пространстве. i-я строка этой таблицы соответствует i-й виртуальной страницы обрабатываемого процесса. Содержание i-й строки таблицы – номер физической страницы оперативной памяти, в которой размещена i-я страница обрабатываемого процесса.

Преобразование виртуального адреса в физический:

- gолучаем исполнительный виртуальный адрес при выполнении программы;
- вырезаем из исполнительного виртуального адреса номер виртуальной страницы;
- индексируемся по таблице страниц с использованием номера виртуальной страницы, оттуда взяли код α_i .

  В устройства управления аппаратно оценивается код α_i , который говорит о том, есть ли виртуальная страница.

  - Если α_i >0, то α_i – номер физической страницы Говорим, что i-й виртуальной странице процесса соответствует физическая страница с номером α_i. Затем формируем физический адрес.
  - Если $\alpha_i = 0$, то либо соответствующей виртуальной страницы нет в памяти, либо она запрещена. Соответственно, срабатывает прерывание.

ФАКУЛЬТЕТ ВЫЧИСЛИТЕЛЬНОЙ МАТЕМАТИКИ И КИБЕРНЕТИКИ МГУ ИМЕНИ М.В. ЛОМОНОСОВА

Управление переходит к поерационной системе. Операционная система обращается к информации об обрабатываемом процессе. Из этой информации она понимает, что либо i-й виртуальной страницы нет ⇒ ошибка, либо i-я виртуальная страница у проуесса есть, но она еще не загружена в физическую память. В этом случае операционная система формирует заказ на дозакачивание этой страницы в память и продолжает выполнение как-то другой программы, а эту программу закрывает по обмену.

То есть берем строку с номером, равным номеру виртуальной страницы. Содержимое этой строки – номер физической страницы, в которой размещена соответствующая виртуальная страница процесса.

- В исполнительном адресе меняем поле "номер виртуальной страницы"на содержимое строки таблицы страниц, на номер физической страницы.

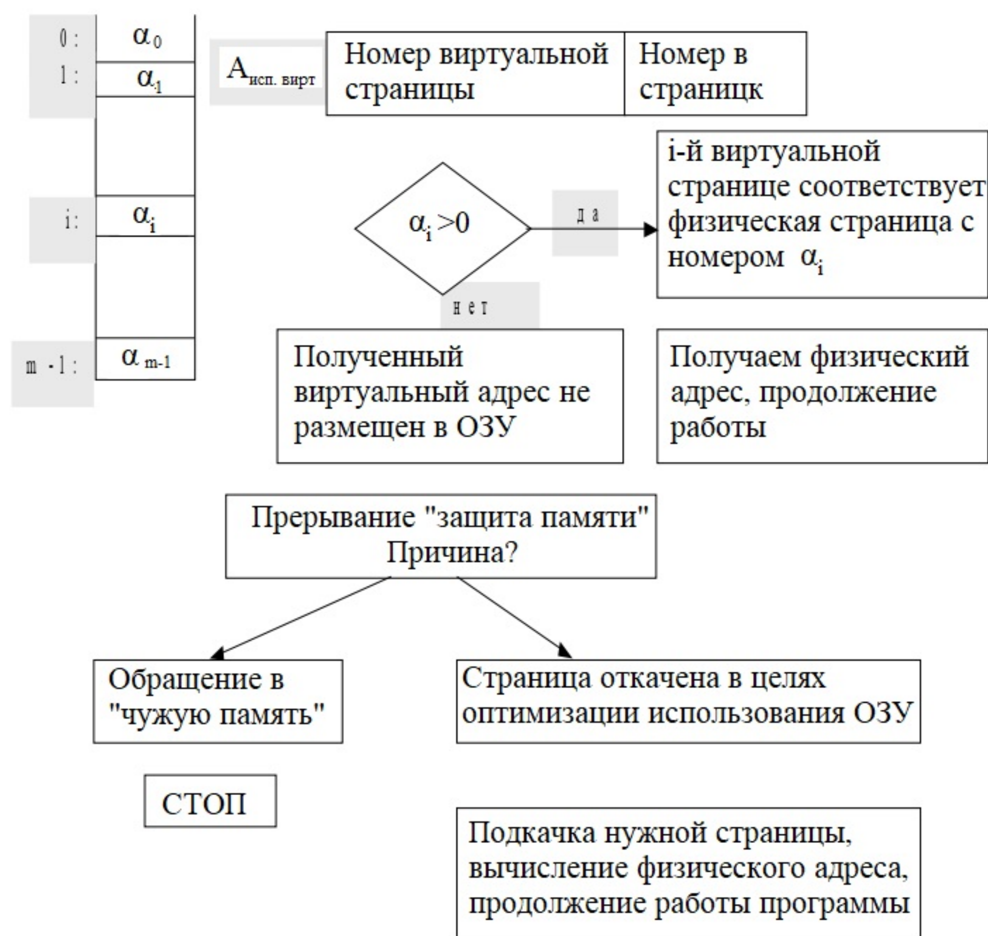

Рис. 6.3. Модельный пример организации страничной виртуальной памяти

ФАКУЛЬТЕТ ВЫЧИСЛИТЕЛЬНОЙ МАТЕМАТИКИ И КИБЕРНЕТИКИ МГУ ИМЕНИ М.В. ЛОМОНОСОВА

Плюсы страничной организации памяти

- решается проблема фрагментации в терминах страниц;
- для запуска программы не нужно иметь свободной физическую память, равную размеру этого процесса. Достаточно иметь свободными несколько страниц.

За счет того, что в оперативной памяти находится только небольшая часть программы, проблемы внутренней фрагментации сходят на нет.

Регистровые окна

Одна из проблем модульных программ, то есть программ, которые состоят из большого количества модулей или подпрограмм – это накладные расходы, связанные со входом/выходом в модуль. При входу в модуль нужно сохранить контекст, сформировать параметры, которые передаются в обращениик подпрограмме, а при выходе необходимо это восстановить. В модуле можно использовать регистры общего назначения. Перед входом используемые в модуле регистры общего назначение нужно сохранить, а затем восстановить. На вход/выход уходят ресурсы работы программы.

Для решения этой пролемы в современных компьютерах появилась возможность использования регистровых окон.

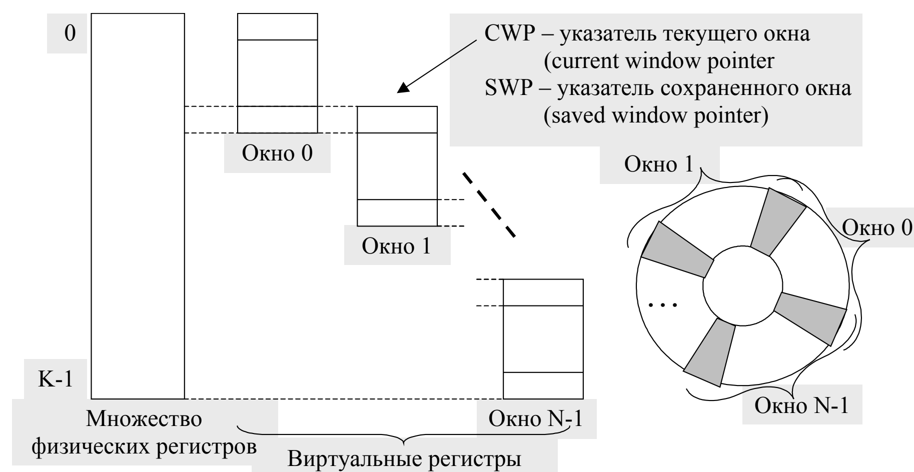

Рис. 6.4. Регистровые окна

ФАКУЛЬТЕТ ВЫЧИСЛИТЕЛЬНОЙ МАТЕМАТИКИ И КИБЕРНЕТИКИ МГУ ИМЕНИ М.В. ЛОМОНОСОВА

Физически в процессоре имеетя k регистров общего назначения, а через систему коменд в программе доступна только часть этих регистров. Аппарат регистровых окон позволяет реализовывать команды, которые при входе в подпрограмму перемещают окошко, в котором находятся доступные регистры общего назначения из отображения в обни реальные регистры на отображение в другие. Например, в основной программе может использоваться первый регистр общего назначения, а после обращения в подпрограмму окошко сменится, и станут доступны виртуальные регистры.

⇒ Вводится виртуализация регистров.

Есть регистровый файл, который содержит физические регистры, есть регистры, доступные для программы – виртуальные регистры. Это окошечко виртуальных регистров может отображаться на разные области физических регистров приобращении к подпрограмме и выходе из нее.

Классификация архитектур многопроцессорных ассоциаций

Рассмотрим классификацию архитекруры, предложенную Майклом Флинном.

В любом компьютере можно выделить два потока информации:

- поток команд – это то, в какой последовательности, с какой интенсивностью обрабатываются машинные программы в компьютере;
- поток операндов или данных.

Было предложено рассматривать каждый из этих поток двумя способами:

- одиночный, когда идут последовательные запросы к командам, последовательные запросы к данным;
- множественный, когда единовременно идут групповые запросы и обработка к командам, групповые запросы и обработка к данным

Исходя из этой концепции, получаем, что есть четыре типа компьютеров:

1. ОКОД (SISD – Single Instruction, Single Data stream)– одиночный поток команд, одиночный поток данных. Этой ситуации соответствует однопроцессорный компьютер(тот, что ближе всего к компьютеру фон Неймана)
2. ОКМД (SIMD – Single instruction, Multiple Data stream)– одиночный поток команд, множественный поток данных. Одна команда инициирует обработку группы данных. Это могут быть векторные или матричные компьютеры.

3. МКОД (MISD – Multiple Instruction stream, Single Data stream)– множественный поток команд, одиночный поток данных. Могут быть некоторые специализированные компьютеры, которые могут обрабатывать видео/информацию, осуществлять медийный поиск.

4. МКМД (MIMD – Multiple Instruction stream, Multiple Data stream) – множественный поток команд, множественный поток данных. К этой группе относятся многопроцессорные компьютеры, которые одновременно обрабатывают свои последовательности данных.

Иерархия MIMD-систем

Группу МКМД можно разделить на две части:

- многопроцессорные системы с общей памятью – такие системы, в которых каждый процессорный элемент имеет доступ к любой точке оперативной памяти.

  Системы с общей оперативной памятью делятся на два класса:

  – системы с унифицированным доступом к памяти – многопроцессорная система в которой каждый процессорный элеменет имеет равнозначные возможности по доступу в любую точку оперативной памяти.
  – системы, в которых отсутствует унификация по доступому к памяти – многопроцессорная система с общей памятью, в которой каждый процессорный элеменет имеет возможности по доступу в любую точку оперативной памяти, но в некоторые области он имеет доступ с одной скоростью доступа, а в некоторые с другой.

- системы с распроделенной памятью – многопроцессорные системы, которые состоят из узлов. Каждый узел включает в себя процессор и свою локальную оперативную память, которая не доступна для других проуессоров. взаимодействие в рамках этой системы осуществляется на уровне межпроцессорного взаимодействия.

Лекция 7. Классификация архитектур многопроцессорных ассоциаций

Классификация архитектур многопроцессорных ассоциаций

Продолжим рассматривать классификацию компьютеров по Майклу Флинну.

Согласно этой классификации, выделяется два потока информации, которая перемещается и обрабатывается в компьютере: поток команд и поток данных/операндов.

Каждый из этих двух потоков можно рассматривать по тому, является ли он одиночным потоком, когда перемещение и/млм обработка команд или данных происходит последовательно, или множественным потоком, когда единовременно осуществляется перемещение и/или обработка группы команд или операндов.

Сответственно, все компьютеры можно разделить на 4 группы по этим признакам.

1. ОКОД (SISD – Single Instruction, Single Data stream)– одиночный поток команд, одиночный поток данных. Этому соответствует традиционная однопроцессорная архитектура, которая модельно близка к компьютеру фон Неймана.

2. ОКМД (SIMD – Single instruction, Multiple Data stream)– одиночный поток команд, множественный поток данных. Последовательное исполнение одиночных команд инициирует множественный поток и обработку данных. Такой тип ассоциируют с векторной или матричной обработкой данных. Примером такого компьютера является векторный компьютер.

3. МКОД (MISD – Multiple Instruction stream, Single Data stream)– множественный поток команд, одиночный поток данных. Условно вырожденная группа. К этой группе можно отнести некоторые специализированные компьютеры, которые занимаются задачами распознавания.

4. МКМД (MIMD – Multiple Instruction stream, Multiple Data stream) – множественный поток команд, множественный поток данных. Компьютеры, относящиеся к этой группе, имеют множество процессорных элементов, которые единовременно обрабатывают свои или общие данные.

Иерархия MIMD-систем

Представим категорию MIMD в некоторой структурной организации.

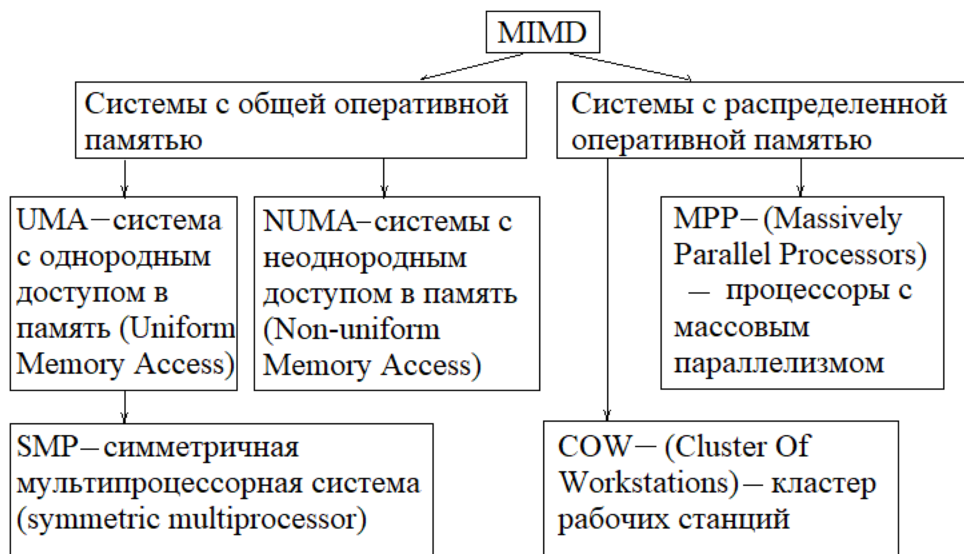

Рис. 7.1. Иерархия MIMD- систем

- *Системы с общей оперативной памятью* – многопроцессорные системы, в которых каждый процессорный элемент может иметь доступ к любой точке оперативной памяти.

  Системы с общей оперативной памятью можно разделить на две категории:

  – UMA система – многопроцессорная система в с общей памятью, в которой характеристики доступа каждого процессорного элемента в любую точку оперативной памяти одинаковы.

  – NUMA система – многопроцессорная система в которых есть доступ к общей памяти, но каждый процессорный элемент может иметь разные характеристики доступа к разным точкам оперативной памяти. То есть для каждого элемента есть своя локальная память и удаленная память.

- *Системы с распределенной памятью* – такие многопроцессорные системы, в которых оперативная память разделена на области, каждая из областей принадлежит одному процессорному элементу. Т.е. каждый процессорный элемент работает со своей локальной оператвной памятью.

  Системы с распределенной памятью тоже можно разделить на две категории:

  – MPP системы – массивно-параллельные процессорные системы.

  – COW системы – кластер рабочих станций

ФАКУЛЬТЕТ ВЫЧИСЛИТЕЛЬНОЙ МАТЕМАТИКИ И КИБЕРНЕТИКИ МГУ ИМЕНИ М.В. ЛОМОНОСОВА

SMP-системы

Рассмотрим системы с общей оперативной памятью. SMP-системы являются подвидом UMA-систем.

SMP – Symmetrical MultiProcessing. Компьютеры с симметричными мультипроцессорами имеют общую шину, т.е. есть некоторый физический канал к которому подключены симметричные процессорные элементы. К этой же шине подключена общая память.

⇒ Любой запрос от каждого процессорного элемента в оперативную память проходит по единому каналу. По общей шине в каждый момент времени может двигаться только один запрос.

Минусы:

- В такой архитектуре возможны конфликты, когда для в процессорных элемента единовременно хотят обратиться к общей шине.
- Такая архитектура существенно ограничивает параллелизм системы. Количество процессорных элементов, которые можно включить в SMP-систему, ограничено из-за возможности возникновения конфликтных ситуаций. Можно увеличить допустимое количество процессорных элементов, увеличив скорость шины.

Синхронизация кэша

Для многопроцессорных систем с общей оперативной памятью используются более сложные модели организации кэша, которые называются *кэш с отслеживанием*.

Идея кэш-памяти с отслеживанием заключается в том, что каждый процессорный элемент в каждый момент времени в независимости от того, по общей шине идут запросы, инициированные им или его соседями, может слушать общую шину. Таким образом процессорный элемент отслеживает, какие операции происходят на связке общая шина-оперативная память.

Рассмотрим функционирование кэш-памяти с отслеживанием.

1. Пусть в каком-то процессоре SMP-системы происходит промах при чтении. В локальном кэше не нашлось нужной информации, поэтому происходит дополнение локального кэша недостающей информацией. Другие процессорные элементы видят, что происходит чтение информации и ничего не делают со своим локальными кэшами.

Операции	Локальный кэш	Кэш других процессоров
Промах при чтении	Запись из памяти в кэш	Ничего
Попадение при чтении	Использование кэша	Ничего (операция "не видна")
Промах при записи	Запись в память	Соответствующая запись в кэше удаляется
Попадение при записи	Запись в память и кэш	Соответствующая запись в кэше удаляется

Рис. 7.2. Поведение кэш-памяти с отслеживанием при чтении/записи

2. Если происходит попадание, из локального кэша берется информация процессором, шина остается свободной.

3. Пусть в случае, когда надо записать информацию в оперативную память, происходит промах при записи, нужного адреса нет в локальном кэше. Происходит обновление информации в оперативной памяти, процессорный элемент инициирует обмен по общей шине. Все соседние процессоры узнали, что какой-то элемент пишет в оперативную память. В этом случае процессоры ловят соответствующий адресный операнд, который прошел по шине, и удаляют запись с этим адресным элементом в своих кэшах.

4. Если происходит попадание при записи, осуществляется запись в память и обновление кэша. У всех остальных снова соответствующая запись в кэше удаляется.

Данная модель ориентирована на то, что статистически чаще встречается операция чтения. Организация кэша и алгоритм его работы ориентированы на чтение.

Преимущества SMP- системы :

- простая и недорогая реализация;
- возможность распараллеливания → ускорение работы.

Недостатки SMP-сисемы:

- общая шина, риск появления конфликтов → деградация системы;
- задержки при доступе к памяти из-за общей шины;

ФАКУЛЬТЕТ ВЫЧИСЛИТЕЛЬНОЙ МАТЕМАТИКИ И КИБЕРНЕТИКИ МГУ ИМЕНИ М.В. ЛОМОНОСОВА

- ограничение на количество процессорных элементов (как следствие централизации)

NUMA-системы

Такая система тоже может строиться на основе архитектуры с общей шиной. NUMA-системы предполагают чуть более сложную и дорогую архитектуру, которая предполагает наличие в каждом процессорном элементе локальной памяти(самая высокоскоростная оперативная память), а также появляется контроллер локальной памяти для каждого из процессоров. Контроллер слушает запросы, которые появляются на шине. Если эти запросы попадают в адресное пространство локальной памяти процессорного элемента, контроллер эти запросы переадресовывает в локальную память.

То есть в такой системе запросы идут не на общую память, а на память конкретного процессорного элемента.

Для NUMA-систем используются более сложные и дорогостоящие модели кэширования.

Преимущества NUMA-системы :

- за счет областей локальной памяти, при выполнении процессором программы, сокращается количество обращений через общую шину ⇒ можно повысить степень параллелизма, процессорных элементов может быть больше чем в SMP.

Недостатки NUMA-сисемы:

- сложность системы, накладные расходы, связанные с тем, что некоторая информация перемещается по общей шине;
- сложность кэширования, загрузка шины служебной информацией.

Системы с распределенной памятью (MPP-системы)

Виды систем с распределенной памятью:

- MPP – массивно-параллельные системы;
- COW – кластеры рабочих станций.

MPP – специализированный многопроцессорный компьютер с распределенной памятью, каждый процессорный элемент имеет свою локальную оперативную память, недоступную для других процессорных элементов. В таких компьютерах вычислительные узлы, состоящие из процессорного элемента и оперативной памяти, соединены друг с другом посредством некоторой топологии, которая определяется специализацией данного компьютера. Такие компьютерные системы хороши для макроконвеерной обработки информации, если поток информации можно разделить на последовательные независимые этапы. На каждом вычислительном узле решается функция этапа, а результат передается по линиям связи, соединяющим узлы. Либо для решения задачи поиска, может подойти архитектура в виде n-мерного куба. В каждом элементе решается своя подзадача, может быть коммуникация со всеми соседями, которых предусматривает архитектура.

MPP-системы – специализированные дорогостоящие компьютерные системы, изначально предназначенные для эффективного решения конкретного класса прикладных задач. Например, задачи, связанные с обороной. Могут единовременно обрабатывать большое количество данных.

COW-кластеры (кластеры рабочих станций)

Кластеры рабочих станций появились, когда стали развиваться локальные сети, например, в офисах. В таких случаях не используются вычислительные мощности компьютера.

Идея заключается в том, чтобы дешевые офисные рабочие станции соединить так, чтобы их можно было единовременно использовать как многопроцессорную систему, чтобы на ней можно было решать задачи в распараллеленом режиме. Попытка сделать многопроцессорный компьютер, затратив минимальные средства.

Кластерная архитектура – это многопроцессорная система, в которой процессорные элементы связаны друг с другом посредством стандартных или специализированных протоколов связи.

На сегодняшний день кластерная архитектура является одной из самых востребованных и распространенных архитектур категории МКМД.

Кластеры используются для двух целей:

1. Кластер как основа для построения суперкомпьютера, кластерная система, объединяющие тысячи вычислительных узлов, где каждый вычислительный узел – одноплатный компьютер. Такие одноплатные компьютеры объединены посредством сетевой коммуникационной среды и традиционных сетевых протоко-

лов семейства TCP/IP. На таких системах используются программные пакеты коммуникации, позволяющие организовывать взаимодействие между вычислительными узлами посредством обмена сообщениями по модели Send / Receive. В подавляющем большинстве работают под управлением операционных систем семейства UNIX.

2. Кластеры надежности – это многопроцессорная система, котоная состоит из вычислительных узлов, в которых есть оперативная память и процессорный элемент. Вычислительные узлы связаны посредством физических каналов связи и сетевых протоколов семейства TCP/IP в единую кластерную систему. Назначение такой системы – обеспечение надежной работы комплекса в случае выхода из строя компонентов, то есть с возможным снижением суммарной производительности системы. В случае выхода из строя одного из узлов, кластерная система сама автоматически диагностирует ситуацию, перестраивает архитектуру системы, а прикладная система, которая работает на кластере, продолжает работать на усеченных ресурсах. В части кластеров надежности лидируют кластеры, построенные на базе операционных систем компании Microsoft. Нет фиксированной топологии, которая ориентирована на конкретный тип задач.

Терминальные комплексы

Определение. *Терминальный комплекс* – многомашинная ассоциация, предназначенная для организации массового доступа удаленных и локальных пользователей к ресурсам некоторой вычислительной системы.

Пусть есть некоторая компьютерная система, на ресурсах которой размещается информация электронной библиотеки. Доступ к этим ресурсам можно обеспечить с помощью терминальных устройств.

Определение. *Терминал* – это специализированная аппаратура, позволяющая осуществлять обмен информации между самим терминалом и компьютером, к которому это терминал подключен. Простейшее терминальное устройство – принтер.

На более ранних этапах были устройства с клавиатурой и монитором.

Определение. *Локальный терминал* – это устройство, которое помещается на достаточно небольшом расстоянии от компьютера, к которому подключен локальный терминал. Под небольшим расстоянием подразумевается расстояние, которое не требует использования дополнительных средств для усиления сигнала для передачи информации в компьютер и обратно.

Подключение локального терминала к компьютерной системе может осуществ-

ляться непосредственно через интерфейсы компьютерной системы. При таком подключении затруднительно обеспечивать массовость доступа.

Для обеспечения массовости доступа к аппаратному терминалу подключается мультиплексор, к которому можно подключить группу терминалов.

Определение. *Мультиплексор* – аппаратное устройство, которое позволяет осуществить подключение группы терминалов к компьютерной системе через один аппаратный интерфейс.

В результате подключения большого количества терминалов падает скорость, соответственно рассчитывается предельное количество устройств.

⇒Таким образом решается проблема массового доступа локальных пользователей к вычислительной системе.

Удаленный пользователь – пользователь, который работает за терминалом на таком удалении от целевой вычислительной системы, что для передачи и приема информации между терминалом и компьютером требуется использование некоторой специальной аппаратуры.

Модель: подключение терминала к целевому компьютеру напрямую через аппаратный интерфейс. В этом случае используются аналоговые средства передачи информации. Если все компоненты компьютерной системы взаимодействуют в дискретном формате, то аналоговый сигнал непрерывен. Соответственно появились новые устройства.

Модемы – устройства, предназначенные для преобразования двоичной информации в непрерывный сигнал. Один модем ставится между терминальным устройством и входом в сеть, через которую можно передавать непрерывные сигналы, а второй модем ставится на выходе из этой сети.

Определение. *Удаленный терминал* – терминал, который подключается к целевому компьютеру посредством модемов и физической линии связи для передачи непрерывного сигнала.

Для реализации массового подключения можно подключить мультиплексор между модемом и группой удаленных терминалов. Соответственно возможен удаленный мультиплексор. Линию связи от удаленного терминала можно подключить не напрямую к аппатарному интерфейсу, а к входу локального мультиплексора.

Для каждого удаленного терминала или удаленного узла, где находится мультиплексор и группа терминалов, сложно проложить линию связи к компьютеру. Поэтому для коммуникаций в качестве физической среды передачи данных стали использовать телефонную сеть. Это объединение каналов и линий связи, использующих

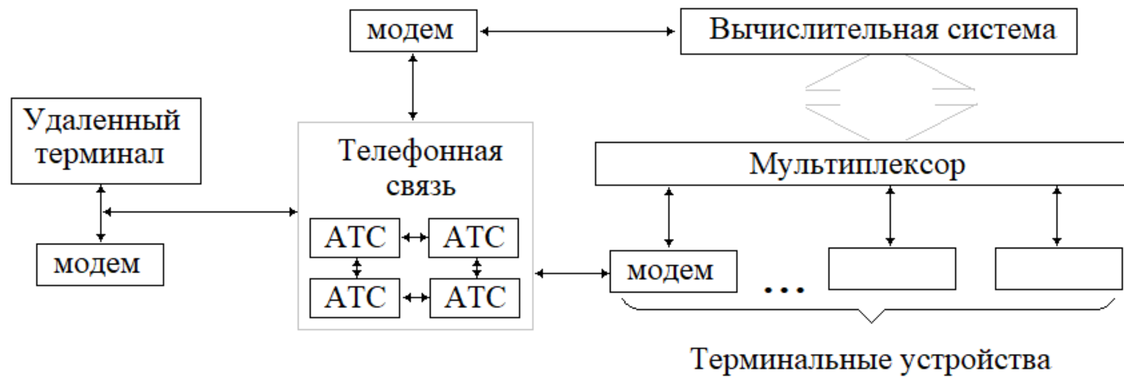

Рис. 7.3. Схема подключения удаленных терминалов

АТС, устройства коммуникации, и непосредственную физическую среду передачи данных, линии связи.

После этого появилась возможность формировать канал связи между удаленным терминалом и целевым компьютеров не выделенный, а коммутируемый.

Линии связи и каналы

Рассмотрим два типа каналов:

1. выделенный канал – такой канал, посредством которого фиксируется соединение терминального устройства с целевым компьютером, постоянно действующая физическая линия передачи и приема информации (дорогое и неэффективное решение);

2. коммутируемый канал – линия связи, которая выстраивается между двумя взаимодействующими устройства на промежуток времени, когда осуществляется взаимодействие (пример: телефонная линия). Возможен отказ на соединение, когда коммуникационная среда не позволяет построить соединение.

Организация канала:

- канал точка-точка, когда на один входной интерфейс одно устройство;

- многоточечный канал в случае, когда группа устройств подключена к одному интерфейсу.

Направление движения информации.

- Симплексный канал – такой канал, который позволяет осуществлять передачу информации только в одном направлении (пример: пейджер);

- Дуплексный канал – канал, который позволяет передавать информацию единовременно в двух направлениях (пример: телефоная связь).
- Полудуплексные каналы – это канал, который позволяет осуществлять передачу информации в обоих направлениях, но в каждый момент времени только в одном (пример: рация).

Компьютерные сети

Развитие вычислительной техники предоставило возможность в простейшей модели многомашинной ассоциации, которая была рассмотрена как многомашинный комплекс, заменить терминальные устройства на компьютеры. Появилась возможность выполнения функций терминала не на специальном оборудовании, а на компьютере. Изначально терминальный комплекс стал многомашинной ассоциацией. С этого момента начало появляться понятие "компьютерная сеть".

Рис. 7.4. Компьютерная сеть

Определение. *Компьютерная сеть* – объединение компьютеров(вычислительных систем), взаимодействующих через коммуникационную среду.

Определение. *Коммуникационная среда* – объединение каналов и/или линий связи и специализированного оборудования, предназначенного для организации передачи данных.

Компьютерная сеть обладает следующими характеристиками.

1. Позволяет объединять в себе значительное количество вычислительных узлов, обеспечивающих решение задачи определенных типов, где каждый узел может быть отдельным компьютером или группой компьютеров.
2. Возможность распределения обработки информации.
3. Расширяемость сети. Компьютерная сеть, в отличие от фиксированного многомашинного комплекса, может быть расширяема территориально и функционально.

ФАКУЛЬТЕТ
ВЫЧИСЛИТЕЛЬНОЙ МАТЕМАТИКИ И
КИБЕРНЕТИКИ МГУ ИМЕНИ М.В.
ЛОМОНОСОВА

4. Применение симметричных интерфейсов обмена информации внутри сети (возможность произвольного распределения функций внутри сети)

Традиционно считается, что компьютерная сеть состоит из

- абонентских(основных) компьютеров – хостов;
- коммуникационных компьютеров – такие специализированные компьютеры, которые обеспечивают выполнение и реализацию коммуникационных функций
  - шлюзы
  - маршрутизаторы и т.д.

В реальности, в рамках одного компьютера решаются сетевые вопросы и абонентской машины, и вопросы коммуникационной обработки

Лекция 8. Компьютерные сети

Компьютерные сети

Далее будут обсуждаться коммутационные сети. В коммутационных сетях используются коммутируемые каналы, которые строятся и выделяются взаимодействующим сетевым устройством на некоторый период взаимодействия. Этот период взаимодействие определяет тип сети.

Рассмотрим три модели организации сети:

- Сеть коммутации каналов
- Сеть коммутации сообщений
- Сеть коммутации пакетов

Сетевые устройства взаимодействуют друг с другом посредством передачи сообщений. Определение. *Сообщение* – логически целостная порция данных, имеющая произвольный размер.

Взаимодействие абонентов осуществляется сеансами связи.

Определение. *Сеанс свзязи* – период времени , в течение которого сетевые устройства обмениваются сообщениями. Обычно с сеансом связи ассоциируются функции начала/завершения связи.

Сеть коммутации каналов

Сеть коммутации канала обеспечивает выделение канала на весь промежуток времени, связанный с сеансом работы сетевых устройств. Канал строится и выделяется на весь сеанс работы. Эта модель является исторически первой и наиболее простой. Дополнительные устройства для обеспечения связи кроме коммутаторов не нужны.

Пример сети коммутации канала: телефонный звонок. Во время сеанса линия связи физически является выделенной и непрерывной. По окончании звонка сети разрывается, канал возвращается в ресурс коммуникационной среды.

Преимущества:

- После установления соединения сеть находится в состоянии готовности;
- Требования к коммуникационному оборудованию минимальны;
- Связь обладает детерминированнными характеристиками;
- Минимизитуются накладные расходы по передаче данных.

Методы математической статистики, имея средние характеристики сеанса связи для телефонной сети, количество абонентов, можно рассчитать количество входов и выходов телефонной сети. Телефонная сеть рассчитана на детерминированную длительность голосовых сеансов связей.

Если в качестве сетевых устройств мы рассматриваем компьютеры, модели изменится, так как сеансы связи для компьютеров недетерминированы. Если на телефонную сеть,рассчитанную на голосовые соединения, положим компьютерные взаимодействие, сети начнет деградировать, количество отказов существенно увеличится.

Недостатки:

- Требование избыточности сети. Необходимо для минимальной частоты отказов на установление канала и минимальное время ожидания.
- Недетерминированный промежуток ожидания выделения канала.
- Неэффективное использование выделенного канала.
- В случае сбоя или отказа повторная передача информации.

Сеть коммутации сообщений

Взаимодействие представляется в виде последовательности обменов сообщениями. Сообщение начинает перемещаться по сети сразу же, как только обнаружится ближайшее к отправителю коммуникационное устройство, которое может принять это сообщение. Канал не строится, передача сообщений от одного коммуникационного устройства к другому.

Преимущества:

- Передача сообщения начинается как только обнаруживается ближайшее свободное коммуникационное устройство на пути к получателю сообщения $\Rightarrow$ время ожидания меньше.
- Отсутствие занятости канала на недетерминированный промежуток времени.

Недостатки

- Существенные вложения в коммуникационную среду. Коммуникационные узлы должны представляться в виде компьютеров, которые могут принимать поступающие сообщения и сохранять их до момента отправки далее.

- Возможна ситуация, когда свободных ресурсов коммуникационного ресурса недостаточно для приема сообщения, тогда коммуникационное устройство будет занято.
- Характеристики коммуникационных устройств недетерминированы, так как сообщение является порцией данных произвольного размера.
- Повтор всего сообщения в случае сбоя передачи .
- Требуется более сложное программное обеспечение, которое обеспечит синхронизацию и контроль за передачей сообщений.

Сеть коммутации пакетов

В начале 1960-х годов сотрудник MIT Клейнрок предложил модель перехода от сети передачи сообщений к сети передачи пакетов. Можно сказать, что Клейнрок является идеологическим основателем всего действующего сетевого обеспечения и интернета, в частности. В подавляющем большинстве случает все сети являются сетями коммутации пакетов.

Взаимодействие осуществляется путем передачи и приема сообщений. При отправке сообщения сетевое устройство разделяет каждое сообщение на некоторые порции фиксированного размера. К этим порциям добавляется служебная информация, включающая в себя координаты получателя, номер сообщения, номер пакета в сообщении, информация об отправителе и т.д. Совокупность служебной информации и содержательных данных называется пакетом. Пакет – порция данных фиксированного размера.

Сеть коммутации пакетов функционирует по принципу "горячей картошки". Отправитель отправляет пакет, так же как и в сети коммутации сообщений, первый свободный коммуникационный узел по пути к получателю получает пакет, если нет возможности его сразу отправить, этот коммуникационный узел сохранит пакет. За счет того, что пакеты детерминированного размера, есть возможность рассчитать статистически характеристики коммуникационных устройств, исходя их пропускной способности каналов, количества абонентов в сети,

Пакеты происходят к получателю в произвольном порядке. Появляется функция сбора пакетов в сетевом устройстве получателя.

Базовое свойство коммуникации пакетов: сеть коммутации пакетов может быть использована для решения коммуникационных в территориально распределенных компьютерных сетях, обладающих недетерминированным качеством коммуникационных линий.

Именно это базовое свойство сетей коммуникации пакетов позволило построить интернет.

Преимущества:

- Так как известна топология сети и характеристики ее элементов, то возможно определение требований в коммутационных узлах

  ⇒ возможна оценка размера буфера и времени доставки пакетов

Недостатки:

- Увеличение трафика(накладные расходы) из-за наличия заголовочной информации. Полезная пропускная способность 10
- Проблема сборки пакетов
- Требуется сложное программное обеспечения

Если пакет по одному пути не может дойти, он пойдет по другому пути. Это позволяет осуществлять взаимодействие в сетях с недетерминированным качеством.

В 1967 году был начат проект по распроеделенной территориально продолжительной компьютерной сети, которое получило название ARPANET. Этот военный проект затем был превращен в интернет.

Модель ISO/OSI организации взаимодействия в сети

Необходимость появления компьютерных сетей сформировалась в конце 1950-х, начале 1960-х. Тогда же возникли первые разработки многомашинных ассоциаций, которые затем развились в компьютерные сети.

Рассмотрим некоторые корпоративные разработки, которые создавались для решения задач автоматизации и распределенной обработки информации в рамках отдельных компаний. Все решения, которые использовались в этих корпоративных сетях были уникальных и не являлись унифицированными, что не позволяло объединять сети. Соответственно, возникла проблема стандартизации и унификации.

В 1970-х годах ISO (Международная организация по стандартизации) объявила о начале проекта ISO/OSI. Идея этого проекта – сформулировать требования и содержания стандартизации всего взаимодействия, которое происходит в компьютерный сетях, от взаимодействия на уровне сред передачи данных(аппаратном уровне) до взаимодействия на прикладном уровне, где ботают конкретные прикладные системы. Было предложено разделить все взаимодействие на 7 уровней.

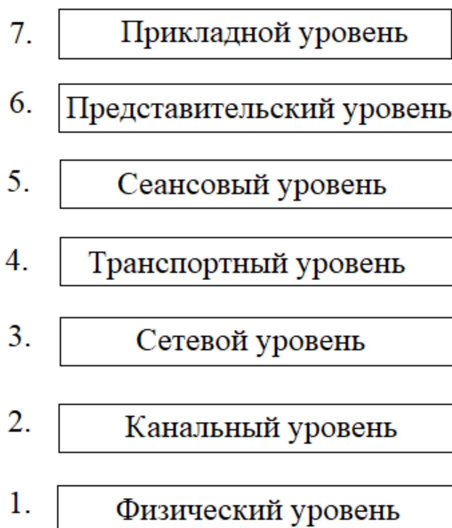

Рис. 8.1. Уровни взаимодействия в сети

Самый нижний уровень соответствует уровню физической среды передачи данный. На этом уровне специфицируются те физические средства, которые используются для передачи данных.

На прикладном уровне специфицируются прикладные сетевые приложения. Пример: средства коммуникаций.

Логическое взаимодействие по протоколу

Рассмотрим два сетевых устройства. Такие устройства подключаются к физической среде передачи данных. Если они взаимодействуют по модели ISO/OSI, в них должны быть реализованы уровни от первого до предельного. Реализованные на конкретном сетевом устройстве уровни называются *стеком проколов*.

Уровни сетевых устройств могут взаимодействовать с одноименными уровнями других сетевых устройств. Правила взаимодействия этих уровней называются *сетевыми протоколами*.

Каждый запрос одного уровня i-го уровня к i-му уровню другого сетевого устройства реализуется посредством уточнения и обработки через все уровни стека протоколов. То есть запрос должен быть переведен на нижестоящие уровни этого сетевого устройства, дойти до физической среды, передаться на соответствующее устройство и подняться на нужный уровень. Для реализации этого пути используются сетевые интерфейсы.

Сетевой интерфейс – это правило взаимодействия вышестоящего уровня с нижестоящим. Каждый нижестоящий уровень для вышестоящего уровня предоставляет

ФАКУЛЬТЕТ ВЫЧИСЛИТЕЛЬНОЙ МАТЕМАТИКИ И КИБЕРНЕТИКИ МГУ ИМЕНИ М.В. ЛОМОНОСОВА

сервисы (набор операций). Каждый запрос протокола отображается в последовательность обращений, использующих интерфейсы и сервисы.

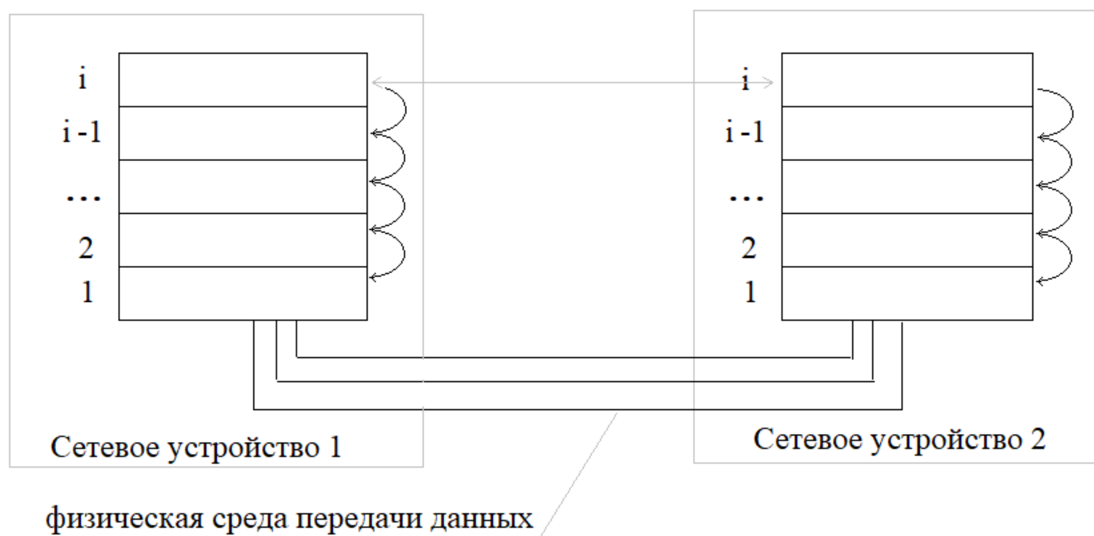

Рис. 8.2. Логическое устройство взаимодействия сетевых устройств по i-му протоколу

Вернемся к семиуровневой модели.

1. На физическом уровне осуществляется передача неструктурированного потока данных. На данном уровне происходит стандартизация сигналов и аппаратных сочленений.

2. На канальном уровне обеспечивается решение задач передачи данных по физической линии, доступность канала(занят/свободен), синхронизация, обнаружение и обработка ошибок. Канальный уровень – уровень пакетного взаимодействия. На канальном уровне решается задача локальной адресации сетевых устройств в рамках локальной сети.

3. На сетевом уровне происходит взаимодействие между сетями, сетевыми устройствами. Сетевые устройства могут принадлежать различным локальным сетям.

4. Транспортный уровень. Обеспечивается взаимодействие между сетевыми программами, которые работают совместно на разных сетевых устройствах. На транспортном уровне принимается решение о выборе типа транспортных услуг. Пример: транспорт с установлением виртуального канала. Обеспечивается функция контроля доставки пакетов, реализуется виртуальный канал и, соответственно сеанс связи. Контролируется доставка всех пакетов, которые идут по транспорту с установлением виртуального канала. Доставка контролируется с

помощью пакетов уведомлений, которые отправляет принимающая сторона в ответ на получение пакета.

Транспорт с установлением виртуального канала является надежным, хорош для работы в условиях недетерминированных коммуникаций. Очень много накладных расходов.

5. На сеансовом уровне определяются функции сеанса, начало/конец, подтверждение полномочий, решаются средства, предназначенные для обеспечения диагностирования и обработки ошибок.

6. Представительский уровень нужен для унификации представления данных.

7. Прикладной уровень определяет целевую функцию создания компьютерной сети. На прикладном уровне стандартизуются все те средства построения и использования прикладных сетевых систем.

Эта модель была предложена как некий вариант для реализации. Но она не получила полной реализации, так как к моменту появления модели ISO/OSI появилось семейство протоколов TCP/IP, которые стали реальным инструментом для построения сетевых конфигураций.

Лекция 9. Организация сетевого взаимодействия

Модель ISO/OSI взаимодействия в сети

Данная модель была предложена международной организацией стандартизации в начале 1970-х годов с целью стандартизации и унификации. Она представляла собой семь взаимодействующих уровней, на которые предполагалось разделение любого взаимодействия внутри компьютерных сетей. Иерархически эти уровни располагались от нижнего физического уровня, который связан с реальной физической средой передачи данных, на котором стандартизовывались сигналы,аппаратные средства сопряжения и прочее, что может характеризовать физическую среду передачи информации, до прикладного уровня, на котором стандартизовывались и унифицировались средства взаимодействия программ в рамках компьютерных сетей, которые решали прикладные задачи. Прикладной уровень – целевой уровень взаимодействия, который обеспеспечивает реализацию прикладных сетевых приложений.

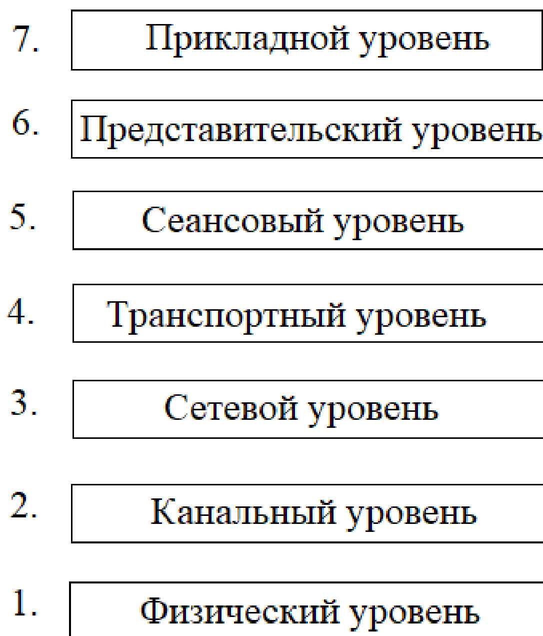

Рис. 9.1. Уровни модели ISO/OSI

Эта модель предполагала, что в сети взаимодействуют объекты на одноименных уровнях. Правило взаимодействия объектов на одинаковых уровнях называлось "сетевой протокол".

Реальные сетевые устройства связаны друг с другом связаны только на физическом уровне. Объекты, находящиеся на остальных уровнях, связаны виртуально.

ФАКУЛЬТЕТ ВЫЧИСЛИТЕЛЬНОЙ МАТЕМАТИКИ И КИБЕРНЕТИКИ МГУ ИМЕНИ М.В. ЛОМОНОСОВА

Это взаимодействие реализуется посредством отображения в последовательность запросов на нижестоящие уровни в рамках машины, в которой обрабатывается запрос, затем по физической среде информация передается на объект-получатель, затем информация о запросе передается от физического уровня на уровень, на котором происходит взаимодействие.

В этой модели работают:

- протоколы – правила взаимодействия на одноименном уровне;
- интерфейсы – правила взаимодействия вышестоящего уровня с нижестоящим;
- сервисы и службы – функционал, который предоставляется нижестоящим уровнем для вышестоящего.

В 1960-х годах активно стал развиваться проект ARPNET американского агентства стратегических разработок DARPA, который затем развился в интернет. Этот проект был создан для надежного управления системой в случае недетерминированного качества коммуникационной среды. Создавалось средство надежной коммуникации для ведения войны.

Начались работы по созданию компьютерных сетей, взаимодействие которых основывалось на пакетной обработке. Подход пакетного взаимодействия лег в основу надежных, устойчивых к недетрминированным характеристикам коммуникационной среду, сетям компьютеров. Есть возможность отслеживать, дошел ли полностью пакет до получателя. Этот подход лег в основу сети ARPANET.

Таким образом к моменту создания ISO/OSI уже были разработки семейства TCP/IP. Влияние модели ISO/OSI на TCP/IP – логическая стандартизация.

Соответствие модели ISO/osi модели семейства протоколов TCP/IP

Семейство протоколов TCP/IP – четырехуровневая модель (см. рис. 9.2).

Для семейства протоколов TCP/IP существуют два ключевых протокола:

1. IP – это протокол, который предоставляет средства и возможности унифицированного именования сетевых устройств во всевозможных сетях;
2. TCP – транспортный протокол (пакетный протокол) с установлением виртуального канала. Этот протокол обеспечивает надежное взаимодействие (передачу информации) в рамках сетей, в которых коммуникационная среда недетерминированна.

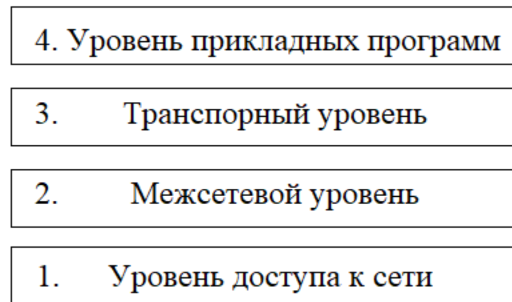

Рис. 9.2. Уровни семейства протоколов TCP/IP/

Эти протоколы и дали название семейству протоколов TCP/IP.

Рассмотрим уровни этой модели.

1. Уровень доступа к сети – это спецификация физического взаимодействие и решение задач адресации в рамках локальной сети. Этот уровень TCP/IP соответствует канальному и физическому уровню модели ISO/OSI.

2. Межсетевой уровень – уровень протокола IP(Internet Protocol). соответствует сетевому уровню протоколов ISO/OSI. Две задачи:

   - модель именования;
   - задача построения маршрута

3. Транспортный уровень соответствует сеансовому уровню и транспортному уровню протоколов ISO/OSI.

   На транспортном уровне есть две модели организации взаимодействия:

   - взаимодействие с установлением виртуального канала (протокол TCP);
   - протокол UDP, который организует взаимодействие без установления виртуального канала;

4. Уровень прикладных программ – спецификация взаимодействия на прикладном уровне, на котором решаются задачи и функции уровня представления и уровня прикладных программ модели ISO/OSI. Функция представления интегрирована с решением задач стандартизации и унификации прикладного взаимодействия.

Уровень модели TCP/IP	Уровень модели ISO/OSI
1. Уровень доступа к сети Специфицирует доступ к физической сети.	Канальный уровень Физический уровень
2. Межсетевой уровень В отличие от сетевого уровня модели OSI, не устанавливает соединений с другими машинами.	Сетевой уровень
3. Транспортный уровень Обеспечивает доставку данных от компьютера к компьютеру, обеспечивает средства для поддержки логических соединений между прикладными программами. В отличие от транспортного уровня модели OSI, в функции транспортного уровня TCP/IP не всегда входят контроль за ошибками и их коррекция. TCP/IP предоставляет два разных сервиса передачи данных на этом уровне. Протокол TCP, UDP.	Сеансовый уровень Транспортный уровень
4. Уровень прикладных программ Состоит из прикладных программ и процессов, использующих сеть и доступных пользователю. В отличие от модели OSI, прикладные программы сами стандартизуют представление данных.	Уровень прикладных программ Уровень представления данных

Рис. 9.3. Соответствие модели ISO/OSI модели семейства протоколов TCP/IP

Взаимодействия между уровнями протоколов TCP/IP

Все протоколы пакетные. На каждом уровне свой пакет со своей структурой и со своим наименованием.

Перемещая информацию от вышестоящего уровня к нижестоящему, прикладное сообщение разбивается на пакеты соответствующего уровня, которые затем передаются на следующий уровень. На этом уровне снова пакеты разбиваются, либо добавляется информация, которая образует пакет нового уровня, и т.д. (см. рис. 9.4)

ФАКУЛЬТЕТ ВЫЧИСЛИТЕЛЬНОЙ МАТЕМАТИКИ И КИБЕРНЕТИКИ МГУ ИМЕНИ М.В. ЛОМОНОСОВА

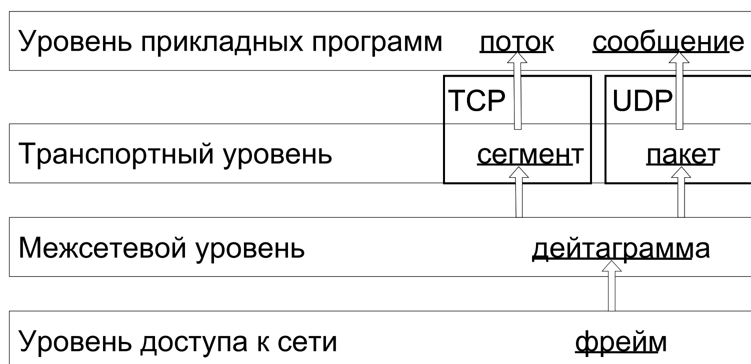

Рис. 9.4. Взаимодействие между уровнями протоколов TCP/IP

Уровень доступа к сети

Уровень доступа к сети семейства протоколов TCP/IP представляет тоже семейство протоколов, каждый их которых реально формализует и стандартизует систему и правила использования конкретной физической среды передачи данных. Базовым протоколом этого смемейств аявляется семейство протоколов Ethernet. Этот протокол является широковещательным протоколом, который использует общую шину. Протокол к Ethernet можно классифицировать как протокол, обеспечивающий множественный доступ с контролем и обнаружением конфликтов.

Единая шина означает, что к единому физическому каналу может подключаться группа сетевых устройств. Каждое из этих устройств может слушать информацию, которая перемещается по общей шине и забирать ту информацию, которая целевым образом адресуется к этому устройству. По такой шине могут идти *персонифицированные сообщения* или *широковещательные сообщения*.

Эта модель обеспечивает множественный широковещательный доступ к шине, в котором отсутствует централизация, нет управляющего устройства.

Такая модель способствует возникновению конфликтов, когда два и больше устройств единовременно начинают осуществлять передачу информации. Если возник конфликт, прерывается передача у обоих конфликтующих сторон. Каждая из сторон ожидает пропуска.

Этапы развития среды передачи данных.

- Протокол Ethernet появился в 1970-х годах, первой средой передачи данных была среда «толстый Ethernet» – медная полоса, к которой приклепывались сетевые устройства.

- «тонкий Ethernet» – аналог телевизионного кабеля (медная жила, которая имеет экран) с унифицированнными разъемами, к которым подключаются сетевые

устройства.Появление экранированного кабеля позволяло избавиться от значительной части помех.

- Затем средой передачи данных стала «витая пара». Эти пары снижают возникающие в проводах помехи
- Физическая среда – оптоволокно. Сигнал передается уже светом.

Параллельно развивались беспроводные средства передачи данных. Первым беспроводным каналом для Ethernet был протокол Radio Ethernet.

Внутри протокола Ethernet реализуется своя внутренняя адресация. Она основывается на том, что есть международное соглашение, которое регламентирует при создании аппаратных Ethernet адаптеров присвоение уникального имени каждому из них.

Протокол IP. Межсетевой уровень

Это межсетевой уровень. Основная задача на межсетевом уровне – именование. Для именования используется IP-адресация.

Две модели IP-адресов:

- разбиение и обратная сборка дейтаграмм

1. короткая IP-адресация появилась вместе с самим протоколом;

•IP - протокол без логического установления соединения

•Протокол IP не обеспечивает обнаружение и исправление ошибок

2. длинная IP-адресация (64-х разрядная).

система адресации протокола **TCP / IP**

Разберем 32-х битную модель.

Рис. 9.5. Система адресации протокола IP

3. Транспортный уровень. Обеспечивает доставку данных от компьютера к компьютеру, обеспечивает средства для поддержки логических соединений между прикладными программами. В отличие от транспортного уровня модели OSI, в функции транспортного уровня TCP/IP не всегда входят контроль за ошибками и их коррекция. TCP/IP предоставляет два разных сервиса передачи данных на этом уровне. Протокол TCP, UDP.

•Протокол контроля передачи (TCP, Transmission Control Protocol) - обеспечивает надежную доставку данных с обнаружением и исправлением ошибок и с установлением логического соединения.

•Протокол пользовательских дейтаграмм (UDP, User Datagram Protocol) – отправляет

- *Класс А.* Младшие(правые) три байта IP-адреса класса А – номер сетевого устройства. Для класса А в старшем байте сраший (левый) разряд равен нулю. Оставшиеся семь разрядов используются для нумерации сетей . Таких сетей в мире может быть 2^7. Это гигантские сети, которые внутри себя могут иметь порядка $2^2$4 сетевых устройств. Обладателями сетей класса А являются ведущие мировые IT-корпорации.

- *Класс В.* Для идентификации адреса используются два старших бита IP-адреса. Признаком адреса класса В является комбинация 10. Младшие два байта используются для номерации сетевых устройств в сети. Таких устройств в одной сети может быть $2^1$6. Старшие два байта используются для нумерации сетей. Таких сетей может быть $2^1$4.

- Класс С. Для идентификации адреса используются три старших бита IP-адреса. Признаком адреса класса С является код 110, расположенный в трех старших битах адреса. Для нумерации сетевых устройств класса С используется младший байт, т.е. в сети класса С может быть 2^8 устройств. Для нумерации сетей используются три старший байта. Количество сетей в мире определяется порядком $2^2$1.

Есть не только содержательные адреса, но и служебные.

Задача маршрутизации

На межсетевом уровне используется задача маршрутизации. Для организации взаимодействия между сетями используются шлюзы.

Определение. *Шлюз* – сетевое устройство, обладающее двумя и более сетевыми адаптерами и IP-адресами. В функции шлюза входит организация маршрута, по которому нужно осуществить передачу пакета за пределы локальной сети.

Шлюз– специализированное сетевое устройство, которое единовременно подключено к двум и более локальным сетям. В рамках шлюза стек протоколов может быть организован не из четырех уровней TCP/IP, а только из двух: из уровня, который обеспечивает доступ к сети, и уровня межсетевого взаимодействия.

Разберем пример. Если из сетевого компьютера А1 нужно передать информацию на сетевой компьютер А2, который находится не в локальной сети А1. Запрос от А1 пойдет в его локальную сеть, сетевое устройство "шлюз"получит этот запрос и выберет маршрут (сеть, по которой осуществится передача пакета дальше). Через шлюзы запрос дойдет до сетевого устройства-получателя.

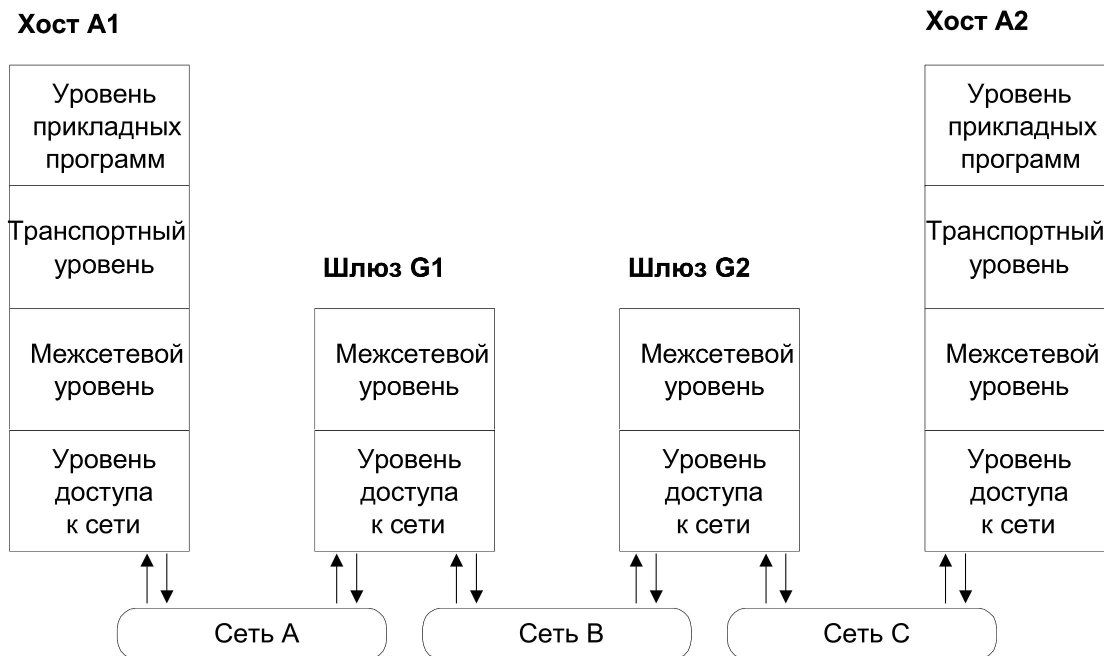

Рис. 9.6. Маршрутизация дейтаграмм

Транспортный уровень

Обеспечивает доставку данных от компьютера к компьютеру, обеспечивает средства для поддержки логических соединений между прикладными программами. В отличие от транспортного уровня модели OSI, в функции транспортного уровня TCP/IP не всегда входят контроль за ошибками и их коррекция. TCP/IP предоставляет два разных сервиса передачи данных на этом уровне. Протокол TCP, UDP.

1. Протокол контроля передачи (TCP, Transmission Control Protocol) обеспечивает установление виртуального канала, за счет чего обеспечивается контроль факта доставки пакета. В случае получения пакета сетевое по контрольным суммам проверяет целостность данных. По результатам проверки отправляется устройству-отправителю пакет-уведомление. Если сетевое устройство-отправитель получает уведомление об ошибке, отправка пакета повторяется (этот пакет пойдет уже по другому маршруту физических коммуникаций). Если же устройство-отправитель не получает пакет-уведомление в течение некоторого промежутка времени после отправки, считается, что пакет потерян, устройство-отправитель повторяет передачу. Настройки протокола естественным образом определяют количество итераций/повторов, через которое определяется, что нет связи с устройством отправителем, и прекращаются попытки передачи пакета.

Прокотол TCP хорош тем, что за счет коммуникационной среды можно попытаться обойти аварийные точки в передаче данных.

→ Таким образом устройство-получатель проверяет надежную доставку данных с обнаружением и исправлением ошибок и с установлением логического соединения.

2. Протокол пользовательских дейтаграмм (UDP, User Datagram Protocol) отправляет пакеты с данными, «не заботясь» об их доставке.

   Протокол UDP хорош для условий взаимодействия в рамках сети с детерминированными характеристиками коммуникационной сети. Такой сетью является локальная сеть.

   → При таком взаимодействии функции контроля берет на себя получатель.

Определение. Дейтаграмма – пакет протокола UDP.

Уровень прикладных программ

Протоколы уровня прикладных программ базируются на основе транспортных протоков TCP или UDP. Они обеспечивают доступ и работу с заведомо корректной информацией, которая осуществляется в сети Интернет.

- Протоколы, опирающиеся на TCP:
  - TELNET (Network Terminal Protocol);
  - FTP (File Transfer Protocol);
  - SMTP (Simple Mail Transfer Protocol) используется для отправления/ получения почты;
- Протоколы, опирающиеся на UDP:
  - DNS (Domain Name Service) – протокол доменных имен. Самой правый домен (нулевого уровня) – домен страны, затем идет домен организации и т.д. Протокол DNS обеспечивает трансляцию мнемонического имени в IP-адреса;
  - RIP (Routing Information Protocol);
  - NFS (Network File System) – сетевая файловая система . Этот протокол позволяет объединить ресурсы сетевых UNIX-систем и сделать общее файловое пространство в рамках локальной сети (детерминированной среды).

Основы архитектуры операционных систем

Под операционной системой понимаются два уровня иерархии: уровень управления физическими ресурсами и уровень управления виртуальными/логическими ресурсами.

Определение. *Операционная система* – это комплекс программ, обеспечивающий контроль за существованием, распределением и использованием ресурсов ВС. Расшифруем:

- контроль – управление;
- существование – реализация логических и/или виртуальных ресурсов. В рамках операциооной системы реализуются качества виртуальных ресурсов, таких ресурсов, в которых часть эксплуатационных характеристик (или все) реализуется программно;
- распределение – все ресурсы имеют характеристики, которые определяют пределы. Следовательно есть задача корректного распределения ресурсов между потребителями. Эта задача решается в рамках операционной системы, которая реализуется посредством двух моделей
  - предварительное выделение потребителю запрашиваемого ресурса;
  - выделение ресурса по запросу.
- использование – проблема организации контроля за использованием, например, учет времени ЦП.

Процесс – это одно из базовых понятий. Синонимами процесса являются управление задачами, заданиями. Каждая ОС рассматривает некоторую сущность, которую мы называем процессом. Это элементарная единица, которая осуществляет управление ОС.

Определение. *Процесс* – это совокупность машинных команд и данных, которые обрабатываются в рамках ВС и обладают правами на доступ/использование некоторых ресурсов ВС (которые были выделены либо предварительно, либо по запросу).

ФАКУЛЬТЕТ ВЫЧИСЛИТЕЛЬНОЙ МАТЕМАТИКИ И КИБЕРНЕТИКИ МГУ ИМЕНИ М.В. ЛОМОНОСОВА

Лекция 10. Основы архитектуры ОС

Основы архитектуры ОС

Определение. *Процесс* – это совокупность машинных команд и данных, обрабатываемых в рамках данной вычислительной системы (ВС) и обладающих правами на некоторый набор ресурсов ВС .

Совокупность команд и данных называют *исполняемой программой*, которой поставлены в соответствие некоторые ресурсы. При формировании процесса используется программа, которая должна исполняться. В процессе формирования запуска этой программы, ей ставятся в соответствие некоторые ресурсы, например, оперативная память.

Процесс может обладать ресурсами по двум моделям:

1. Ресурсы, монопольно принадлежащие данному процессу. К таким ресурсам можно отнести реальное время центрального процессора, которое использовал процесс, оперативную память;

2. Разделяемые ресурсы. Ресурс может единовременно принадлежать двум и более процессам.

На сегодняшний день, подавляющее количество операционных систем являются мультипроцессными.

Определение. *Мультипроцессная операционная система* – такая операционная система, которая позволяет единовременно обрабатывать два и более процесса.

Если такая операционная система работает на однопроцессорной вычислительной системе, то из обрабатываемых процессов один может исполняться, часть может ожидать завершения обменов, которые они заказали, а остальные могут быть готовы к исполнению и ожидать предоставления ресурса ЦП.

Требования к ОС

Требования к операционной системе зависят от области применения операционной системы.

Операционная система должна удовлетворять основным требованиям:

1. *Надежность.* Количество ошибок, связанных с программными ошибками, должно быть минимизировано. ОС должна быть, по меньшей мере, так же надежна, как аппаратура, на которой она работает.

2. *Защита.* Защита должна обеспечивать исключение возможности несанкционированного доступа к данным и другим ресурсам вычислительной системы. При работе в однопользовательноской мультипроцессной системе (пример: смартфон) нужна защита внутри системы, для многопользовательской системы защита внутри системы крайне важна. Также система должна обеспечивать защиту от внешнего проникновения, когда внешний пользователь получает права пользователя целевой вычислительной системы.

   ⇒ Система должна обеспечивать комплексую защиту от внешнего несанкционированного доступа.

3. *Эффективность.* Удовлетворение критериям эффективности.

4. *Предсказуемость.* Известны заранее проблемы и последствия различных действий. Система должна функционировать предсказуемо в форс-мажорных ситуациях, при выходе из стоя аппаратных или программных компонентов.

Структурная организация ОС

Рассмотрим иерархическую схему по структуре компонент, функционируюзих в вычислительной системе.

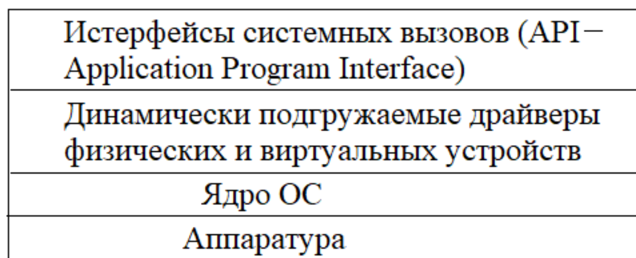

Рис. 10.1. Структура ОС

Определение. *Ядро* – (kernel) – резидентная часть ОС, работающая в режиме супервизора. («обычно» работает в режиме физической адресации).

Определение. *Системный вызов* – обращение к ОС за предоставление той или иной функии (возможности, услуги, сервиса).

- В основе всего находится аппаратура. Операционная система очень сильно интегрирована с аппаратной частью.

- Ядро – часть программ операционной системы, которая постоянно находится в оперативной памяти, работает в режиме операционной системы или режиме супервизора (тот режим процессора, который позволяет выполнять все

ФАКУЛЬТЕТ ВЫЧИСЛИТЕЛЬНОЙ МАТЕМАТИКИ И КИБЕРНЕТИКИ МГУ ИМЕНИ М.В. ЛОМОНОСОВА

возможные команды процессора). Во многих случаях ядро работает в режиме физической адресации, в ядре не работает виртуальная адресация.

Ядро реализует некоторый базовый набор функций, который реализуется посредством включенных в ядро драйверов. Состав этих функций зависит от конкретной операционной системы.

- Драйвера физических и виртуальных ресурсов, которые могут подгружаться к ядру в процессе функционирования системы. Современные операционные системы предусматривают поддержку большого количества внешних устройств. Многие из этих устройств относятся к категории Plug and Play. В случае подключения такого устройства к системе, ядро получает прерывание о том, что появилось новое устройство, процесс обработки этого прерывания включает поиск в базе данных ОС драйвера этого устройств. В случае наличия драйвера он подгружается в память, устройство может начать работать. Таким образом, драйвер находится в памяти только временно, когда нужно обеспечить работоспособность устрой. Такие драйвера называются *нерезидентными*. Нерезидентные драйверы работают в режите ОС.

- Интерфейсы системных вызовов. Большинство операционных систем обеспечивают возможность обращения из процессов пользователей за функциями ОС. Эти обращения определенным образом унифицированы и называются системными вызовами. При обращении к системному вызову происходят действия аналогичные обработке прерывания, т.е. управление передается ОС, ОС выполняет определенные действия. Примеры системного вызова: Открыть/закрыть файл, изменить имя файла и т.д.

  Наличие системных вызовов показывает, что система работает в пользовательском режиме пока выполняются обычные команды пользовательского процесса, а когда ядро ОС реализует обращение к супервизору от имени какого-то из процессов, система начинает работать в системном режиме.

Архитектура ядра

Структурная организация ядра может быть различных моделей. Исторически первая модель – *монолитное ядро.*

1. *Монолитное ядро* представляет собой реализацию, в которой все компоненты связаны друг с другом, поэтому выделить какой-то отдельный компонент из программного кода тяжело, так как у него имеется много различных взаимосвязей. Есть

ФАКУЛЬТЕТ ВЫЧИСЛИТЕЛЬНОЙ МАТЕМАТИКИ И КИБЕРНЕТИКИ МГУ ИМЕНИ М.В. ЛОМОНОСОВА

значительное количество функциональных узлов, которые связаны друг с другом напрямую без структуризации и бессистемно. Такая организация позволяет получить наиболее эффективные с точки зрения времени решения.

- Плюсы: быстро.
- Минусы: нетехнологично. Замена какого-то из модулей требует пересборки всего ядра.

2. *Многоуровневое ядро* представляется в виде иерархии фиксированного количества уровней, связанного между собой межуровневыми интерфейсами. Каждый уровней связан с одним вышестоящим и одним нижестоящим уровнями.

- Плюсы: такая организация позволяет повысить эффективность технологического обслуживания ядра.
- Минусы: фиксированная коммуникация накладывает дополнительные расходы, снижается эффективность с точки зрения скорости за счет структуризации.

3. *Микроядерная архитектура.* Есть микроядро, в котором реализован минимальный набор функций: реализация драйвера управления оперативной памятью и драйвера процессора. В микроядре реализованы интерфейсы для подключения прочих драйверов. К микроядру может подключаться произвольное количество драйверов как физических устройств, так и виртуальных.

Микроядерная архитектура позволяет подключать идентичные драйвера. Например, в одной и той же модели может быть реализовано два и более файловых систем, что невозможно ни в монолитной системе, ни в многоуровневой, где зафиксированы количество, состав и характеристики реализованных возможностей и функций.

- Плюсы: микроядерная система гибкая, как конструктор.
- Минусы: большие накладные расходы, эффективность с точки зрения производительности достаточно невысокая.

⇒ Решение – гибридные системы, в архитекуре которых используются возможности микроядерных систем, монолитных и многоуровневых систем.

Логические функции ОС

Каждая операционная система реализует некоторый предопределенный набор логических функций. Есть типовой набор логических функций, который реализуется в операционной системе. К нему относятся следующие функции:

- Управление процессами – создание процесса и организация его обработки и выполнения. Т.е. создание процесса, выделение ресурсов, обеспечение защиты от несанкционированного доступа, завершение работы процесса.
- Управление оперативной памятью. Современная аппаратура процессора может предоставлять возможность варьировать различные модели управления оперативной памяти.
- Планирование. Использование ресурсов значительным количеством процессов приводит к тому, что возникает конкуренция за доступ к ресурсам. Одной из основных функций ОП является функция планирования, когда на входе имеем поток запросов к ресурсам, а на выходе имеем структурированную очередь.
- Управление устройствами и файловая система(ФС). Модель, которая используется для организации файлов, существенно определяет эксплуатационные характеристики операционной системы.
- Сетевое взаимодействие. Любая система ориентирована на сетевое взаимодействие.
- Безопасность. Защита информации и ресурсов от несанкционированнного доступа.

Типы ОС. Пакетная ОС

Рассмотрим типы операционных систем или модели функционирования операционных систем с точки зрения планирования времени использования работы центрального процессора.

Далее разберем три типа систем.

1. Пакетная система
2. Система разделения времени
3. Система реального времени

Пусть есть некоторая система, работающая в мультипрограммном режиме, которая предназначена для эффективного выполнения процессов, которые требуют значительных расчетных операций. Критерием эффективности работы такой системы может быть максимальная загрузка центрального процессора. Время центрального процессора соотнесем с временем, потраченным на выполнение команд процессов, это отношение должно быть близко к единице.

Для этого надо минимизировать накладные расходы, связанные с работой операционной системы. То есть ОС должна включаться в работу только когда это необходимо.

Система планирования времени процессора пакетной ОС переключает выполнения процессов только в случае одного из трех событий:

- выполнение процесса завершено;
- возникло прерывание (в случае, если прерывание требует значительного времени, ОС может переключиться на другой процесс);
- зацикливание процесса.

Эффективность загрузки процессора для пакетной системы составляет 95-98 % .

Системы разделения времени

В некотором смысле альтернативой пакетной системы является система разделения времени.

Будем считать, что процессы, которые должны обрабатываться данной мультипроцессорной системой, не расчетные, а интерактивные, например, текстовый редактор. При работе в интерактивной системе критерием эффективности является ощущение, что мы работаем в однопользовательском режиме (мгновенный ответ на запрос). Для мультипроцессной системы, которая работает с несколькими процессами, важной характеристикой является время отклика.

Определение. *Квант времени ЦП* – некоторый фиксированный ОС промежуток времени работы ЦП.

Когда некоторый процесс становится исполняемым процессом, ОС выделяет для этого процесса квант веремени ЦП.

Переключение выполнения процессов происходит в следующих случаях:

- исчерпался выделенный квант времени исполняемого процесса;
- выполнение процесса завершено;

- возникло прерывание ;
- зацикливание процесса.

Чтобы создавалась иллюзия монопольного использования вычислительной установки, надо следующим образом выделить кванты времени ЦП: кванты времени маленькие, их размер статистически достаточен для выполнения одного запроса пользователя. Время ожидания пользователя в случае обрабатывания N процессов составляет N квантов времени, соответственно можно рассчитать задержку, исходя из мощности процессора и количества пользователей.

Минусом этой системы является масса накладных расходов, вызванных переключением с процесса на процесс. Эффективность загрузки

Если эффективность загрузки процессора для пакетной системы составляет 95-98% , то для системы разделения времени эта доля может падать до 20-30%.

Если увеличить квант времени до бесконечности (5-10 минут), система превратится в пакетную систему. Этот используется в комбинированных системах.

Системы планирования

Для систем, которые обслуживают некоторое количество пользователей, генерирующих поток из процессов разных типов (интерактивных процессов, процессов, связанных с вычислительными задачами), хорошим решением является комплексное планирование. Рабочее время делится на периоды времени по несколько часов.

- В первый промежуток времени (8:00-18:00) интерактивный процесс получает наивысший приоритет и получает свой квант времени, а остальные процессы прерываются, если интерактивных процессов нет, следующим по приоритету является отладочный процесс, который получает уже другой квант времени.
- Во второй промежуток времени (18:00-24:00) наибольшим приоритетом обладают отладочные процессы, вторым приоритетом обладают расчетные задачи, а наименьшим приоритетом обладают интерактивные процессы.
- В третий промежуток времени (00:00-8:00) наивысшим приоритетом обладают расчетные задачи, затем отладка и интерактивные процессы.

Если в промежуток времени появляется наиболее приоритетный из процессов, то выполнение процесса с низшим приоритетом прерывается, и процессор передается в распоряжение наиболее приоритетного процесса.

Лекция 11. Процесс в ОС

Разновидности операционных систем (ОС). Пакетная

Ранее рассматривались модели функционирования ОС с точки зрения принципов использования времени ЦП, которое заложено в основу компонентов планирования моделей ОС.

По такому принципу можно выделить несколько разновидностей ОС.

- *Пакетная система* предназначена для выполнения процессов, связанных с вычислением, которые требуют для своего выполнения большие промежутки времени.

  Критерием эффективности такой системы является степень загрузки процессора, как много времени работы процессора тратится на выполнение пользовательских команд.

  Пакетная ОС минимизирует действие операционной системы. Управление работой вычислительной системой от пакетной системы передается ОС только в трех случаях:

  - выполнение процесса завершено;
  - возникло длинное прерывание, требующее обработки (например, обращение к внешнему устройству) ;
  - зацикливание процесса (в пакетных системах регламентируется предельное время работы процессора для каждого процесса).

  ⇒ В пакетной системе минимальное количество передач управления между исполняемыми процессами и ОС, т.е. накладные расходы, связанные с обработкой переключений, минимизированы.

  Нормальной характеристикой загрузки центрального процессора является 95-98%.

- *Системы разделения времени* направлены на решение интерактивных задач, которые предполагают взаимодействие с человеком-пользователей в оперативном режиме. В таких системах разделяется время работы процессора на некоторые временные промежутки, кванты времени. ОС следит за возникновением событий, которые могут появиться в пакетной системе и за исчерпанием кванта времени. Появилась еще одна точка переключения.Переключение выполнения процессов происходит в следующих случаях:

ФАКУЛЬТЕТ ВЫЧИСЛИТЕЛЬНОЙ МАТЕМАТИКИ И КИБЕРНЕТИКИ МГУ ИМЕНИ М.В. ЛОМОНОСОВА

  – исчерпался выделенный квант времени исполняемого процесса;
  – выполнение процесса завершено;
  – возникло прерывание ;
  – зацикливание процесса.

  Для интерактивных процессов кванты времени рассчитываются в зависимости от мощность процессора и количества пользователей.

  Критерий эффективности работы систем разделения времени определяется временем отклика системы.

  Для систем разделения времени характеристика загруженности ЦП снижается до 20-30%.

- *Гибридные системы* могут быть организованы так, что для каждого класса процессов в каждый момент времени определены два параметра: приоритет процессов класса и квант времени для процессов определенного класса. Эти параметры могут быть динамическими.

  Система планирования организуется таким образом, что если система работает и выполняет какой-то процесс, то в случае появления процесса, готового к обработке, имеющего более высокий приоритет, текущий процесс может быть прерван, и управление передается более приоритетному процессу.

ОС реального времени

Операционные системы реального времени – это такие операционные системы, в которых выделен набор событий, при возникновении которых гарантируется их обработка за время, не превышающее некоторый предел. Определение. *Системы реального времени* являются специализированными системами, в которых все функции планирования ориентированы на обработку фиксированного набора событий за время, не превосходящее некоторого предельного значения.

Пример системы реального времени. Допустим, молоко нагревается, а затем закипает. Система реального времени должна обработать факт достижения температуры кипения за время, которое не превосходит Δt, за которое молоко вскипит.

Можно выделить несколько типов систем:

- Системы с *жестким реальным временем* – такие системы, в которых временные промежутки ограничены и обработка должна гарантированно произойти в этот промежуток времени. Например, системы управления в самолете. Такие

системы сугубо специфические. Они строятся и разрабатываются для конкретного применения. Во многих случаях для систем жесткого реального времени используются некие модификации UNIX систем.

- Системы *мягкого реальным временем* – системы, в которых длительность обработки не смоль критична. Например, посудомоечная машина.

  Для систем мягкого реального времени для основы можно использовать самые разные системы.

Современные системы обеспечивают сетевое взаимодействие, распределенное взаимодействие.

Распределенная операционная система – это такая машина, которая работает на многомашинном/многопроцессорном комплексе, в рамках которой реализуются распределенные функции ОС. Пример: в кластерной вычислительной системе, являющейся суперкомпьютером, в рамках системы, управляющей суперкомпьютером, будут функции планирования процессов, будет распределенная файловая система.

Управление процессами

Определение. *Процесс* – совокупность машинных команд и данных, обрабатываемых в рамках вычислительной системе и обладающих правами на использование или владение некоторым набором ресурсов вычислительной системы.

Выделенные процессу ресурсы могут либо эксклюзивно принадлежать данному процессу, либо эти ресурсы могут использоваться/принадлежать единовременно двум и более процессам. Например, разделяемым ресурсом является виртуальный принтер.

Выделение ресурсов процессу может происходить по двум моделям:

- декларативно – до начала обработки процесса
- динамически – во время выполнения процесса, "по запросу".

Мы привыкли работать с системами, в которых выделения ресурсов происходит по запросу.

Основной функцией ОС является поддержание жизненного цикла процессов.

Определение. *Жизненный цикл процессов* – это связь состояний, в которых может находиться обрабатываемый в системе процесс. Эту связь можно представить в виде ориентированного графа. В узлах этого графа находятся состояния, а ориентированные ребра связывают возможности перехода из одного состояния в другое.

ФАКУЛЬТЕТ ВЫЧИСЛИТЕЛЬНОЙ МАТЕМАТИКИ И КИБЕРНЕТИКИ МГУ ИМЕНИ М.В. ЛОМОНОСОВА

Типовыми этапами обработки процесса в операционной системе могут быть следующие состояния:

- Формирование процесса. Состояние, в котором процесс еще не начинал выполнение, но идет его формирование: выделение ресурсов.
- Выполнение процесса
- Ожидание. Процесс уже сформирован. Два варианта:
  - процесс готов к исполнению и ожидает, когда он перейдет в состояние исполнения;
  - ожидание завершения обмена (подкачка системной информации для процесса), который необходим для функционирования процесса.
- Завершение процесса. В это время ОС занимается освобождением ресурсов, которые были выделены процессу (оперативная память, работа с устройствами)

Имея состояния и правила перехода из одного состояния в другое, можно формализовать различные типы операционных систем.

Модельная ОС

Пусть в ОС есть две системных структуры данных, в которых ОС аккумулирует специализированные данные, связанные с обработкой процессов.

1. *Буфер ввода процессов (БВП)* – некоторое системное пространство, в котором аккумулируются процессы с момента их создания до начала обработки.

   Для привычных нам систем буфер ввода процессов состоит из пространства только для одного процесса. Только один процесс находится в этом буфере, и как только он будет сформирован, сразу же начнет обрабатываться, попадет в пул обрабатываемых. В любой такой системе можно запускать в обработку произвольное количество процессов. Такое допустимо для интерактивных систем.

   Для счетных систем этот буфер аккумулирует заказы на обработку. В рамках буфера может быть своя система планирования и т.д.

2. *Буфер обрабатываемых процессов (БОП)* – буфер для размещения процессов, находящихся в системе в мультипрограммной обработке.

Смоделируем различные ОС, используя граф состояний.

ФАКУЛЬТЕТ
ВЫЧИСЛИТЕЛЬНОЙ МАТЕМАТИКИ И
КИБЕРНЕТИКИ МГУ ИМЕНИ М.В.
ЛОМОНОСОВА

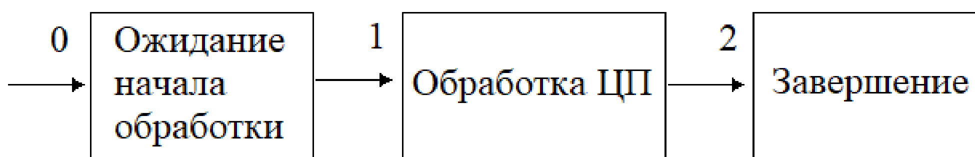

Рис. 11.1. Граф состояний пакетной однопроцессной системы

1. *Пакетная однопроцессная система.*

- 0. Процесс попадает в буфер ввода процессов, где аккумулируются процессы, которые ожидают обработки.
- 1. Система планирования из состояния ожидания начала обработки может перевести какой-то из процессов в состояние выполнения. В этом состоянии процесс будет находиться до завершения.
- 2. Завершение выполнения процесса, освобождение системных ресурсов.

2. *Пакетная мультипроцессная система*

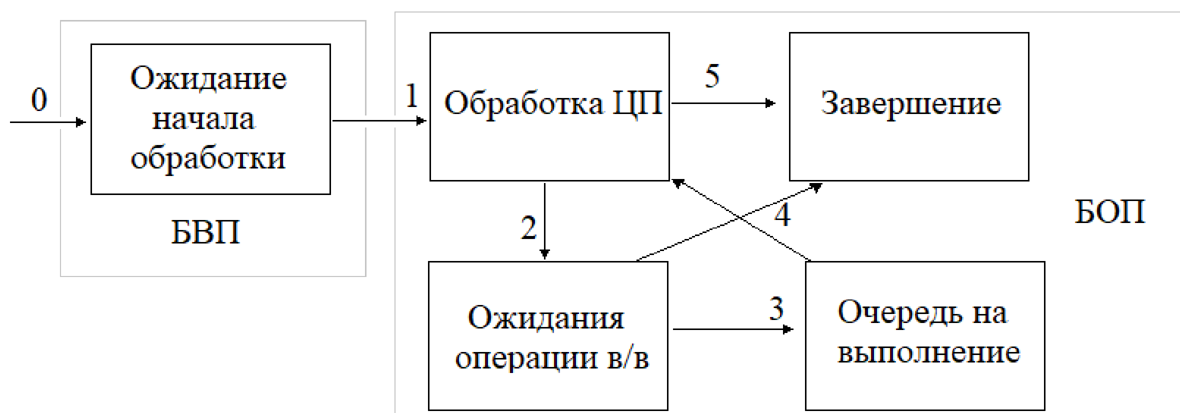

Рис. 11.2. Граф состояний пакетной мультипроцессной системы

- 0. Поступление процесса в очередь на начало обработки ЦП (процесс попадает в БВП)
- 1. Из состояния ожидания в буфере ввода процессов какой-то из процессов переходит в состояние обработки ЦП.
- 2. Из состояния обработки ЦП процесс может уйти либо на ожидание ввода/вывода (прерывание), либо на завершение (5).
- 3. Из состояния ожидания в/в процесс может перейти в очередь готовых к исполнению, когда успешно завершится обмен.

- 4. Из состояния ожидания операции в/в процесс может перейти на завершение, если возникла ошибка в обработке заказа на в/в. Из очереди на выполнение процесс может попасть в состояние обработки ЦП.
- 5. Затем из состояния обработки ЦП процесс уходит на завершение.

3. *Модель ОС с разделением времени*

   В системе с разделением времени определяется квант времени, который выделяется каждому процессу на период обработки.

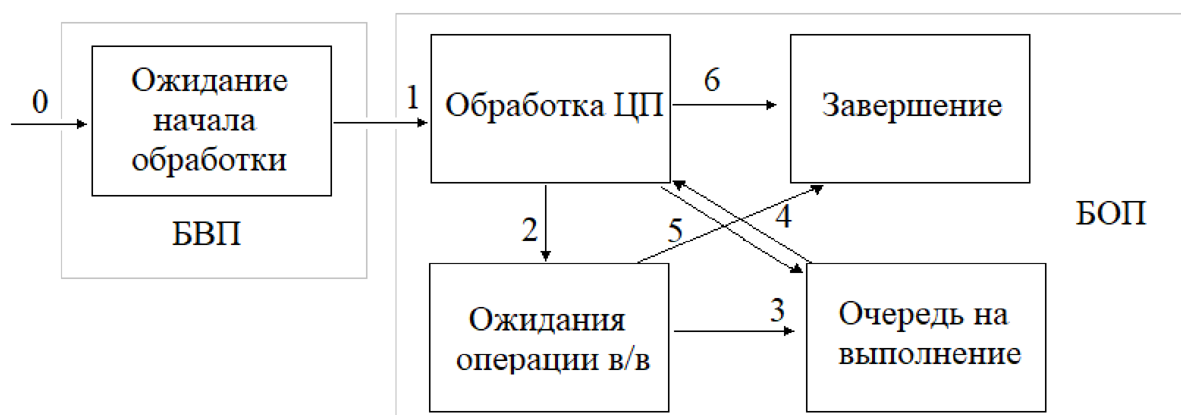

Рис. 11.3. Граф состояний ОС с разделением времени

   Все как и в пакетной системе, только добавляется перемещение процесса из состояния обработки ЦП в состояние очереди готовых на выполнение. Это перемещение происходит в случае исчерпания кванта времени.

4. *Модель ОС с разделением времени (модификация)*

   Пусть у нас есть мультипроцессная система, которая является системой с разделением времени. Добавим возможность откачки процесса во внешнюю память – это и есть процесс свопинга. Добавляется перемещение процесса в область свопинга в состояние очереди готовых на выполнение процессов (см. рис. 11.4). Эта операция добавляется именно в эту очередь, чтобы не зависали операции воода/вывода.

Типы процессов

До настоящего момента под понятием "процесс"понималась совокупность команд и данных, обрабатываемых в системе и обладающих правами на некоторые ресурсы, которые обладали эксклюзивными правами на владение своей оперативной памяти.

ФАКУЛЬТЕТ ВЫЧИСЛИТЕЛЬНОЙ МАТЕМАТИКИ И КИБЕРНЕТИКИ МГУ ИМЕНИ М.В. ЛОМОНОСОВА

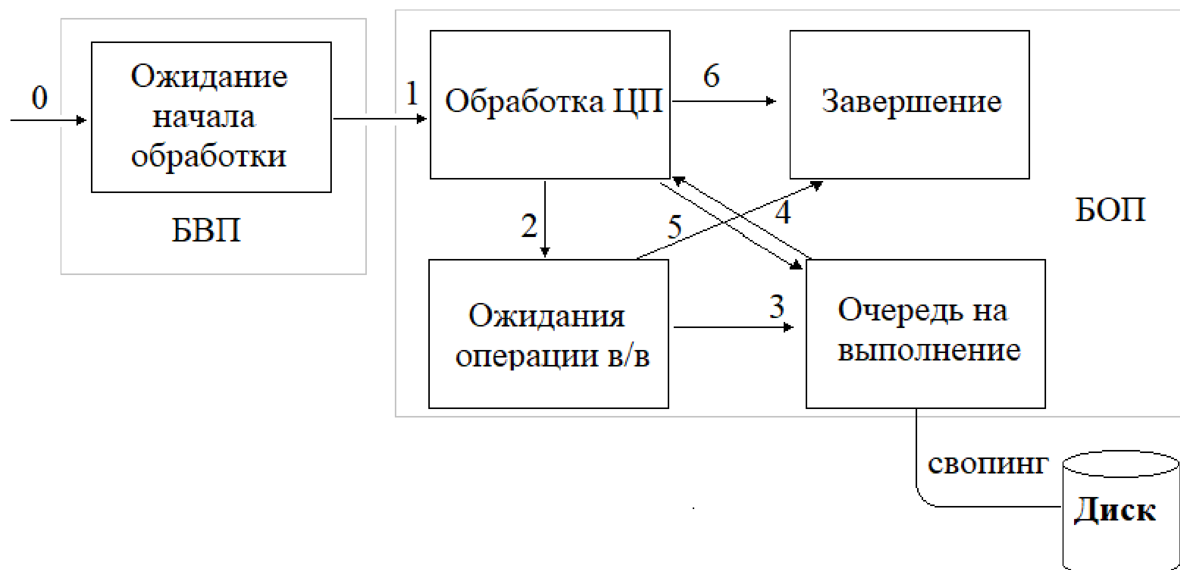

Рис. 11.4. Граф состояний модифицированной ОС с разделением времени

Мы говорили о тех процессах, которые работали в защищенных областях оперативной памяти.

Основным требованиям для мультипрограммирования/мультипроцессирования была защита памяти.

Определение.Полновесные процессы – процессы, выполняющиеся внутри защищенных участков оперативной памяти.

До этого момента предполагалось, что мы говорили о полновесных процессах. Существует альтернатива полновесным процессам: легковесные процессы. Определение. *Легковесные процессы (нити)* работают в мультипрограммном режиме одновременно с активировавшим их полновесным процессом и используют его виртуальное адресное пространство.

Нити могут работать в общей области ОП. Используемая внутри легковесных процессов память не защищена от доступа или влияния процессов, которые работают в этой области.

⇒ В полновесном процессе могут быть реализованы легковесные процессы на ресурсе выделенной для полновесного процесса оперативной памяти. Эти нити работают в точно таком же мульпроцессном режиме, что и полновесный процесс. Нити, работающие в пространстве защищенной памяти полновесного процесса, не защищены друг от друга.

Мы привыкли работать с полновесными процессами, внутри которых существует одна нить.

ФАКУЛЬТЕТ ВЫЧИСЛИТЕЛЬНОЙ МАТЕМАТИКИ И КИБЕРНЕТИКИ МГУ ИМЕНИ М.В. ЛОМОНОСОВА

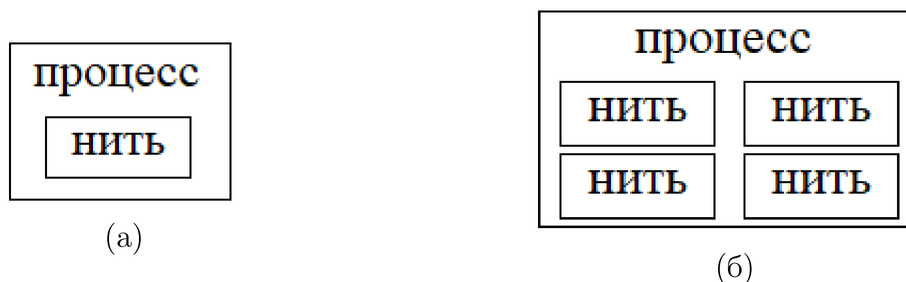

Рис. 11.5. а) Однонитевая организация процесса. б) Многонитевая организация процесса.

Для мультипроцессирования нитей внутри одного полновесного процесса возможна некорректная работа процессов. Но эти нити принадлежат одному пользователю, поэтому ответственность за корректность лежит на пользователе, владеющем полновесным процессом.

Процесс в Unix

Переопределим понятие "процесс".

Определение. *Процесс* – команды и данные, которые обрабатываются, которым выделены ресурсы, выделена оперативная память, и в рамках выделенных ресурсах функционирует хотя бы одна нить.

Понятие "процесс"включает в себя

- исполняемый код
- собственное адресное пространство, которое представляет собой совокупность виртуальных адресов, которые может использовать процесс
- ресурсы системы, которые назначены процессу ОС
- хотя бы одну выполняемую нить

⇒Нити нужны для оптимизации. Переключение между полновесными процессами – очень ресурсоемкая операция. Поэтому при работе в общем адресном пространстве при переключении между нитями нет необходимо перестраивать настройки виртуальной память.

Далее будем говорить о Unix системах.

Процесс в Unix может быть определен двумя способами:

1. это объект, зарегистрированный в таблице процессов ОС
2. это объект, порожденный системным вызовом fork().

Рассмотрим оба случая.

1. Определение. *Процесс в Unix* – объект, зарегистрированный в таблице процессов UNIX.

В ОС имеется таблица предопределенного размера, который определяется параметром настройки ОС. Таблица состоит из записей фиксированного размера, каждая запись имеет позиционный номер(идентификатор). Каждая запись таблицы процессов может соответствовать одному из процессов в системе.

Идентификатором процесса (PID) является номер записи в таблице процессов. Структура записи таблицы процессов предполагает хранение некоторого набора характеристик процесса. В записи таблицы процесса имеется ссылка на данные, которые называются *контекстом процесса.*

Определение. *Контекст процесса* – это аппаратно-системные данные, характеризующие актуальное состояние обрабатываемого процесса.

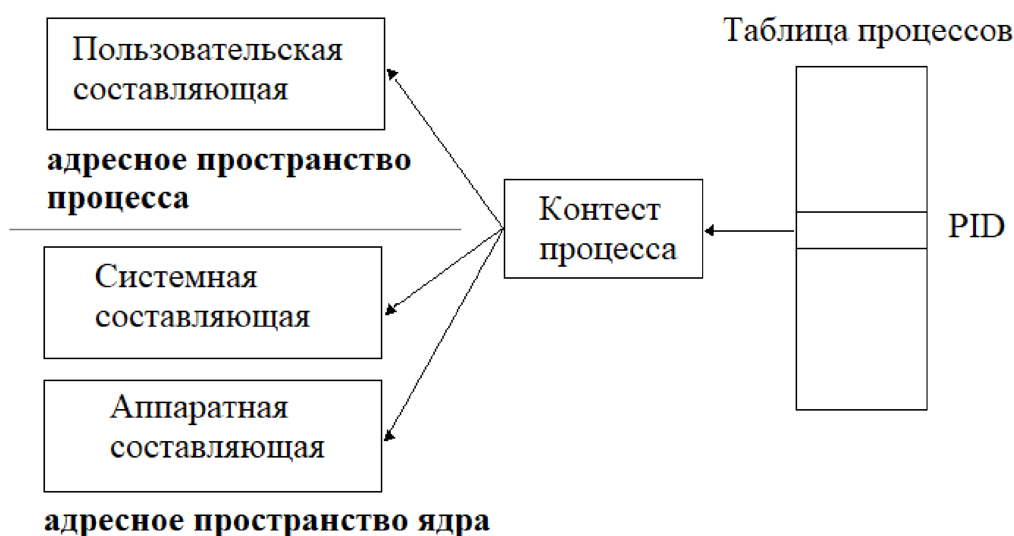

Рис. 11.6. Идентификатор процесса (PID)

Контекст процесса состоит из

1. пользовательской составляющей – это все, что относится к коду исполняемого процесса;

2. аппаратной составляющей – это содержимое регистров общего назначения и специальных регистров;

   Аппаратная составляющая присутствует только когда процесс находится в состоянии выполнения.

3. системной составляющей – это системная структура данных, которая формируется и хранится в ОС, которая описывает характеристики каждого процесса(приоритеты, информация о выделенных процессы ресурсах и т.д.), а также это может быть копия аппаратной составляющей, если процесс находится в состоянии ожидания.

Рассмотрим подробнее каждую из компонент контекста процесса.

- Пользовательская составляющая состоит из
  - сегмента кода, который содержит код процесса и может содержать некоторые константы. Т.е. это неизменяемая часть контекста процесса.
  - сегмента данных – это те данные, которые используются в процессе. В него входят статические данные (переменные, которые существуют в течение всего времени жизни процесса), стек, который используется для организации работы динамической память. Через стек реализуются автоматические переменные и фактические параметры функций. Т.е. эта часть изменяется динамически.

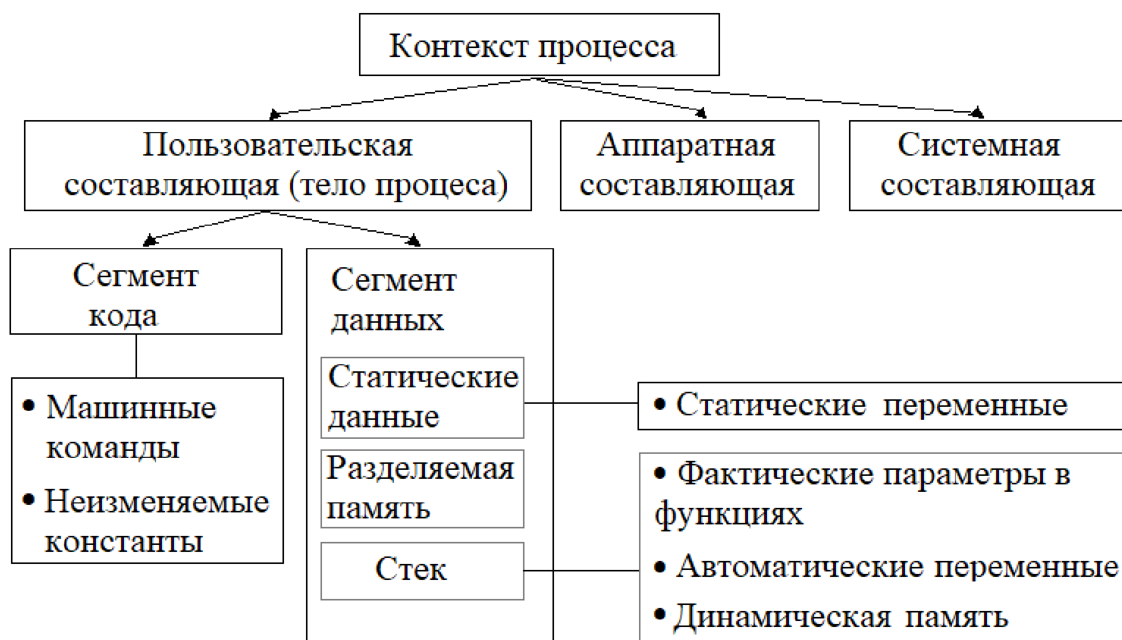

Рис. 11.7. Контекст процесса

ФАКУЛЬТЕТ ВЫЧИСЛИТЕЛЬНОЙ МАТЕМАТИКИ И КИБЕРНЕТИКИ МГУ ИМЕНИ М.В. ЛОМОНОСОВА

Лекция 12. Контекст процесса

Контекст процесса

1. Определение. *Процесс в Unix* – объект, зарегистрированный в таблице процессов UNIX.

Продолжим рассмотрение контекста процесса.

- Пользовательская составляющая состоит из
  - сегмента кода – совокупность данных, которые относятся к процессу и являются неизменными
  - сегмента данных – это те данные, которые изменяются динамически во время выполнения процессе.

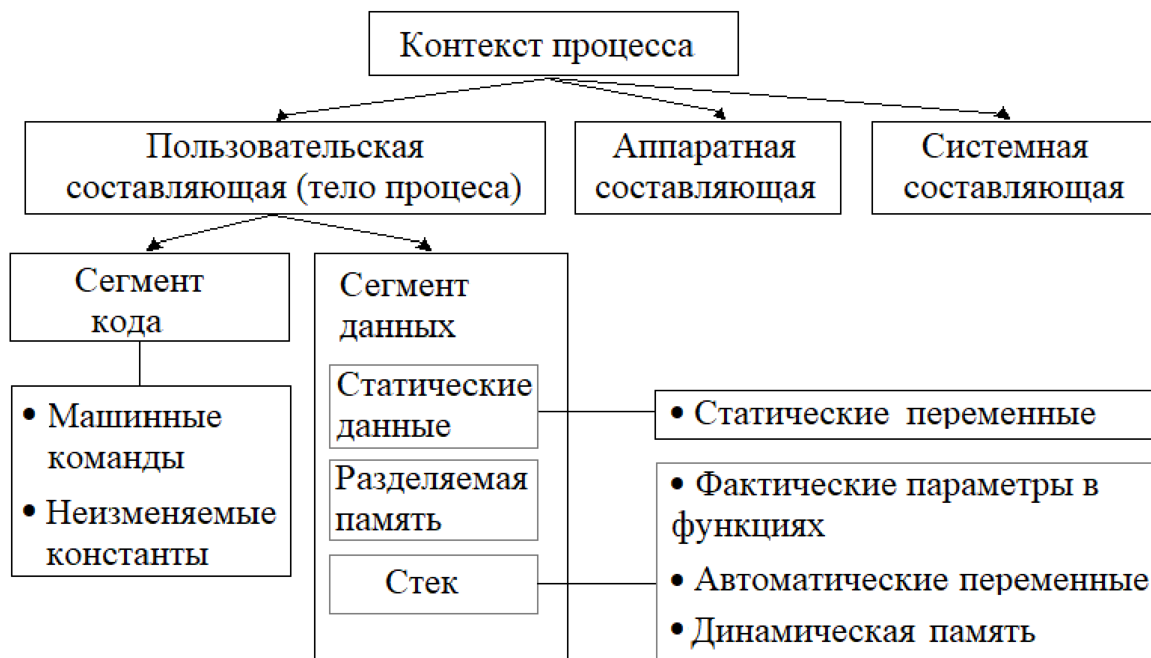

Рис. 12.1. Контекст процесса. Пользовательская составляющая

Рассмотрим ситуацию работы с текстовым редактором на UNIX сервере. Для каждого из пользователей будет запущена своя копия программы текстового редактора. UNIX система позволяет оптимизировать этот процесс.

Для специально помеченных процессов имеется возможность разделения сегмента кода, т.е. идентичные процессы, которые единовременно запущены в системе, в своих контекстах будут иметь один сегмент кода, принадлежащий всем процессам, запущенным с программой текстового редактора. Эта возможность

опирается на свойство неизменности сегмента кода. Это оптимизирует использование ресурсов.

- Аппаратная составляющая контекста – это регистры и аппаратные таблицы ЦП, которые используются в выполняемом процессе. Аппаратная составляющая существует только в тот период времени, когда процесс находится в состоянии выполнения.

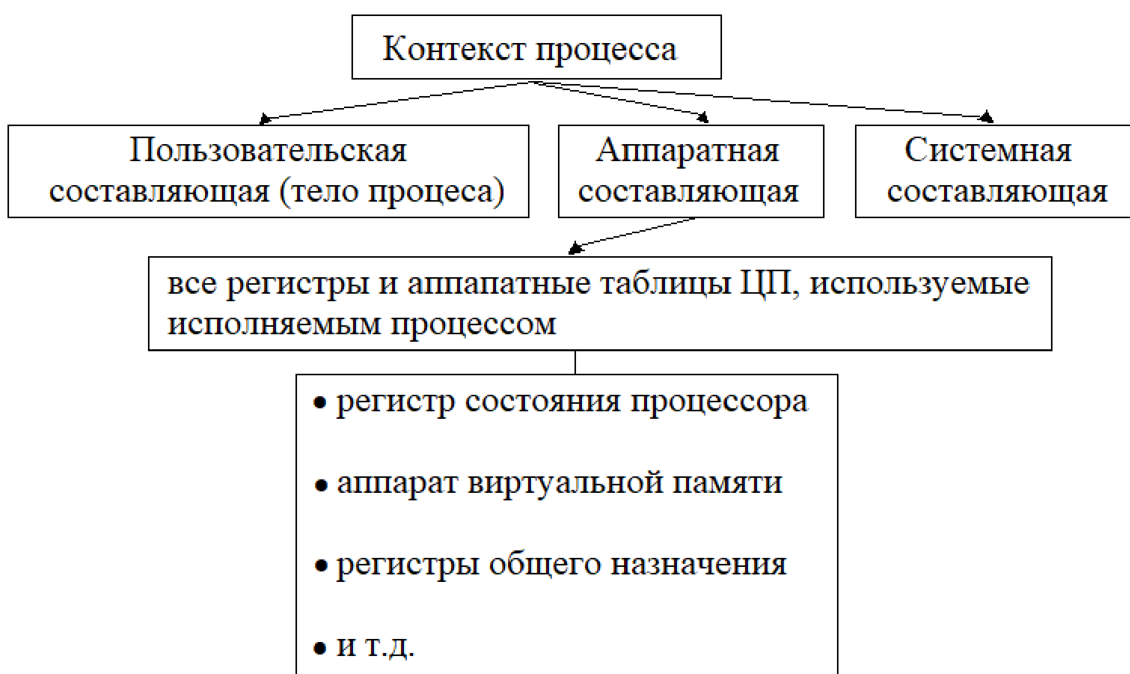

Рис. 12.2. Контекст процесса. Аппаратная составляющая

- Системная составляющая – это все то, что формирует и сохраняет ОС об обрабатываемом процессе. В системной составляющей находится PID родительского процесса, находится приоритет процесса.

  В системе UNIX используется динамическая система приоритетов процесса, которая позволяет относительно справедливо распределять ресурсы системы. Процесс, находящийся в состоянии исполнения, постепенно теряет приоритет. Когда его приоритет становится ниже ожидающего процесса, происходим смена процесса.

  – реальный и эффективный идентификаторы пользователя-владельца. Каждый пользователь многопользовательской системы имеет регистрацию. Каждый файл, который находится в файловой системе связан с каким-то пользователем-владельцем. Процессы, которые формируются в системе зарегистрирован-

ным пользователем, формируются из специальных исполняемых файлов. Когда формируется процесс, есть два пользователя: тот, кто дал команду сформировать процесс, и тот, кому принадлежит этот исполняемый файл.

В результате формирования процесса у него появляются для идентификатора: реальный идентификатор, являющийся идентификатором пользователя, который сформировал процесс, и эффективный идентификатор, являющийся идентификатором того пользователя, которому принадлежит исполняемый файл.

Относительно этих двух идентификаторов формируются. Наличие реального и эффективного идентификаторов позволяет, не предоставляя прав администратора системы, разрешить процессу работать с системными данными.

Рис. 12.3. Контекст процесса. Системная составляющая

ФАКУЛЬТЕТ ВЫЧИСЛИТЕЛЬНОЙ МАТЕМАТИКИ И КИБЕРНЕТИКИ МГУ ИМЕНИ М.В. ЛОМОНОСОВА

2. Определение. *Процесс в Unix* – это объект, порожденный системным вызовом fork().

Определение. *Системный вызов* – это обращение пользователя за выполнением той или иной функцией ОС. Обращение процесса к ядру ОС за выполнением тех или иных действий.

Каждая UNIX-система имеет фиксированный предопределенный набор системных вызовов.

Обращение к библиотечной функии отличается от системного вызова тем, что библиотечная функция подгружается в тело процесса и выполняется на ресурсах процесса, а тело и реализация системного вызова находятся в ядре операционной системы.

Создание нового процесса. Системный вызов fork()

```
#include <sys/types.h>
#include <unistd.h>
pid_t fork(void);
```

Системный вызов fork() дублирует процесс, который обратился к этому системному вызову. Т.е. если выполнялся некоторый процесс со своим PID, то после обращения к системному вызову fork() образуется точно такой же процесс как изначальный. Разница будет только в коде ответа системного вызова fork(). Если код ответа системного вызова fork:

- >0, это родительский процесс обратился к системному вызову, создал сыновий процесс. Т.е. код ответа – это PID сыновнего процесса (мы находимся в процессе-отце)
- =0 это значит, что мы находимся в процессе-сыне, который является копией родительского процесса
- =-1 произошла ошибка - невозможно создать новый процесс, эта ошибка может возникнуть при недостатке места в таблице процессов, при нехватке места в системных областях данных и т.п

Примечание. Полезное свойство UNIX системы. Если системный вызов отказывает выполнение запроса, эта информация передается процессу, который обратился к системному вызову, через код ответа системного вызова, а причина отказа уточняется

специальной переменной errno, которая будет доступна для процесса, если подключить файл с именем errno.h.

После формирования процесса, сыновий процесс создается как новый процесс, т.е. у него появляется свой PID. У сыновьего процесса, в отличие от процесса-родителя, будут отличаться PID-ы родительского процесса.

Когда процесс, который породил другой процесс завершается, родительский PID будет занят другими процессами. В этом случае в системе PID родительского процесса заменяется на PID первого процесса.

$\Rightarrow$

- При удачном завершении вызова fork() возвращается:
  - сыновнему процессу значение 0
  - родительскому процессу PID порожденного процесса
- При неудачном завершении возвращается -1, код ошибки устанавливается в переменной errno
- Заносится новая запись в таблицу процессов
- Новый процесс получает уникальный идентификатор
- Создание контекста для нового (сыновьего) процесса

Рассмотрим составляющие контекста, наследуемые при вызове fork().

- При формировании нового процесса в новом процессе передаются практически все *файлы, открытые в родительском процессе.*

  За счет этого в системе реализованы многие функции.
- В сыновьем процессе передаются *способы обработки сигналов.* UNIX система позволяет обрабатывать асинхронные события, которые называются сигналами (сигналы имеют некоторую аналогию с аппаратом прерывания). Система может посылать любому процессу некоторые уведомления, связанные с тем или иным событием. Аппарат событий и их обработки передается сыновьему процессу.
- Разрешение переустановки эффективного идентификатора пользователя

- Разделяемые ресурсы процесса-отца
- Текущий рабочий каталог и домашний каталоги и т.д.

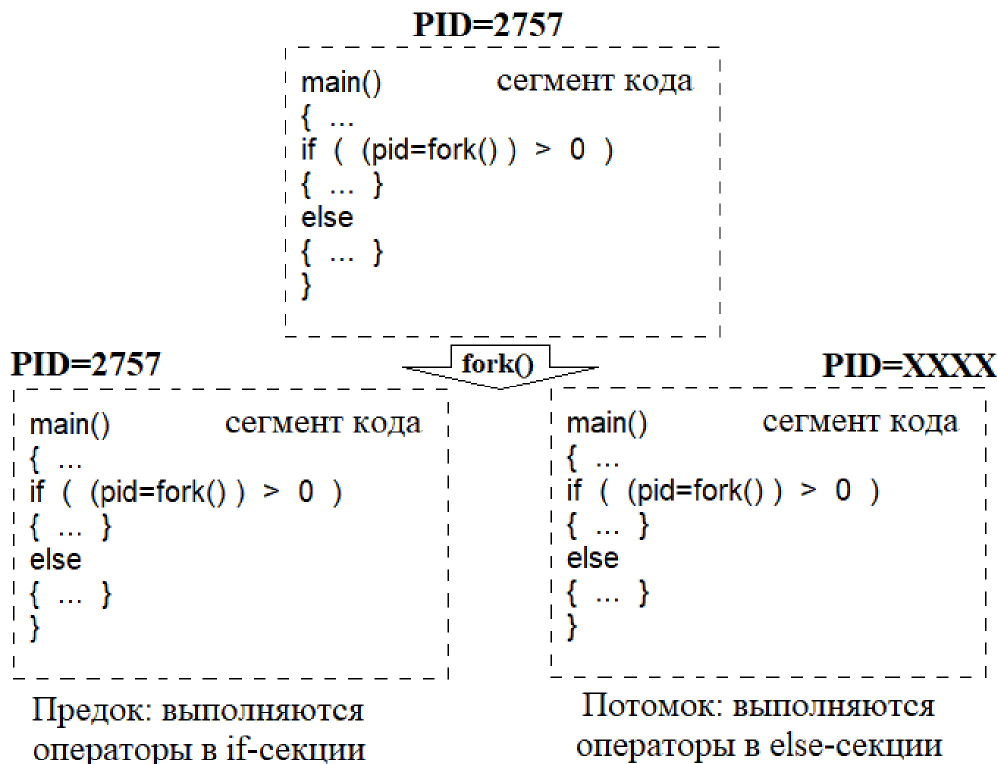

Рис. 12.4. Схема создания нового процесса

Пример

```
int main (int argc, char **argv)
{
  printf("PID=%d; PPID=%d \ n " , getpid(), getppid());
  fork();
  printf("PID=%d; PPID=%d \ n " , getpid(), getppid());
  return 0;
}
```

В первой строке в функции printf находится строка форматирования в двойных кавычках и два параметра, которые нужно выдать на стандартное устройство вывода:

1. getpid() – системный вызов получить PID текущего процесса;
2. getppid() – системный вызов получить пид родителя.

ФАКУЛЬТЕТ ВЫЧИСЛИТЕЛЬНОЙ МАТЕМАТИКИ И КИБЕРНЕТИКИ МГУ ИМЕНИ М.В. ЛОМОНОСОВА

На стандартном устройстве вывода получим
первою строку:

$$PID = 2757; PPID = YYYY,$$

где YYYY – PID родителя.

второй и третьей строками будут в каком-то из порядков левая часть и правая часть, потому что мы не знаем, в каком порядке ОС предоставит процессор отцовскому и/или сыновьему процессу.

$$PID = 2757; PPID = XXXX <--> PID = XXXX; PPID = 2757$$

В первой паре всесто XXXX будут YYYY. Соответственно YYYY может быть либо PID-ом родительского процесса YYYY=PPID, либо YYYY=1, если родительский процесс к моменту обращения завершился.

⇒ Используя fork(), можно формировать параллельные процессы, которые будут выполнять разные веточки: одна веточка по условию fork() > 0, другая по условию fork() = 0.

Но этот способ неэффективен. В теле процесса должны быть заложены и продублированы все возможные ветки.

В UNIX для формирования процессов существует пара fork() и exec().

Семейство системных вызовов exec()

```
# include <unistd.h>
int execl(const char *path, char *arg0, ..., char *argn, 0);
```

Это семейство имеет префиксную часть в виде аббревиатуры exec, а суффиксные части отличает каждый из системных вызовов. Они все работают идентично, разница только в проедставлении информации для них, т.е. в формате фактических параметров, которые им могут быть переданы.

Суть параметров вызова exec():

- path – указатель на имя файла, являющегося исполнительным файлом, содержащего исполняемый код программы. Через системный вызов exec() можно обработать некоторый исполнительный файл, который может быть сформирован как процесс. Может содержать *полное имя файла.*

ФАКУЛЬТЕТ ВЫЧИСЛИТЕЛЬНОЙ МАТЕМАТИКИ И КИБЕРНЕТИКИ МГУ ИМЕНИ М.В. ЛОМОНОСОВА

Определение. *Полное имя файла* – текстовая строка, показывающая путь от корня файловой системы до файла.

- arg0 – дублирование. Это имя файла, содержащего вызываемую на выполнение программу, который будет использован как исполняемый. Может содержать *относительное имя файла* Определение. *Относительное имя файла* – это имя файла относительно какого-то директория файловой системы.

- arg1, ..., arg n – аргументы программы, которые будут переданы в исполняемый модуль, когда он будет запущен как процесс.

В параметрах arg0, arg1 и т.д. разложены структурно все компоненты командной строки.

Структуру и детали интерпретации смотреть по документации.

Рассмотрим схему работы системного вызова exec(). Системный вызов exec() меняет тело исполняемого процесса (того процесса, что обратился к системному вызову) на новое тело, которое построено с использованием исполняемого файла, имя которого передается через системный вызов exec().

Пример

Пусть процесс обращается к execl. Полное имя файла – ls, по пути "/ bin / ls" находится реализация команды ls.

Примечание.В каталоге bin можно найти реализации большинства тех команд, которые используются в UNIX. Это просто исполняемые файлы, т.е. имя команды является именем файла.

В результате обращение к exec тело процесса 2760 будет замещено на тело процесса ls. Происходит смена тела процесса.

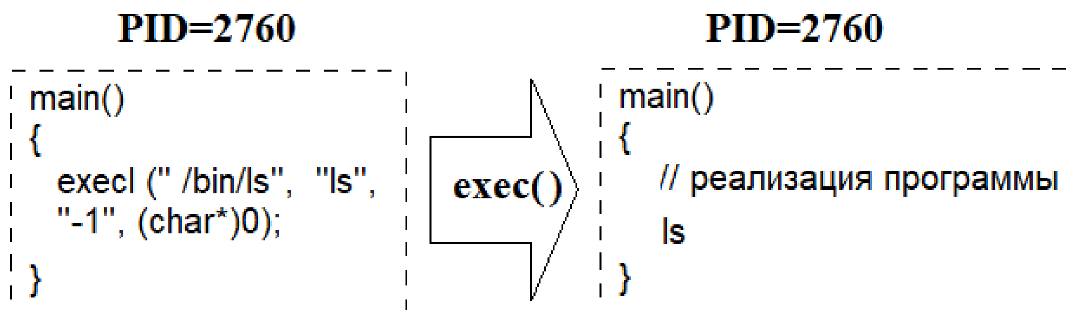

Рис. 12.5. Схема работы системного вызова exec()

Процесс порождения нового процесса. Порождение нового процесса в UNIX происходит следующим образом: fork() создает копию, которая является "посадочным местом"для нового процесса, а затем внутри сыновьего процесса с помощью exec() меняем тело на тело того процесса, который планируется запустить. Таким образом реализуется вызов любой команды, которая связана с формированием процесса.

Если осуществляется обращение к системному вызову exec(), выхода из этого системного вызова нет. Тело меняется, и управление попадает на точку входа нового тела.

Если все же осуществился выход из системного вызова exec(), это означает, что exec() не сработал. Причины ошибки: нет такого файла; файл есть, но не является исполняемым; файл есть, исполняем, но недоступен и т.д.

<u>Важное свойство:</u> при выполнении системного вызова exec(), многие параметры контекста сохраняются. Особенно важно сохранение открытых файлов и обработки сигналов.

В результате exec() сохраняются:

- Идентификатор процесса;
- Идентификатор родительского процесса;
- Таблица дескрипторов файлов;
- Приоритет и большинство атрибутов.

Изменяются:

- Режим обработки сигналов;
- Эффективные идентификаторы владельца;
- Файловые дескрипторы (закрытие некоторых файлов).

Использование схемы fork-exec

Рассмотрим типовой пример связки fork-exec (см. рис. 12.6).

Хотим запустить команду UNIX ls. Есть интерпретатор команд 2757. Через команду fork() создали сыновний процесс, для которого PID=XXX. Сыновний процесс обращается к exec(), меняем тело.

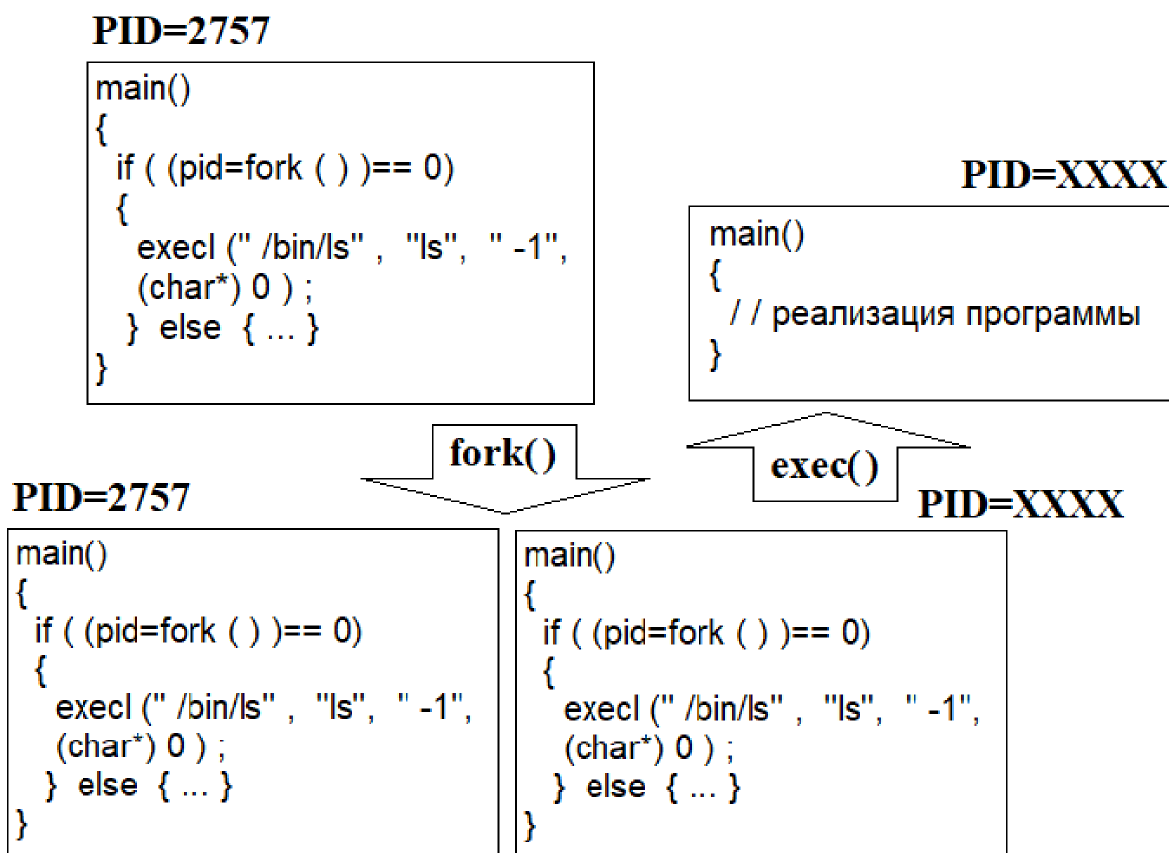

Рис. 12.6. Пример использования схемы fork-exec

Завершение процесса _exit()

Завершение процесса происходит в следующих случаях:

- если мы обращаемся к системному вызову _*exit*();
- если мы выходим на закрывающую операторную скобку объемлющего блока в программе, в этом случае система компиляции подставляет туда _*exit*();
- выполнение оператора **return** , входящего в состав функции **main** (в объемлющем блоке программы), в этом случае система компиляции подставляет туда _*exit*();
- если процесс получает сигнал от другого процесса или от ОС.

```
# include <unistd.h>
void_exit ( int status);
```


ФАКУЛЬТЕТ ВЫЧИСЛИТЕЛЬНОЙ МАТЕМАТИКИ И КИБЕРНЕТИКИ МГУ ИМЕНИ М.В. ЛОМОНОСОВА

Через системный вызов $_exit()$ можно передать программный код завершения процесса. Код завершения, который передается для тех, кто его хочет получить. Это параметр системного вызоыв $_exit()$.

status =0 при удачном завершении процесса

$\neq 0$ при неудаче

При завершении процесса

- происходит возврат занятых ресурсов, т.е. закрываются все открытые дескрипторы файлов;
- будет изменен PID родителя во всех сыновних процесса, все потомки завершающегося процесса получат PPID=1;
- и другое.

Получение информации о завершении своего потомка wait

```
# include <sys/types.h>
# include <sys/wait.h>
pid_t wait(int *status);
```

В подавляющем большинстве случаев получается информация о завершении потомка, но может быть получена информация о возникновении события в одном из потомков.

Системный вызов *wait* работает нетривиально. Если к моменту обращения к системному вызову *wait* какая-то часть потомков уже завершилась, в ответ будет получена информация об одном из завершенных потомков.

В *код ответа* входит PID процесса, который завершился, а в параметрах системного вызова будет указатель на переменную, которая будет содержать код завершения.

Код завершения (status) состоит из двух полей:

- программный код завершения – это код завершения процесса-потомка, который мы устанавливаем в _exit, т.е. можно передать родителю причину завершения
- системный код завершения процесса – это индикатор причины завершения процесса-потомка. В UNIX это получение сигнала. Кот содержит номер сигнала, который вызвал завершение процесса.

При обращении к системному вызову *wait* в случае, если есть завершенные сыновние процессы, то на каждый завершенный процесс при очередном обращении к системному вызову *wait* будет получена информация.

Когда будет получена вся информация о завершенных сыновних процессах, либо если ни один сыновний процесс еще не завершился, процесс обращения к системному вызову *wait* будет приостановлен. Этот процесс будет ожидать завершения или возникновения событий в одном из сыновних процессах. Если же сыновних процессов нет, будет получен код ответа -1.

⇒ wait-процесс ожидает события в одном из сыновних процессов.

Пример.

```
# include <stdio.h>
int main(int argc, char **argv)
{
  int i;
  for ( i=1; i<argc; i++ ) {
    int status;
    if ( fork () > 0 ) {
      wait( &status );
      printf( " process-father\n " );
      continue;
    }
    execlp ( argv[i], argv[i], 0);
    exit ();
  }
}
```

Предположим, был запущен процесс командной строкой **file prog1 prog2 prog3**, где file, prog1, prog2, prog3 – это исполняемые файлы. При запуске основная программа будет сформирована из исполняемого файла с именем file.

Затем запускается цикл по всем параметрам функции main. Для каждого из параметров формируем процесс. После этого ожидаем завершение процесса.

В каждом сыновнем процессе формируем процесс с телом, имя которого передано через параметры prog1, prog2, prog3. Каждый из этих процессов может вывести какой-то текст на стандартное устройство вывода.

Жизненный цикл процессов в UNIX

Жизненный цикл процесса может быть изображен следующим образом:

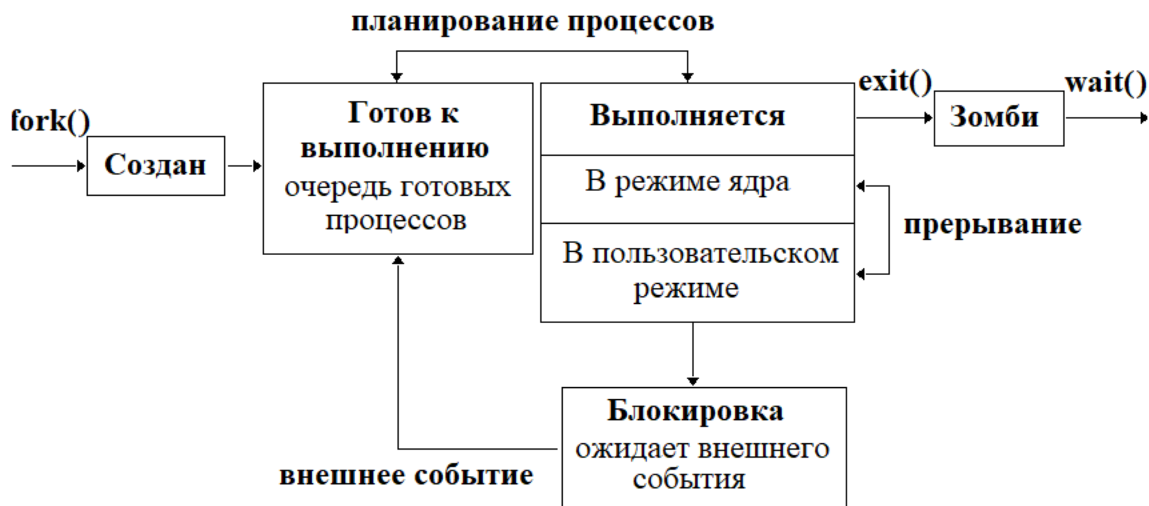

Рис. 12.7. Схема жизненного цикла процессов

1. Процесс создан, еще не начал обрабатываться
2. Когда процесс будет сформирован и все необходимые ресурсы будут выделены, процесс попадает в очередь готовых к выполнению процессов.
3. Затем, согласно обработке приоритетов процессов, которые находятся в контексте, процесс может перейти в состояние выполнения.
4. В состоянии выполнения процесс может работать либо в пользовательском режиме (работает тело процесса), либо в режиме ядра (идет обращение к системным вызовам)
5. Из состояния выполнения процесс может перейти в состояние завершения "зомби из процесса начинают отбираться ресурсы, а затем уничтожается.

   Либо из состояния выполнения процесс может перейти в состояние блокировки.
6. Из состояния блокировки процесс может перейти в состояние готовых к исполнению процессов

ФАКУЛЬТЕТ ВЫЧИСЛИТЕЛЬНОЙ МАТЕМАТИКИ И КИБЕРНЕТИКИ МГУ ИМЕНИ М.В. ЛОМОНОСОВА

Лекция 13. Параллельные процессы

Жизненный цикл процесса в Unix

Разберем жизненный цикл процесса.

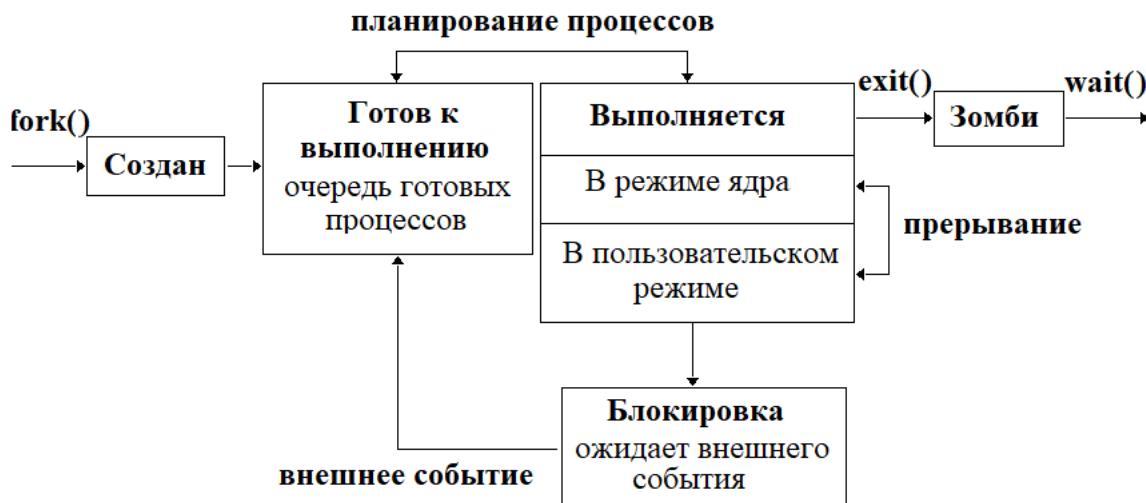

Рис. 13.1. Схема жизненного цикла процессов

1. Процесс **создается** посредством обращения к системному вызову fork().

2. После создания он переходит в **состояние готовности к исполнению**, переходит в очередь процессов, находящихся в обработке в системе UNIX, которые готовы начать исполнение. Они ждут решение системы планирования о предоставлении ресурса процессора.

3. Из состояния "готов к исполнению"на основании средств планирования системы UNIX процесс переходит в **состояние выполнения**.

   В состоянии выполнения процесс может исполняться

   (a) *в пользовательском режиме программы*, когда выполняются машинные коды, принадлежащие телу процесса;

   (b) *в режиме операционной системы (режим ядра)*, когда процесс обращается к системным вызовам, а ОС обрабатывает системные вызовы, которые были инициированы данным процессом.

4. Из состояния выполнения процесс может перейти в **состояние завершения (зомби)**, когда у процесса отбираются ресурсы.

   Процесс завершается в случае

ФАКУЛЬТЕТ ВЫЧИСЛИТЕЛЬНОЙ МАТЕМАТИКИ И КИБЕРНЕТИКИ МГУ ИМЕНИ М.В. ЛОМОНОСОВА

(a) обращения к системному вызову _exit;

(b) обращения к оператору return в объемлющем блоке, попадения на закрывающую операторную скобку объемлющего блока;

(c) приход в процесс некоторого сигнала, уведомления от ОС о том, что в процессе произошло какое-то событие

5. Из состояния выполнения процесс может перейти в состояние **блокировки**, когда процесс не может выполняться, ожидает какого-то внешнего события (пример:ожидание завершения обмена);

6. Из состояния ожидания можно перейти в **состояние готовности к исполнению**.

Начальная загрузка

Определение. *Начальная загрузка* – загрузка ядра системы в оперативную память, запуск ядра.

В результате включения питания компьютера управление передается на предопределенный физический адрес оперативного запоминающего устройства, начиная с которого находится *постоянное запоминающее устройство (ПЗУ)*. В ПЗУ находится *аппаратный загрузчик*. Эта программа может быть настраиваемой. Аппаратный загрузчик информируется о системных устройствах компьютера (внешние запоминающие устройства). Он предполагает, что на этих устройствах может находится ОС. Для аппаратного загрузчика каким-то образом вводится приоритетный перечень системных устройств. Наиболее приоритетным делается резервное устройство, так как такие устройства легче поменять в случае выхода из строй ОС.

Примечание. Обычно системными устройствами являются блок-ориентированные устройства, обмен с которыми происходит порциями фиксированного размена, которые называются блоками.

При запуске аппаратного загрузчика он выбирает наиболее приоритетное готовое к работе системное устройство и читает некоторый предопределенный (нулевой) блок этого системного устройства. Предполагается, что в данном блоке находится *программный загрузчик (загрузчик операционной системы)*. Аппаратный загрузчик считывает программный загрузчик и передает управление на *предопределенную точку входа программного загрузчика*. После этого загрузчик операционной системы начинает работать.

Загрузчик операционной системы UNIX "знает"структуру системного диска. А также знает, что в корне файловой системы есть *исполняемый файл*, в котором находится *ядро ОС*. Обычно этот исполняемый файл созвучен с названием ОС. Программный загрузчик загружает исполняемый файл в физическую память и передает управление на точку входа.

Инициализация системы

После входа в ядро ОС, ядро приводит аппаратуру и информационное программное обеспечение в некоторую каноническую форму.

- Ядро инициирует все возможные аппаратные устройства, которые нужно инициировать: аппарат виртуальной памяти, часы и т.д.
- Ядро инициализирует системные структуры данных. Формируются системные таблицы, которые используются ОС, на основе количественных характеристик из параметров настройки ОС. Например, таблица процессов.
- Формирование процесса с номером ноль заключается в записи некоторой информации в нулевую запись таблицы процессов.

  Нулевой процесс

  – не имеет корректной информации;
  – не имеет кодового сегмента;
  – существует в течении всего времени работы системы, так как считается, что нулевой процесс соответствует ядру ОС.

    Можно сказать, что нулевой процесс создан нестандартно. Он создан не посредством обращения к системному вызову fork().

  Аналогичным образом создается первый процесс.

  ОС не обращается системному вызову fork(), просто заполняет первой записи таблицы страниц.

  Первый процесс

  – корректен с точки зрения структуры таблицы процессов;
  – есть контекст;
  – есть сегмент кода;

– есть выделенная память, в которую ОС загружает реализацию системного вызова exec. После этого идет обращение по стандартной цепочке на замену тела этого процесса на процесс init;

– существует в течении всего времени работы системы.

Процесс *init* (первый процесс) смотрит системные данных, параметры настройки и запускает работу ОС либо в однопользовательском(однотерминальном) режиме или в многопользовательском(многотерминальном) режиме.

Если первый процесс запускается в многотерминальном режиме, то *init* обращается к некоторому системному текстовому файлу, в котором описаны все возможные терминальные устройства. Для всех зарегистрированных таким образом терминальных устройств, готовых к исполнению, будет запущен процесс *getty* по стандартной схеме fork-exec.

На каждое устройство вывода getty запросит ввод имени.

Схема дальнейшей работы системы

Далее getty начинает последовательно видоизменяться.

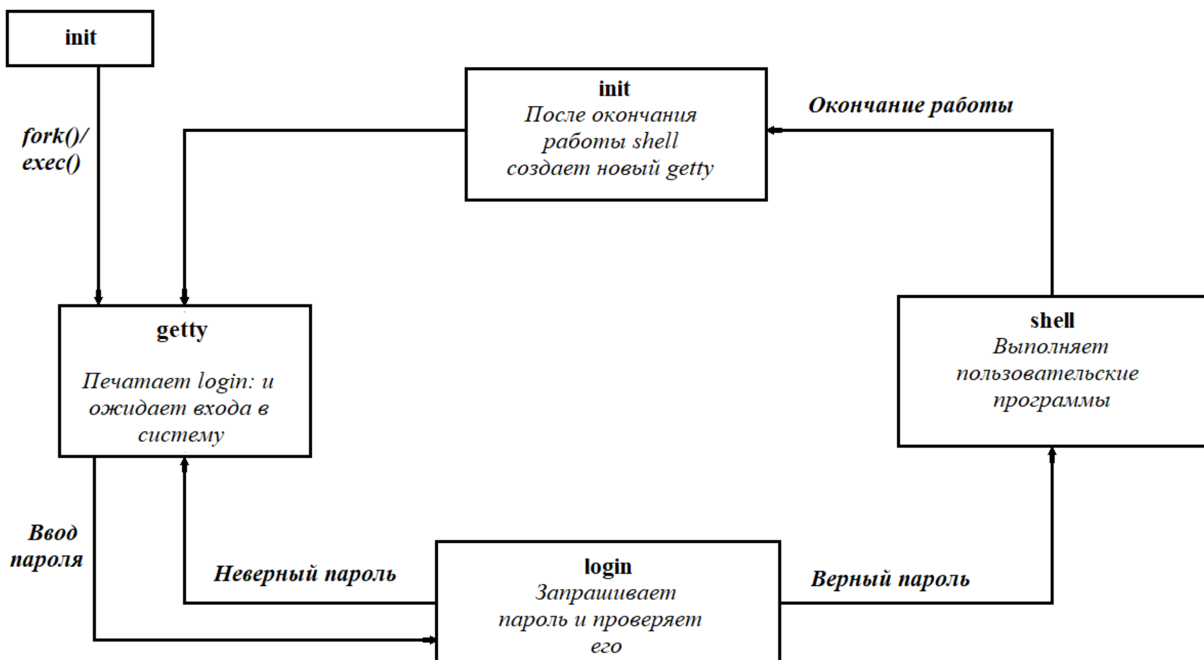

Рис. 13.2. Схема дальнейшей работы системы

ФАКУЛЬТЕТ ВЫЧИСЛИТЕЛЬНОЙ МАТЕМАТИКИ И КИБЕРНЕТИКИ МГУ ИМЕНИ М.В. ЛОМОНОСОВА

После ввода имени загружается программа контроля пароля. По имени и по паролю находится информация о пользователе в файле регистрации пользователя /etc/passwd. В этом файле для каждого пользователя имеется текстовая строка, которая последовательно содержит параметры настройки работы этого пользователя. В этой строке находятся:

1. пароль в некотором зашифрованном виде;

2. домашний каталог – каталог файловой системы, который становится текущим при успешном входе пользователя в свой аккаунт. В этом каталоге находится полное имя интерпретатора команд, который должен быть запущен при входе пользователя в свой аккаунт.

При входе пользователя в свой аккаунт процесс login заменяет свое тело на тело этого исполняемого файла при помощи системного вызова exec. После этого происходит запуск программы. Это стандартный интерпретатор команд.

Для интерпретатора команд вся информация, которая в него поступает представляется как файл. Когда интерпретатор команд при очередном чтении информации со стандартного устройства ввода прочтет код конца файла, процесс закончит свою работу.

Init видит, что закончился процесс, связанный с getty, если устройство остается готовым к работе, он снова запускает getty.

Примечание. В строке регистрации есть еще один файл, в котором можно указать полное имя файла, который будет выполняться при выходе из shell.

Примечание. Можно во многие места подставлять свои решения. Например, можно заменить интерпретатор команд своей программой, также можно заменить файл, который будет запущен перед завершением.

Параллельные процессы

Были рассмотрены многопрограммные/многопроцессные/мультипроцессные системы. Это такие системы, в которых единовременно в обработке могут находиться два и более процессов.

Определение. *Параллельные процессы* – процессы, выполнение (обработка) которых хотя бы частично перекрывается по времени.

Такие процессы могут быть двух типов:

- *Взаимодействующие процессы* могут использовать часть своих ресурсов совместно, а работа одного процесса может оказывать влияние на работу другого процесса.

  Пример взаимодействующих процессов: процессы, имеющие общую страница оперативной памяти. Т.е. единовременно все, что записывает в эту страницу один процесс, становится доступным для других процессов. Каждый из процессов, которым доступна эта страница, может оттуда что-то считать или туда записать.

- *Независимые процессы* – такие процессы, которые при своей обработке или выполнении используют независимые/непересекающиеся множества ресурсов (в основном их программировали в курсе).

Разделение ресурсов

Ресурс, который единовременно доступен двум и более процессам, может называться разделяемым ресурсом.

Определение. *Разделение ресурса* – совместное использование несколькими процессами ресурса ВС.

Возможность бесконтрольного взаимодействия с разделяемым ресурсом не есть хорошо. Поэтому важным элементом является *организация контроля за доступом к разделяемым ресурсам.*

Существуют такие разделяемые ресурсы, что стратегия их использования позволяет в каждый момент времени быть доступным только одному процессы.

Определение. *Критические ресурсы* – разделяемые ресурсы, которые должны быть доступны в текущий момент времени только одному процессу.

Определение. *Критическая секция/критический инетрвал* – Фрагмент кода, в котором реализуется работа с критическим ресурсом.

Требование мультипрограммирования

Одним из ключевых требований корректности организации мультипрограммирования является требование, чтобы результат работы выполнения процесса/процессов не зависел от порядка переключения выполнения между процессами.

Ситуация, когда результат становится зависимым от порядка переключения выполнения между процессами называется *гонками* между процессами.

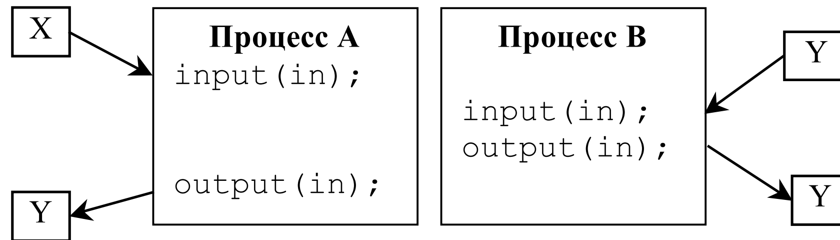

Рис. 13.3. Конкуренция процессов за ресурс. Гонка.

Пусть существует разделяемая переменная in. Запускаются два процесса, в которых реализуется следующая программка.

```
void echo ()
{
  char in;
  input ( in ) ;
  output ( in ) ;
}
```

in – разделяемая переменная, которая доступна единовременно и бесконтрольно каждому из процессов: процессу A и процессу B. Получаемый результат будет зависеть от порядка, по которому система планирования будет передавать ресурс центрального процессора.

Рассмотрим разные варианты развития событий:

- Процесс A может ввести переменную X и сразу же ее вывести, если процесс B не будет запущен.
- Процесс A вводит X, затем запускается процесс B, который вводит и выводит значение Y, после этого управление возвращается к процессу A, который выведет уже значение Y (см. рис 13.3)

⇒ Нельзя допускать гонки.

Одним из средств борьбы с проблемами, связанными с разделением ресурсов, является использование *взаимоного исключения.*

Определение. *Взаимное исключение* – способ работы с разделяемым ресурсом, при котором в тот момент, когда один из процессов работает с разделяемым ресурсом, все остальные процессы не могут иметь к нему доступ.

Проблемы, связанные с этим подходом:

ФАКУЛЬТЕТ ВЫЧИСЛИТЕЛЬНОЙ МАТЕМАТИКИ И КИБЕРНЕТИКИ МГУ ИМЕНИ М.В. ЛОМОНОСОВА

- *Тупик (deadlock)* – ситуация, при которой несколько процессов, работающих в режиме взаимного исключения с разделяемыми ресурсами, могут блокировать друг друга из-за своей активности.
- *Блокирование (дискриминация)* – ситуация, когда активность одного/одних процессов не позволяет работать другим процессам. Т.е доступ одного из процессов к разделяемому ресурсу не обеспечивается из-за активности других, более приоритетных процессов.

Тупики (deadlocks)

Пусть у нас есть два критических ресурса и есть два процесса, которые могут работать с этими ресурсами (см. рис. 13.4).

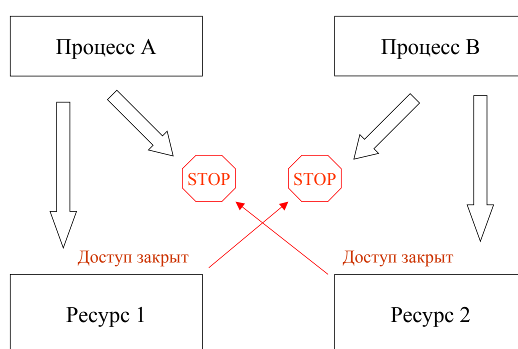

Рис. 13.4. Тупики

Пусть в некоторый начальный момент времени *процесс A* вошел в критическую секцию *ресурса 1* и начал с ним работать. Тем временем *процесс B* вошел в критическую секцию *ресурса 2* и тоже начал с ним работать. Затем *процесс A* пытается войти в критическую секцию *ресурса 2* и заблокируется, потому что в критической секции *ресурса 2* находится *процесс B*. Далее *процесс B* пытается войти в критическую секцию *ресурса 1*, в которой находится находится заблокированный *процесс A*, и тоже заблокируется. Произойдет блокировка. Эти два процесса могут быть закончены только принудительно.

Способы реализации взаимного исключения

Далее будут обсуждаться наиболее популярные инструменты для организации взаимного исключения.

1. **Семафор Дейкстры**. Семафоры Дейкстры – формальная модель синхронизации. Считается, что есть некоторая целочисленная переменная S типа семафор.

Примечание. Тип данных - набор значений и набор операций.

Над переменной S типа семафор вводятся две операции, которые являются атамарными:

(a) *down (S) или P(S) – Proberen (проверить)*. Проверяется значение S.

 i. Если S>0, то S уменьшается на единицу.

 ii. Если S=0, то процесс, который выполняет операцию down, блокируется. Операция down (опустить семафор) не доработала.

(b) *up (S) или V(S) – Verhogen (увеличить)*. Операция поднятия семафора: происходит увеличение значения S на единицу.

Если с системе были процессы, заблокированные на семаформе S, то один из процессов разблокируется, и для него доработает операция down (из содержимого S вычитается единица). → в этом случае значение S не изменится.

Бытовой пример. Пусть есть супермаркет со следующими правилами использования:

- в начале рабочего дня у входа в супермаркет находится N тележек;
- вход в торговую зону супермаркета возможен только при наличии тележки;
- количество тележек у входа фиксирует счетчик.

Первый покупатель взял тележку, счетчик показывает N-1(операция down). Второй покупатель забрал вторую тележку. После N-го покупателя счетчик показывает 0. N+1-й покупатель ждет у входа свободную тележку(заблокировался), N+2-й покупатель тоже блокируется и т.д.

Покупатели выходят из магазина и оставляют тележки, счетчик увеличивается на 1 (операция up).

Если есть "заблокированные"покупатели, наиболее шустрый забирает освободившуюся тележку, счетчик снова уменьшается на 1(down), и т.д.

⇒ начальное значение семафора определяется количеством процессов, которые одновременно могут работать с разделяемыми ресурсами, находится в секции.

Если начальное значение N=1, получается *двоичный семафор*, который используется для взаимного исключения.

Пример. Классическая схема использования двоичного семафора.

процесс 1
```
int semaphore;
…
down(semaphore);
/*критическая секция
процесса 1 */
...
up(semaphore);
…
```

процесс 2
```
int semaphore;
…
down(semaphore);
/*критическая секция
процесса 2 */
...
up(semaphore);
…
```

В каждом из параллельных процессов критическая секция объемлется кодом процесса между операцией опустить семафор и поднять семафор.

Если процесс 1 находится в критической секции процесса 1, то процесс 2 в свою критическую секцию не войдет. Справедливо обратное.

Считается, что семафоры Дейксстры в их реализации – низкоуровневые средства, которые позволяют организовывать синхронизацию и взаимное исключения. Это те средства, которые хорошо, если решаются на уровне аппаратуры или на уровне ОС. Это вызвано тем, что внутри семафоров есть требование атамарности.

Семафор Дейкстры – средство, работающее на одном компьютере, под централизованным управлением.

2. **Мониторы Хоара.** Мониторы Хоара являются альтернативой семафоров Дейкстра. Это высокоуровневое средство, которое можно использовать на уровне программных систем, систем программирования.

   Определение. *Монитор Хоара* – совокупность структур данных и функций/процедур, посредством которых может осуществляться доступ к этим данным. Совокупность функций и данных объединяется в программный модуль специального типа, который называется Монитором Хоара.

   Эта реализация на уровне языков высокого уровня.

   Характеристики монитора Хоара:

   - Все данные монитора доступны только посредством обращения к функциям/процедурам монитора.
   - Процесс "вошел"в монитор Хоара в том случае, есть он обратился к одной из функций/процедур монитора.
   - В любой момент времени внутри монитора может находиться не более одного процесса.

Если какой-то процесс хочет обратиться к одной из функций монитора, но монитор занят, этот процесс будет заблокирован и будет ожидать освобождения монитора.

Пример. Один телефон на всех. Если один человек взял телефон, он становится недоступным для других на какое-то время.

3. **Обмен сообщениями**

Это средство send/receive. Один процесс может оправить "куда-то"сообщение, либо процесс может получить "откуда-то"сообщение. За счет того, что можно варьировать характеристики этой конструкции, она широко используется и сейчас.

Синхронизация и передача данных:

- для однопроцессорных систем и систем с общей памятью;
- для распределенных систем (когда каждый процессор имеет доступ только к своей памяти).

Операции send/receive могут быть :

- блокирующие: блокирующий send – ожидание после отправки до момента получения получателем, блокирующий receive – ожидание до момента получения сообщения;
- неблокирующие.

Адресация может быть двух типов:

- прямая/явная : четко указывается получатель;
- косвенная/безымянная : сообщение отправляется для всех, его читают те, кому нужно (аналог почтового ящика, лекции)

Средство send/receive является основой для программирования суперкомпьютерных систем. Это средство позволяет организовывать и взаимодействие(через передачу сообщений), и синхронизацию (через блокирующие/неблокирующие составляющие).

Примитивы send/receive в отличие от семафора Дейкстры и монитора Хоара, могут работать на многопроцессорных и многомашинных системах.

Средство send/receive позволяет взаимодействовать в распределенных средах.

Лекция 14. Классические задачи синхронизации

Классические задачи синхронизации процессов

Будем рассматривать различные модели задач синхронизации для различных приложений (классические задачи синхронизации).

Разберем три задачи:

- Обедающие философы – это задача, связанная с равнозначным доступом процессов к общему ресурсу.
- Задача о читателях и писателях – задача резервирования
- Задача о спящем парикмахере – задача клиент-сервер с ограниченной очередью ожидания.

Обедающие философы

Это модель доступа равнозначных процессов к общему разделяемому ресурсу.

N философов собираются за круглым столом, перед каждым из них стоит блюдо со спагетти, и между каждыми двумя соседями лежит вилка (всего N вилок). Для того, чтобы поесть, необходимо взять две вилки. Каждый из философов некоторое время размышляет, затем берет две вилки (одну в правую руку, другую в левую) и ест спагетти, затем опять размышляет и так далее. Каждый из них ведет себя независимо от других. Таким образом, философы должны совместно использовать имеющиеся у них вилки (ресурсы). Задача состоит в том, чтобы найти алгоритм, который позволит философам организовать доступ к вилкам таким образом, чтобы каждый имел возможность насытиться, и никто не умер с голоду.

Простейшее решение. Пусть за столом сидят N=5 философов. Для каждого философа определим следующий процесс, в котором философы находятся в бесконечном цикле.

Когда философам голодно, они берут сначала левую вилку, потом правую вилку, ест, затем кладут левую и правую вилки обратно на стол.

```
#define N 5 /* число философов*/
void philosopher (int i)      /* i – номер философа от 0 до 4*/
{
while (TRUE)
{
```

```
think();    /*философ думает*/
take_fork(i);    /*берет левую вилку*/
take_fork((i+1) % N ) ; /*берет правую вилку*/
eat () ; /*ест*/
put_fork(i);    /*кладет обратно левую вилку*/
put_fork((i+1) % N);    /* кладет обратно правую вилку */
}
}
```

Проблема заключается в том, что если сразу все философы проголадуются, может случиться deadlock.

Попробуем составить более правильное решение.

Правильное решение с использованием семафоров

- определяем именованную константу N=5;
- Определяем два макроса Left и Right. Вычисление левого и правого по номеру философа ;
- Определение константы размышления;
- Определение константы состояния "голоден";
- Определение константы состояния "ест";
- Определение типа semaphore;
- Определяем массив состояний, состоящих из N элементов. i-й элемент – состяние i-го философа.
- Определяем начальное состояние этого массива. Этот массив инициализирован нулями, т.к. это статическая глобальная переменна.
- Определяется semaphore mutex, который используется для доступа в критическую секцию.
- Определяется массив семафоров s из N элементов. Ассоциируем каждый элемент массива со своим философом.

```
# define N 5     /* Определяем именованную константу */
# define LEFT (i-1) % N     /* номер легого соседа для i-ого философа */
# define RIGHT (i+1) % N     /* номер правого соседа для i-ого философа*/
```

```
# define THINKING 0    /* философ думает */
# define HUNGRY 1    /* философ голоден */
# define EATING 2    /* философ ест */
typedef int  semaphore ;    /* определяем семафор */
int state [ N ] ;    /* массив состояний каждого из философов */
semaphore mutex = 1;    /* семафор для критической секции */
semaphore s [ N ] ;    /* по одному семафору на философа */
```

Далее рассмотрим процесс philisipher.

- *Функция "философ".* У каждого философа циклический процесс: он размышляет, берет вилки, кушает, затем возвращает вилки. i- номер философа.
- *Функция "взять вилки".*
  - Опускаем семафор mutex, т.е. если придет первый философ, процесс войдет в критическую секцию s=0 (философ станет голодным).
  - Запуск функции test (см. ниже), которая проверяет возможность взять вилки. Если для философа выполняются условия теста, его семафор становится s=1 (выход из критической секции) и он берет вилки.
  - Опускается семафор, связанный с этим философом. Если философ получил вилки, семафор становится s=0.
  - Если же философ не получил вилки, то при опускании семафора он блокируется.
- *Функция "положить вилки"*
  - Вход в критическую секцию .
  - Замена статуса философа из состояния "ест"в состояние "размышляет".
  - Проверка для левого и правого соседей ситуации с вилками.
  - Если для какого-то из соседей выполняется условие test, семафор такого соседа поднимается, а его состояние разблокируется.
- *Функция test*
  - Если философ голодный, а левый и правый соседи не едят, то состояние философа переводится из состояния "голодный"в состояние "ест".
  - Семафор для этого философа поднимается.

Если для философа верно это условие, семафор для него станет s=1 (выход из критической секции).

```
void philosopher ( int i)     /* i : номер философа от 0 до N-1 */
{  while ( TRUE )     /* бесконечный цикл */
   {
   think () ;     /* философ думает */
       take_forks(i);     /*философ берет обе вилки или блокируется */
       eat();     /* философ ест */
       put_forks(i);     /* философ кладет обе вилки на стол */
   }
}

void take_forks ( int i ) /* i : номер философа от 0 до N-1 */
{
   down ( & mutex ) ;     /* вход в критическую секцию */
   state [ i ] = HUNGRY;     /*записываем, что i-ый философ голоден */
   test ( i ) ;     /* попытка взять обе вилки */
up ( & mutex ) ;     /* выход из критической секции */
down ( & s [ i ] ) ;     /* блокируемся, если вилок нет */
}

void put_forks (int i )     /* i : номер философа от 0 до N-1 */
{
    down ( & mutex ) ;     /* вход в критическую секцию */
   state[i] = THINKING;     /* философ закончил есть */
   test ( LEFT ) ;     /* проверить может ли левый сосед сейчас есть */
   test ( RIGHT ) ;     /* проверить может ли правый сосед сейчас есть*/
   up ( & mutex ) ;     /* выход из критической секции */
}

void test (int i )     /* i : номер философа от 0 до N-1 */
{
   if ( ( state [ i ] == HUNGRY )
   && ( state [ LEFT ] != EATING )
   && ( state [ RIGHT ] != EATING ) )
   {    state [ i ] = EATING ;
   up ( & s [ i ] ) ;
```

```
  }
}
```

Это модель доступа равнозначных процессов к общему разделяемому ресурсу.

Читатели и писатели

Вспомним систему бронирования. В такой системе есть две функции:

- функция чтения, знакомства с информацией;
- функция писателя – функция внесения информации в систему.

Рассмотрим модельную задачу, в которой читатели являются наиболее приоритетными.

Пока в системе есть "читающие"процессы, процесс "писатель"будет ожидать.

Будем считать, что

- есть переопределенный тип semaphore
- есть разделяемая переменная и связанная с ней критическая секция (счетчик читателей rc, изначально инициализированный нулем);
- есть семафор mutex, который используется для входа в критическую секцию, связанную с доступом к переменной rc;
- есть семафор db, который блокирует/открывает доступ к базе данных, которые можно модифицировать.

```
typedef int semaphore ;   /* некий семафор */
int rc = 0;    /* кол-во процессов читающих или пишущих */
semaphore mutex = 1;   /* контроль за доступом к «rc» (разделяемый ресурс) */
semaphore db = 1;    /* контроль за доступом к базе данных*/
```

- *Процесс "читатель".*
  - Вход в критическую секцию, опускание семафора mutex до 0.
  - Увеличивается счетчик читателей на 1.
  - Если счетчик rc=1, опускается семафор db.

    Т.е. если это первый читатель, блокируется доступ к базе данных на модификацию. Блокировка длится до тех пор, пока в системе есть хотя бы один читатель. Это условие приоритетности.

ФАКУЛЬТЕТ ВЫЧИСЛИТЕЛЬНОЙ МАТЕМАТИКИ И КИБЕРНЕТИКИ МГУ ИМЕНИ М.В. ЛОМОНОСОВА

– Выход из критической секции.
– Чтение данных.
– Вход в критическую секцию mutex.
– Уменьшение счетчика читателей на 1.
– Если ушел последний читатель (rc=0), поднимается семафор db, что разблокирует доступ к базе данных на модификацию.
– Выход из критической секции.

У каждого читателя такой процесс.

```
void reader ( void )
{
  while ( TRUE )      /* бесконечный цикл */
  {
    down ( & mutex ) ;      /* получить эксклюзивный доступ к «rc»*/
    rc ++ ;      /* еще одним читателем больше */
    if ( rc == 1 ) down ( & db );      /* если это первый читатель, нужно
заблокировать эксклюзивный доступ к базе */
    up ( & mutex );      /*освободить ресурс rc */
    read_data_base ();      /* доступ к данным */
    down ( & mutex ) ;      /*получить эксклюзивный доступ к «rc»*/
    rc - - ;      /* теперь одним читателем меньше */
    if ( rc == 0 ) up ( & db ) ;      /*если это был последний читатель,
разблокировать эксклюзивный доступ к базе данных */
    up ( & mutex ) ;      /*освободить разделяемый ресурс rc*/
    use_data_read () ;      /* некритическая секция */
  }
}
```

- *Процесс "писатель".*

  Действия по модификации базы данных объемлем в критическую секции по семафору db.

```
void writer ( void )
{
  while (TRUE )       /* бесконечный цикл */
  {
```

```
    think_up_data () ;          /* некритическая секция */
    down ( & db ) ;         /* получить эксклюзивный доступ к данным*/
    write_data_base () ;         /* записать данные */
    up ( & db ) ;         /* отдать эксклюзивный доступ */
  }
}
```

Спящий парикмахер

Рассмотрим модель клиент-сервер с ограничением на длину очереди ожидания. Являетя самой популярной моделью.

Рассмотрим парикмахерскую, в которой работает один парикмахер, имеется одно кресло для стрижки и несколько кресел в приемной для посетителей, ожидающих своей очереди. Если в парикмахерской нет посетителей, парикмахер может спать в своем рабочем кресле. Появившийся посетитель должен его разбудить, в результате чего парикмахер приступает к работе. Если в процессе стрижки появляются новые посетители, они должны либо подождать своей очереди, либо покинуть парикмахерскую, если в приемной нет свободного кресла для ожидания. Задача состоит в том, чтобы корректно запрограммировать поведение парикмахера и посетителей.

- Определяется именованная константа, в которой назначается функция количества кресел в комнате ожидания. Пусть кресел пять.
- Определяется тип semaphore.
- Определяется тип semaphore barbers. Состояние semaphore barbers = 0 означает, что парикмахер спит или занят.
- Определили тип semaphore customers. semaphore customers показывает количество посетителей, ожидающих в очереди.
- Определили тип semaphore mutex. Семафор будет использоваться для организации критической секции доступа к переменной waiting, равной количеству ожидающих клиентов.

```
define CHAIRS 5
typedef int semaphore ;       /* тип данных «семафор» */
semaphore customers = 0 ;        /* посетители, ожидающие в очереди */
semaphore barbers = 0 ;        /* парикмахеры, ожидающие посетителей */
semaphore mutex = 1 ;        /* контроль за доступом к переменной waiting */
```

```
int waiting = 0 ;
```

- *Процесс "парикмахер"*
  - Опускаем семафор customers. Если посетителей нет, на этом семафоре процесс будет заблокирован.

    Если customers=1, семафор разблокируется и будет осуществлен вход в критическую секцию.
  - В критической секции изменяется количество ожидающих клиентов.
  - Поднимаем семафор barbers.
  - Идет стрижка.

```
void barber ( void )
{
  while ( TRUE )
  {
    down ( & customers ) ;      /* если customers == 0, т.е. посетителей нет, то
заблокируемся до появления посетителя */
    down ( & mutex ) ;      /* получаем доступ к waiting */
    waiting = wating – 1;       /* уменьшаем кол-во ожидающих клиентов */
    up ( & barbers ) ;       /* парикмахер готов к работе */
    up ( & mutex ) ;      /* освобождаем ресурс waiting */
    cut_hair();       /* процесс стрижки */
  }
}
```

- *Процесс "клиент"*
  - Вход в критическую секцию.
  - Проверка:

    Если количество ожидающих меньше количества кресел, то
    * клиент заходит в комнату ⇒ количество ожидающих увеличивается;
    * поднимается семафор customers, который используется для синхронизации доступа к парикмахеру;
    * выход из критическойй секции;
    * опускание семафора barbers: если парикмахер занят (barber=1), посетитель блокируется, когда парикмахер освододится (barber=0), начнется стрижка.

Если количество ожидающих больше количества кресел, то посетитель блокируется.

```
void customer( void )
{
  down ( & mutex ) ;        /* получаем доступ к waiting */
  if (waiting < CHAIRS)       /* есть место для ожидания */
  {
     waiting = waiting + 1 ;       /* увеличиваем кол-во ожидающих клиентов */
     up ( customers ) ;       /* если парикмахер спит, это его разбудит */
     up ( & mutex ) ;       /* освобождаем ресурс waiting */
       down ( barbers ) ;       /* если парикмахер занят, переходим в состояние
ожидания, иначе – занимаем парикмахера*/
     get_haircut () ;       /* процесс стрижки */
  }
  else
  {
    up ( & mutex ) ;      /* нет свободного кресла для ожидания – придется уйти */
  }
}
```

Реализация взаимодействия процессов

В этой теме будут обсуждаться как общие концепции организации взаимодействия процессов, так и средства взаимодействия процесса, которые предоставляются в системе UNIX.

Пусть есть два и более процессов. Нужно, чтобы эти процессы могли организовывать взаимодействие

1. исользуя совместный разделяемый ресурс;
2. выздействуя друг на друга.

Взаимодействие процессов:

1. *Взаимодействие в рамках локальной ЭВМ (одной ОС).* Компьютер работает под управлением одной ОС, т.е. есть централизованное управление. В рамках этой системы организуется взаимодействие

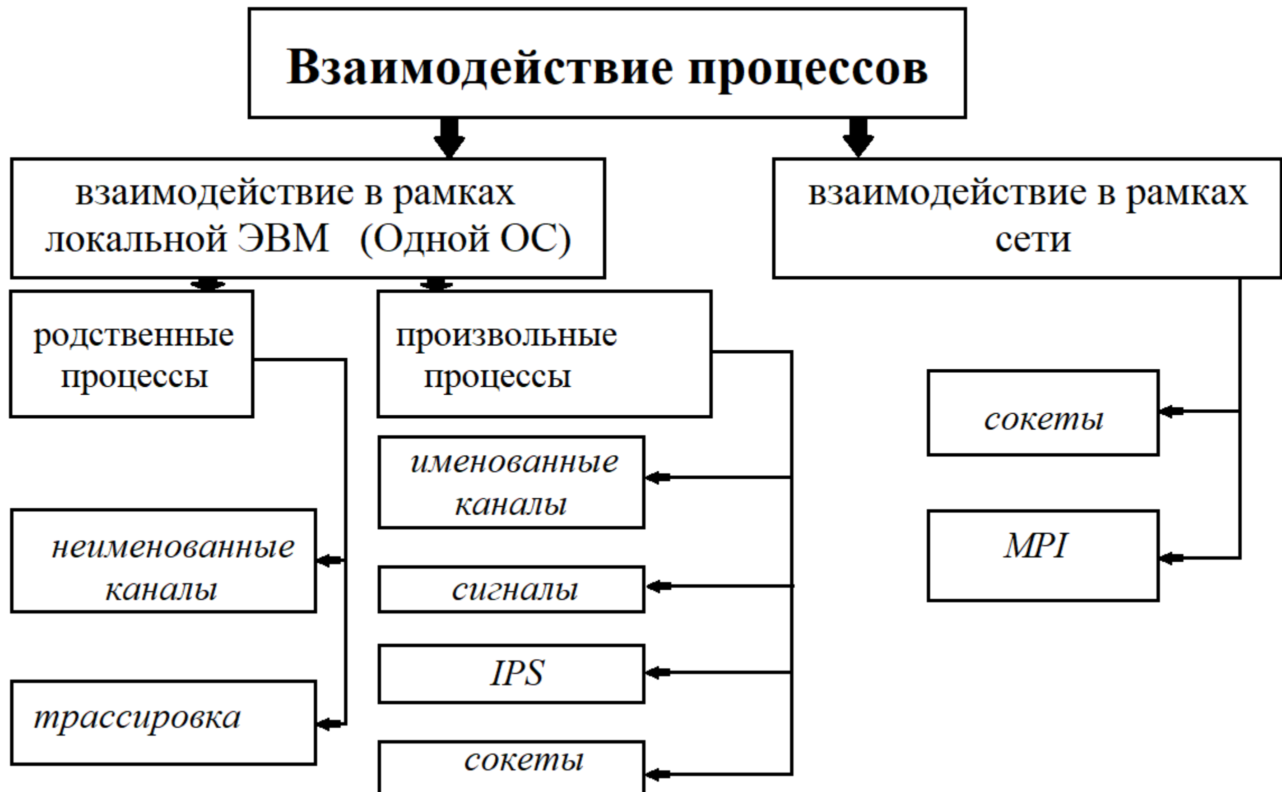

Рис. 14.1. Взаимодействие процессов

2. *Взаимодействие в рамках компьютерной сети* . Один процесс работает на одном компьютере, под управлением своей ОС, а другой процесс работает на другом компьютере, под управлением уже своей ОС. ОС этих компьютеров могут быть разными.

1. Взаимодействие в рамках локальной ЭВМ (одной ОС).
Можно разделить на две группы:

- *Взаимодействие родственных процессов.* Родственные процессы – такие процессы, которые при создании получили потенциальные возможность по организации взаимодействия (сыновний и родительский процессы).
  - В системе UNIX для родственных процессов возможно использование *неименованных каналов.* Это взаимодействие по модели send/receive.
  - *Трассировка* – взаимодействие по модели главный/подчиненный. В паре взаимодействующих процессов один из них на одновании каки-то условий объявляется главным процессом. Главный процесс имеет права по управлению подчиненным процессом.

ФАКУЛЬТЕТ ВЫЧИСЛИТЕЛЬНОЙ МАТЕМАТИКИ И КИБЕРНЕТИКИ МГУ ИМЕНИ М.В. ЛОМОНОСОВА

Пример: Программа отладчик – главный, а отлаживаемый процесс – подчиненный.

В радственных процессов нет проблем с именованием.

– *Взаимодействие произвольных процессов*

  Возникает проблема взамного именования, чтобы один процесс мог обращаться к другому процессу.

  Взаимодействие можно организовать через PID процессов. Возникнет следующая проблема: если один процесс завершится, на его место (PID с этим же номером) сформируется другой процесс.

  Для взаимодействия произвольных процессов существуют разные модели именования.

  * *Именованные каналы.* Для организации взаимодействия произвольных процессов используется некоторый специальный файл. Два взаимодействующий процесса могут работать со специальным файлом и обмениваться информацией через него.
  * *Сигналы*
  * *Системы IPC*
  * *Сокеты*

2. Взаимодействие в рамках сети

- *MPI* – интерфейсы передачи сообщений. Концепция взаимодействия по модели send/receive используется для прикладных нужд в распределенных системах
- *Сокеты* – базовое средство для реализации системных компонентов в распределенных система. На сокетах организуется подавляющее большинство сетевых приложений.

Сигналы

Определение. *Сигнал* – средство асинхронного уведомления процесса о наступлении некоторого события в системе.

Система UNIX позволяет осуществлять передачу взаимодействий от имени одного процесса к другому или от имени ОС к процессу. Эти взаимодействия передаются асинхронно.

Такие взаимодействия концептуально можно разделить на две группы: сигналы, которые приходят в процесс от имени ОС, уведомляющие о некотором событии, произошедшем в системе или в процессе.

Пусть выполняется некоторый процесс, в котором произошло деление на ноль. Произойдет прерывание. ОС "ловит прерывание"и посылает сигнал процессу, имеющий код, связанный с этой ошибкой. Процесс получает сигнал.

Три варианта обработки процессом прихода сигнала:

- Действие по умолчанию. Чаще всего процесс начинает завершаться с системным кодом завершения, равным номеру пришедшего сигнал.
- Процесс может определить для себя сигналы, на которые не будет никакой реакции.
- Внутри процесса заранее объявлено, что в случае прихода конкретного сигнала/сигналов будет вызов некоторой предопределенной функции. Возврат из функции будет осуществлен в точку, в которую пришел сигнал.

В каждой системе UNIX предопределен набор сигналов, которые могут быть в этой системе. Перечень этих сигналов и их описания представлены в файле signal.h, который находится в директории etc.

Работа с сигналами

Рассмотрим разные системные вызовы.

- #include<sys/types.h>
  #include<signal.h>
  int kill (pid_t pid, int sig) ;

  Для передачи сигнала от одного процесса другому используется системный вызов kill.

  Два параметра вызова kill:

  - pid – идентификатор процесса, которому посылается сигнал;
  - sig – номер посылаемого сигнала.

  Системный вызов возвращает 0 в случае успешного выполнения, а -1 в случае неуспешного выполнения (например, процесса с таким PID нет).

- #include<signal.h>
  void (*signal (int sig, void (*disp) (int))) (int) ;

  Системный вызов signal.

В прототипе сигнала описано обращение к некой функции signal. В этой функции два параметра:

- sig – номер посылаемого сигнала;
- disp – либо определенная пользователем функия-обработчик сигнала, либо одна из констант:

  SIG_DFL – обработка по умолчанию

  SIG_IGN – игнорирование.

Системный вызов возвращает указатель на функцию, которая задавала режим обработки этого сигнала.

Пример. Рассмотрим программу, в которой идет обращение к системному вызову signal. Для него назначаются функции обработчика SigHndlr. Эта функция описана в программе.

В этой функции определяем целочисленную переменную s, а затем ,в случае прихода сигнала, будет выдаваться некоторый текст. Этот текст может выдаваться только пять раз. После пятого раза обработка вернется в дефолтную(по умолчанию).

Один обработчик можно назначить на несколько сигналов. Через параметр s, который передает ОС, мы получим номер сигнала, из-за которого произошел вызов обработчика.

```
#include<sys/types.h>
#include<signal.h>
#include<stdio.h>
int count =0;
void SigHndlr ( int s )
{ printf ("\ n I got SIGIGNT %d tine(s) \ n ", count ++) ;
  if (count == 5 )
      signal ( SIGIGNT, SIG_DFL );   */  ????  */
}
int main (int argc, char **argv)
{   signal ( SIGINT, Sighndlr );
    while (1); /* " тело программы "*/
    return 0; }
```


ФАКУЛЬТЕТ ВЫЧИСЛИТЕЛЬНОЙ МАТЕМАТИКИ И КИБЕРНЕТИКИ МГУ ИМЕНИ М.В. ЛОМОНОСОВА

Пример. Программа "будильник"

Хотим заправить ввод некоторой информации со стандартного устройства ввода. Внутри процесса будет заведен будильник, который через положенный срок будет инициировать отправку сигнала будтльника от имени ОС в процесс.

- Определяем основную функци. Определили, что стандартный сигнал SIGALARM будет обрабатываться функцией alrm.
- Завели будильник. Системный вызов alarm установили на 5 временных единиц.
- Запрашивается некая строка. Если через 5 секунд не придет ввод, система пришлет сигнал SIGALARM и будет вызвана функция alrm с параметром s, в котором будет значение SIGALARM.
- Функция обработчик выведет строчку снова. Обработчик опять заведет будильник на 5 единиц и вернется.

```
#include<unistd.h>
#include<signal.h>
#include<stdio.h>
void alarm ( int s )
{
  printf ("\ n жду имя \ n ") ;
     alarm (5) ;
}
int main (int argc, char **argv)
{ char s[80] ;
{   signal ( SIGALARM, almr ); alarm(5) ;
    printf ("Введите имя \ n") ;
    for (;;) {          printf ("имя:") ;
         if (gets (s) !=NULL) break;
    }
    printf ("OK! \ n")
    return 0;
}
```


ФАКУЛЬТЕТ ВЫЧИСЛИТЕЛЬНОЙ МАТЕМАТИКИ И КИБЕРНЕТИКИ МГУ ИМЕНИ М.В. ЛОМОНОСОВА

Пример. Двухпроцессный вариант программы

- В основной программе устанавили обработчик сигнала SIGALRM на функцию alr.
- Сформировали сыновний процесс, который запрашивает строку.
- В родительском процессе бесконечный цикл. Идет обращение к системному вызову sleep, который блокирует процесс на 5 единиц времени. Затем идет обращение к системному вызову kill и сыновьему процессу отправляется сигнал SIGALRM.
- У сыновьего процесса вызывается обработчик и выдаст сообщение.
- Как только осуществляется ввод, выход из цикла и завершение работы.
- Отправляется сигнал SIGKILL родительскому процессу, после чего завершается родительский цикл и программа.

```
#include<signal.h>
#include<sys/types.h>
#include<stdio.h>
#include<unistd.h>
void alr ( int s )
{
   printf ( " \ n Быстрее!!! \ n " )
}
int main (int argc, char **argv)
{ char s[80] ;
{   int pid ;
   signal ( SIGALRM, alr) ;
   if ( pid = fork () ) { /* " отец " */}
   else { /* " сын "*/}
   return 0;
}
```

```
/* " отец "*/
for (;;) {     sleep(5);
   kill ( pid, SIGALRM );
}
—————————————————
/* " сын "*/
printf ( " \ n Введите имя \ n " )
for (;;)
{
   printf ("имя: ") ;
    if (gets (s) !=NULL) break;
}
printf ("OK! \ n") ;
kill ( getppid (), SIGKILL ) ;
}
```

Лекция 15. Взаимодействие процессов

Взаимодействие процессов

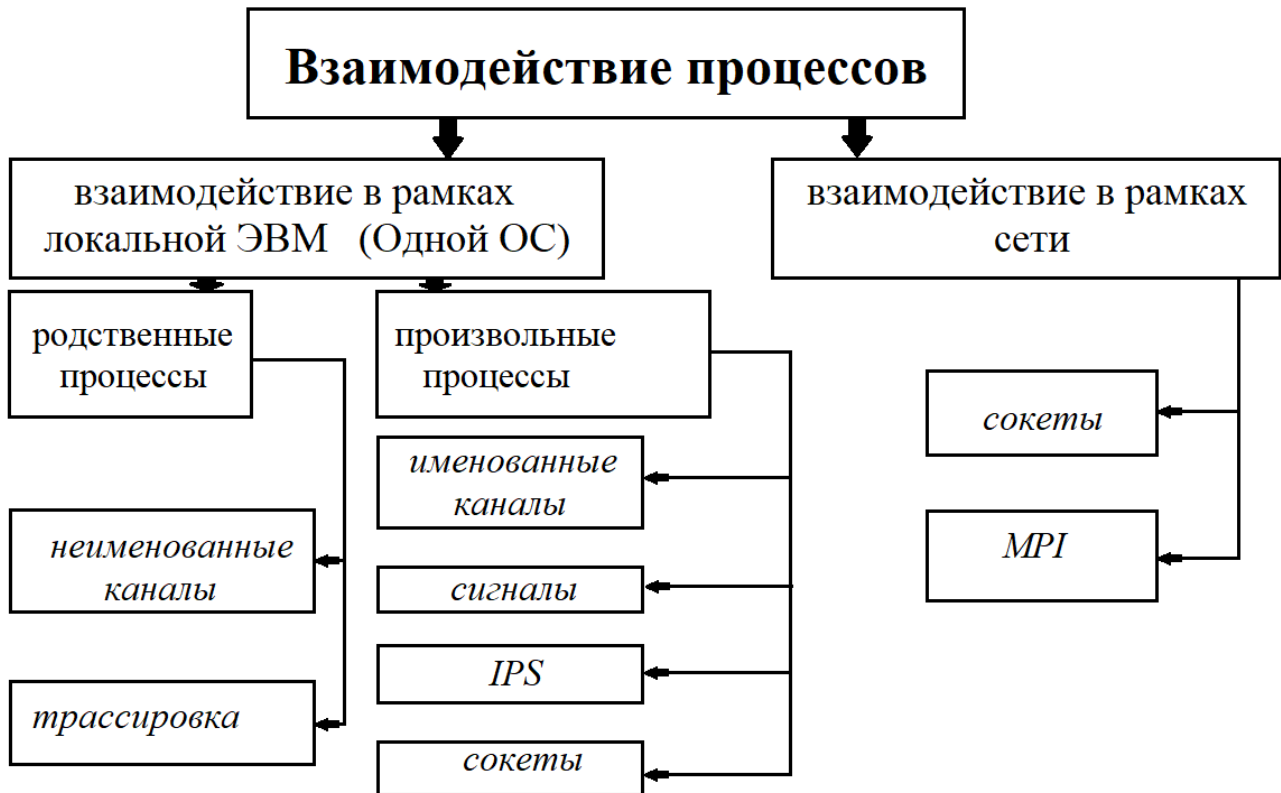

Рис. 15.1. Взаимодействие процессов

Взаимодействие процессов:

1. *Взаимодействие в рамках локальной ЭВМ (одной ОС).* Компьютер работает под управлением одной ОС, т.е. есть централизованное управление. В рамках этой системы организуется взаимодействие
2. *Взаимодействие в рамках компьютерной сети* . Один процесс работает на одном компьютере, под управлением своей ОС, а другой процесс работает на другом компьютере, под управлением уже своей ОС. ОС этих компьютеров могут быть разными.

1. Если рассматривать с точки зрения взаимодействия в рамках одного ловкального компьютера, можно разделить это взаимодействие на две группы:

- взаимодействие произвольных процессов

ФАКУЛЬТЕТ ВЫЧИСЛИТЕЛЬНОЙ МАТЕМАТИКИ И КИБЕРНЕТИКИ МГУ ИМЕНИ М.В. ЛОМОНОСОВА

- взаимодействие родственных процессов, которые связаны определенными условиями и характеристиками, которые формируются при их создании.

Была рассмотрена схема взаимодействий *произвольных процессов*, которая предполагает передачу некоторых воздействий от одного процесса другому. В качестве реализации этой схемы был рассмотрен аппарат сигналов операционной системы UNIX.

Аппарат сигналов позволяет передавать некоторое событие в процессе от имени ОС или от имени другого процесса.

Передача сигналов – модель взаимодействия произвольных процессов в системе UNIX, которая позволяет организовывать взаимодействие произвольных процессов.

- Для работы с сигналами есть системный вызов "передать сигналill.

  Системный вызов kill имеет два параметра:

  - pid – идентификатор процесса, которому посылается сигнал;
  - sig – номер посылаемого сигнала.

  Все взаимодействия в виде сигналов имеют свое имя, каждому имени дан номер. Можно посмотреть перечень всех сигналов, которые возможны в данной реализации системы. В этом файле есть мнемоническое обозначение сигнала и его целочисленное значение, а также указано, как сигнал может обрабатываться.

В системе UNIX приход сигнала в процесс может вызвать три варианта обработки:

- Обработка по умолчанию. Чаще всего это завершение процесса и определение системной составляющей завершения как номера сигнала.

  Системный вызов wait. Ожидание завершения сыновнего процесса. Когда процесс завершился или у него произошло какое-то событие, родительский процесс получает значение статуса. В статусе есть системная составляющая (номер сигнала) и пользовательская составляющая (значение, которое передается через системный вызов exit).

- Процесс может определить для себя некоторые сигналы, на которые не будет никакой реакции.

ФАКУЛЬТЕТ
ВЫЧИСЛИТЕЛЬНОЙ МАТЕМАТИКИ И
КИБЕРНЕТИКИ МГУ ИМЕНИ М.В.
ЛОМОНОСОВА

- Процесс может назначить функцию-обработчик пришедшего сигнала. Если используется установка функции-обработчика для некоторого сигнала, этот сигнал позволяет устанавливать обработку. Функция-обработчик вызывается асинхронно в тот момент, когда приходит сигнал, для которого эта функция определена. Эта связь осуществляется через системный вызов signal.

  Эта функция обработчик имеет два параметра:

  - sig – номер сигнала;
  - disp – указатель на функию-обработчик. В виде этого параметра может быть одна из функций, определенных в программе. Эта функция может принимать целочисленные значения и возвращать указатель на функцию-обработчик.

Работа с сигналами

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

При корректном обращении к системному вызову "сигнал если сигнал, который указан первым параметром перехватываемый, то ф-ия, указатель на которую определен вторым параметром, назначается обработчиком этого сигнала.

Значит, если эта функция определена в программе, то при приходе данного сигнала будет вызвана эта функция. Номер пришедшего сигнала, который инициировал вызов этой функции будет передан в качестве параметра в функцию обработчика.

Это позволяет назначить одну функцию на обработку нескольких сигналов, а разделять внутри через анализ этого параметра, через который получаем номер сигнала.

Системный вызов signal при обращении возвращает предыдущий режим обработки.

Пример. Можно назначить одну функцию-обработчик f_1 и вторую функцию-обработчик f_2. При втором обращении, системный вызов signal вернет указатель на функцию f_1 .

Или же истемный вызов сигнал при обращении вернет указатель на тот статус обработки, который был в предыдущий момент.

Две предопределенные константы типа указателя:

- SIG_DFL – константа режима обработки по умолчанию;
- SIG_IGN – игнорирование.

Если было игнорирование сигнала, а затем нужно назначить функцию-обработчик f_1 с помощью системного вызова signal, то после обращения к этому вызову будет получена константа SIG_IGN.

Возврат из функции-обработчика. Пришел сигнал, вызвана функция- обработчик, внутри нее произошли какие-то действия, затем происходит из функции. Возврат осуществляется в точку прихода сигнала (некоторая аналогия программной обработки прерывания).

Аппарат сигналов во многом связан с обработкой прерывания.

Рассмотрим схему обработки прерывания для системы UNIX с точки зрения связи аппарата прерываний и передачи сигнала:

- на аппаратном уровне фиксируется ошибка;
- управление автоматически передается в точку входа в ту часть ОС, которая обрабатывает прерывания;
- ОС начинает обрабатывать прерывание,затем передает от своего имени соответствующий сигнал процессу (процесс, в котором произошла ошибка не заканчивается);
- далее сигнал обрабатывается программой.

Важно. Передача сигналов – модель взаимодействия произвольных процессов в системе UNIX, которая позволяет организовывать взаимодействие произвольных процессов.

Неименованные каналы. Системный вызов pipe()

Ранее была рассмотрено средство синхронизации send/receive. В общем случае это средство позволяет организовывать взаимодействие не только в рамках локальной машины, но и в рамках сети.

Неименованные каналы – одно из средств отправки и получения сообщений класса send/receive.

Данная модель предполагает взаимодействие родственных процессов.

```
#include <unistd.h>
int pipe (int *pipes)
```

В UNIX системе в параметрах настройки можно объявить ресурсы неименованных каналах:

ФАКУЛЬТЕТ ВЫЧИСЛИТЕЛЬНОЙ МАТЕМАТИКИ И КИБЕРНЕТИКИ МГУ ИМЕНИ М.В. ЛОМОНОСОВА

- размер буфера для размещения всевозможных неименованных каналов;
- предельный размер одного неименованного канала.

Системный вызов pipe(). Его аргументом является указатель на две целочисленные перемнные (массив из двух целочисленных значений).

Если вызов pipe срабатывает успешно, система выделяет ресурс в буфере, определенном для неименованных каналов, и связывает этот ресурс с двумя файловыми дескрипторами. Через один дескриптор может осуществляться чтение информации, а через другой – запись информации в этот ресурс.

- Чтение информации осуществляется по стратегии FIFO. В таком случае, не будет средства работы с файлами "перемещение файлового указателя потому что правило перемещения связано с чтением и записью.

⇒ В нулевом элементе массива получаем файловый дескриптор, через который осуществляется чтение, а первый элемент массива – файловый дескриптор, через который можно писать в неименованный канал.

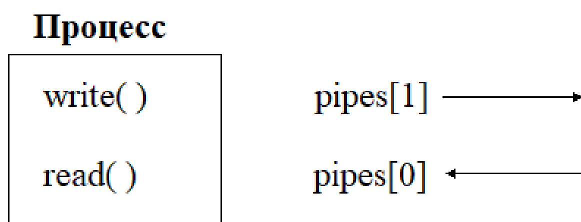

Рис. 15.2. Системный вызов pipe()

Если системный вызов pipe() в качестве кода-ответа выдает -1, значит он не сработал и не смог создать канал.

Пример не срабатывания вызова pipe: нет свободных файловых дескрипторов. Для каждого определено придельное количество файловых дескрипторов, которые могут быть открыты в процессе. В каждом процессе есть позиционная таблица, в которой i-я строка – номер файлового дескриптора. При обращении к pipe() необходимо иметь две свободные строки в таблице. Если ОС не находит таких строк, получаем отказ с кодом ответа -1. Причину можно найти в переменной erno. Для этого нужно подключить системный файл erno.h.

Программа копирования строк.

- Определили массив pipes, обратились к системному вызову pipe(), создали канал.

- Записываем в канал текстовую строку s, которую проинициализировали.
- Прочтение строки s из канала.
- Печать содержания массива в буфер.

```
int main (int argc, char **argv)
{
   char *s = "channel " ;
   char buf[80] ;
   int pipes[2] ;

   pipe ( pipes ) ;
   write ( pipes[1], s, strlen(s) + 1);
   read ( pipes[0], buf, strlen(s) + 1);
   close ( pipes[0] );
   close ( pipes[1] );
   printf (" % \ n ", buf );
}
```

Приведенный выше пример демонстрировал функциональность.

Более осмысленный пример При создании сыновьего процесса открытые файловые дескрипторы родительского процесса наследуются в сыновьем. При смене тела процесса открытые файловые дескрипторы тоже сохраняются.

Системный вызов pipe() создал объект, который находится в памяти ОС, с которым связаны два дескриптора: через один дескриптор можно записывать туда информацию, а через другой – читать. Эти файловые дескрипторы симметричны.

При работа с файловым дескриптором через который можно **читать**, происходит следующее:

- если из канала читается порция данных меньшего объема чем данные, которые находятся в канале, они изымаются из канала и переносятся в читающий процесс, т.е. нет синхронизации по количеству чтения и записи (можно записать 100 байтов и 100 раз прочесть по 1 байту);
- если из канала читается порция данных, которая превосходит по объему данные, которые находятся в канале, и в системе есть хотя бы один открытый дескриптор, который может писать в этот канал, то данные, которые присутствуют в канале, переносятся в процесс (освобождается ресурс в неименованном канале), затем процесс блокируется. Условия разблокировки процесса:

– в канале появилась недостающая порция данных

⇒ дочитаем и канал разблокируется

– в системе закроется последний файловый дескриптор, который может писать в этот канал.

⇒ процесс разблокируется и получит "end of file".

Процесс **записи** происходит следующим образом:

- при записи порции данных, которая превышает объем свободного пространства в неименованном канале, сначала записывается порция данных, равная размеру свободного пространства, затем процесс будет ожидать освобождения места в неименованном канале. Потом снова перенесет еще порцию и т.д.

  ⇒ ресурс переместится из процесса в часть ОС, ассоциированную с неименованным каналом.

Неименованный канал – средство для взаимодействия родственных процессов.

В родительском процессе посредством системного вызова pipe() создается неименованный канал. При создании сыновних процесса, в каждом сыновнем процессе передаются в наследство открытые файловые дескрипторы, т.е. фактически передается доступ в канал.

Если в родительском процессе создали неименованный канал, а затем некоторое количество раз был вызван fork(), то каждый сыновний процесс может писать в канал и читать из канала. Во многих случаях нужна синхронизация, чтобы при передаче информации в канал она передавалась именно тому процессу, которому адресована.

Простейшая модель использования канала – связь двух процессов. В одном процессе осуществляется только запись в канал, а в другом – чтение (см. рис. 15.3).

Примечание. Если нужна двунаправленная связь, необходимо создать два канала.

Разберем следующий пример (разбираются только позитивные ветки):

- обращение к системному вызову pipe() – создание канала;

- обращение к системному вызову fork()

  → попали в процесс-родитель. Закрываем нулевой файловый дескриптор (дескриптор, через который осуществляется чтение) и через первый дескриптор записываем информацию в канал. После записи дескриптор закрывается и процесс завершается.

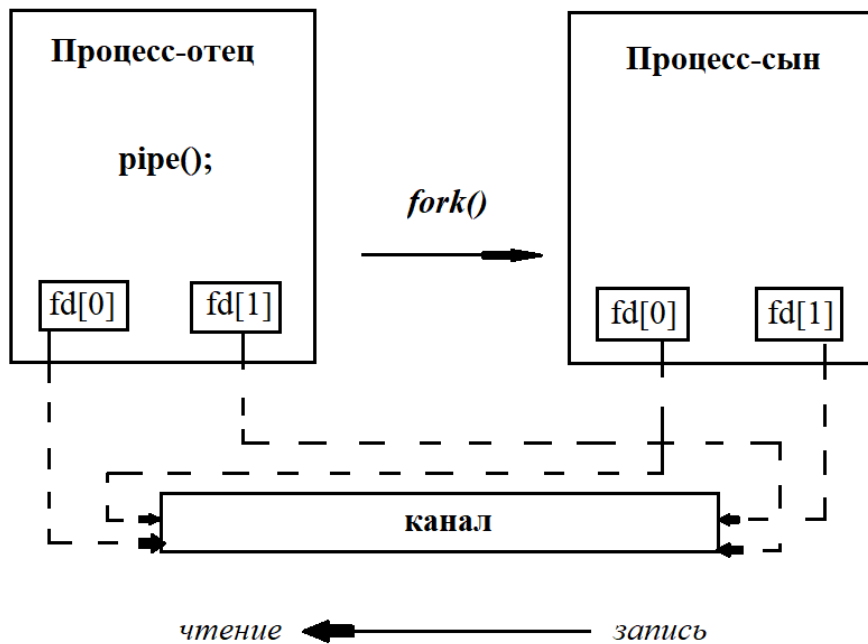

Рис. 15.3. Связь родительского и сыновнего процессов через канал

- else – образение к сыновнему процессу. Закрываем нулевой дескриптор и начинаем читать из канала.

Если не закрыть первый дескриптор (выделено жирным шрифтом), процесс зависнет. Он будет бесконечно ждать end of file.

```
int main (int argc, char **argv)
{
  int fd[2] ;
  pipe (fd) ;
  if (fork () { close (fd [0] ) ;
                write ( fd[1], ... ) ;
                ... ;
                close ( fd [1] ) ;
                ... ;
                }
  else { close ( fd[1]) ;
           while ( read ( fd[0], ... ) ) { ... }
     ...
  }
}
```


ФАКУЛЬТЕТ ВЫЧИСЛИТЕЛЬНОЙ МАТЕМАТИКИ И КИБЕРНЕТИКИ МГУ ИМЕНИ М.В. ЛОМОНОСОВА

Конвейер

Пример. Реализация конвейера

Инетрпретатор команд UNIX позволяет в качестве командной строки выдавать цепочки команд.

В одной командной строке можно связать команды, которые будут специальным образом обработаны.

Одной из возможностей интерпретатора команд является запуск конвейера. Когда командная строка содержит несколько команд, разделенных специальным символом |. При запуске команды интерпретатор команд дает этому запускающемуся процессу предопределенные открытые файловые дескрипторы: 0 – стандартное устройство чтения, 1 – стандартное устройство записи, 2 – стандартное устройство записи диагностических текстов.

Конвейер позволяет связать стандартное устройство вывода предыдущей команды со стандартным устройством вводу последующей команды.

Возьмем две команды: printf и wc. Команда wc позволяет определить кеноторую статистику по той информации, которая была передана в виде файла на стандартное устройство ввода.

Нужно связать эти команды, чтобы получить статистику по файлу, переданному команде printf.

Рассмотрим формирование конвейера.

- Открываем канал
- В сыновнем процессе обращаемся к вызову dup2, у этого системного вызова два параметра (два открытых системе номера файловых дескрипторов).
  - Dup2: закрывается открытый файловый дескриптор, который передается через второй параметр, затем этот файловый дескриптор связывается с файловым дескриптором того файла, файловый дескриптор которого передается первым параметром. Т.е. на файловый дескриптор, который передается вторым параметром, назначается конкретный файл.
  - Закрываем файловые дескрипторы, которые были ассоциированы с каналом.
  - Меняем тело процесса на процесс printf.

    → сыновний процесс стал процессом printf, у которого стандартное устройство вывода – дескриптор записи в неименованный канал.

- В родительском процессе происходит все то же самое, но относительного нулевого файлового дескриптора. Это значит, что вместо стандартного ввода будет использоваться канальный дескриптор чтения.
  - Закрыли файловые дескрипторы, которые связаны с каналом.
  - Меняем тело родительского процесса на процесс wc

```
int main (int argc, char **argv)
{
  int fd[2] ;

  pipe (fd) ;
  if (fork () == 0 ) { dup2 (fd [1], 1 ) ;      */вместо стандартного устройства вывода ставится файловый дескриптор записи в неименованный канал */
          close ( fd [1] ) ;
          close ( fd [0] ) ;
          execl (" /usr/bin/print ", "print", 0) ;
          }
  dup2 (fd [0], 1 ) ;
  close ( fd [0] ) ;
  close ( fd [1] ) ;
  execl (" /usr/bin/wc ", "wc", 0) ;
}
```

Не известно, в каком порядке будут выполнены родительский и сыновний процессы.

Совместное использование сигналов и каналов "пинг-понг"

- Рассмотрим другую модель взаимодействия процессов посредством неименованных каналов. В этой модели каждый из процессов может читать и записывать информацию.
- Будем считать, что роль мячика выполняет некоторый счетчик перемещений. Программа имеет ограничение на количество перемещений, которое задается в переменной MAX_CNT.
- Есть два взаимодействующих процесса, передача мяча с одной стороны на другую – запись значения счетчика в канал. Когда второй процесс принимает подачи, он считывает значение из канала, увеличивает его на 1, а затем записывает.

- В качестве синхронизирующего элемента используются сигналы и системный вызов обработки сигнала. В данном случае используется сигнал SIGUSR1.

  Примечание. Все UNIX-подобные системы имеют в составе своих сигналов сигналы, которые не связаны с каким-то событием внутри системы. Они предназначены для программного взаимодействия. Это пользовательские сигналы.

- Создали канал.
- Установили обработчик приходящего сигнала SIGUSR в функцию SigHndl.
- Обнуляется счетчик подач.
- Запускается процесс игры.

  Родительский процесс:

  - после вызова fork() и у родителя, и у сыновнего процесса наследуются установленные режимы обработки сигналов.
  - записывается счетчик операций в неименованный канал;
  - ожидание завершения в сыновнем процессе. Когда сигнал придет, управление передастся в функцию обработчика, она проработает, затем вернутся в прерванное место. Этот процесс бцдет повторяться, пока сыновний процесс не завершится.
  - Процесс-родитель закрывает файловые дескрипторы и завершается.

  Сыновний процесс:

  - осуществляет чтение из канала;
  - после прочтение первого значения, он определяет PID родителя в переменную getppid;
  - записывает в канал подачу;
  - посылает сигнал родителю ;

  Обработчик сигналов:

  Если счетчик итерации не дошел до границы

  - читаем из неименованного канала переменную;
  - увеличивает ее на единицу;
  - пишем в канал новое значение (передаем подачу);

- синхронизация(сообщаем партнеру о подаче).

Если счетчик итерации дошел до границы

- проверяем, в каком обработчике находимся;
- если находимся в процессе-сыне, выдается стандартное сообщение, закрываеются дескрипторы, совершается выход из программы. Родительский процесс тоже завершается.
- если находимся в процессе-родителе, посылается сигнал, чтобы закончился процесс-сын.

Необходимо адаптировать эту программу.

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

# define MAX_CVT 100
int target_pid, cnt ;
int fd[2] ;
int status;

int main (int argc, char**argv ) ;
{ pipe (fd) ;    signal ( SIGUSR1, SigHndlr ) ;
   cnt = 0 ;
   if ( target_pid = fork() ) { /* процесс-родитель */
          write ( fd[1], &cnt, sizeof(int) ) ;
          while ( wait(&status) == -1 ) ;
          printf ( "Parent is going to be terminated \ n " ) ;
          close ( fd[1] ) ;
          close ( fd[0] ) ;
          return 0 ;
          }
else /* процесс-сын */
          {
```

```
        read ( fd[0], &cnt, sizeof(int) ) ;    /* старт синхроницации*/
        target_pid = getppid() ;
        write ( fd[1], &cnt, sizeof(int) ) ;          kill ( target_pid, SIGUSR1 ) ;
        for (;;) ;          }
}

void SigHndlr (int s )
{
     if ( cnt<MAX_CNT )
     {
          read ( fd[0], &cnt, sizeof(int) ) ;
          printf(" %d \ n", cnt ) ;
          cnt++
          write ( fd[1], &cnt, sizeof(int) ) ;
          kill ( target_pid, SIGUSR1 ) ;
     }
     else
     if ( target_pid = getppid() ) /* процесс-сын*/
     {
          printf ("Child is going to be terminated ;\ n " ) ;
          close ( fd[1] ) ;
        close ( fd[0] ) ;
          exit(0) ;
     } else /*процесс-родитель*/              kill ( target_pid, SUGUSR1 ) ;
}
```

Именованные каналы

int mkfifo (char *pathname, mode_t mode) ;

Вспомним, что НЕименованные каналы – средство взаимодействия родственных процессов.

В UNIX-системах имеется альтернатива – именованные каналы (FIFO файлы).

FIFO-файлы могут быть единовременно открыты в двух и более процессах. Эти процессы могут в определенные моменты времени писать информацию в эти файлы и читать из них информацию. Т.е. доступ к взаимодействию осуществляется посредством открытия существующего FIFO-файла.

Лекция 16. Межпроцессорное взаимодействие

Модель межпроцессорного взаимодействия "Главный-подчиненный"

До настоящего времени было рассмотрено симметричное взаимодейсвие в рамках одного компьютера, когда взаимодействующие стороны в общем случае имеют одинаковые возможности по функциям взаимодействия.

Рассмотрим ассиметричное взаимодействие, в котором разные процессы имеют разные возможности по взаимодействию.

Модель главный-подчиненный в UNIX-системе. Главные и подчиненные процессы имеют разные наборы возможностей по взаимодействию, а также главные процессы могут управлять подчиненными процессами.

Используется системный вызов prtace.

Один из взаимодействующих процессов назначается главным, обычно родительский процесс.

Главный процесс может:

1. остановить/запустить/продолжить выполнение подчиненного процесса;
2. выполнять относительно подчиненных процессов действия по трассировке;
3. может прочесть или модифицировать сегмент кода подчиненного процесса;
4. считать или записать информацию, размещенную в сегменте данных;
5. прочесть и изменить информацию, которая размещается в системной части контекста процесса, т.е. он может прочесть и изменить содержимое регистровой памяти подчиненного процесса;
6. включить и выключить особые режимы работы подчиненного процесса (например, шаговый режим);
7. изменить порядок выполнения команд подчиненного процесса, приостановить и начать с точки заданного адреса.

Подчиненный процесс - это сыновний процесс, который может разрешить свою трассировку.

Для реализации этих функций используется вызов ptrace. Структура параметров:

```
# include<sys/ptrace.h>
int ptrace ( int cmd, int pid, int addr, int data ) ;
```


ФАКУЛЬТЕТ
ВЫЧИСЛИТЕЛЬНОЙ МАТЕМАТИКИ И
КИБЕРНЕТИКИ МГУ ИМЕНИ М.В.
ЛОМОНОСОВА

- cmd – код выполняемой команды
- pid – идентификатор процесса-потомка
- addr – некоторый адрес в адресном пространстве процесса-потомка
- data – слово информации (данные)

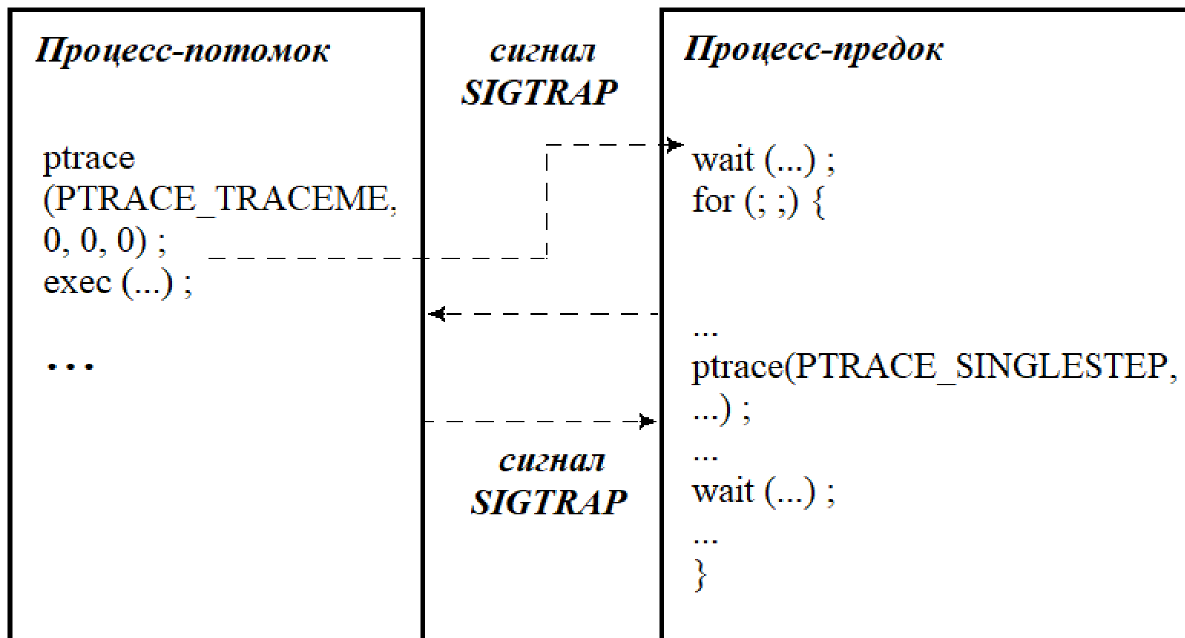

Рис. 16.1. Общвя схема трассировки процессов

Рассмотрим схему трассировки процессов.

- Процесс-родитель создает сыновний процесс. Он обращается с системному вызову wait и блокируется до возникновения события в сыновнем процессе.
- Сыновний процесс обращается к системному вызову ptrace с командой TRACEME (разрешить трассировку)
- Сыновний процесс обращается к системному вызову exec. Если процесс работает в режиме разделенной трассировке, то при обращении к системному вызову exec происходит замена тела процесса, перед выполнением первой входной команды нового тела, ОС генерирует для данного процесса приход некоторого предопределенного сигнала (например, SIGTRAP).
- После того, как в процесс-сын пришел сигнал SIGTRAP, процесс-родитель (главный процесс) получает информацию об этом событии.

- Процесс-родитель разблокируется, получит PID процесса-сына. (Сыновний процесс приостановлен.) В переменной, указатель на которую передавался системным вызовом wait, будет указано, что пришел сигнал.

  ⇒ сыновний процесс приостановлен, процесс-родитель получил информацию о том, что трассируемый процесс сменил тело, и все готово для начал трассировки.

- Главный процесс может прочесть/изменить информацию подчиненного процесса. Главный процесс может продолжить выполнение подчиненного процесса с заданного адреса или с прерванной точки.

- Обращение к системному вызову ptrace с указанием информации, откуда продолжить сыновний процесс.

- Запуск сыновнего процесса.

- Процесс-родитель снова встает на ожидание события в подсиненном процессе.

Главный процесс может остановить подчиненный процесс, отправив ему некоторый предопределенный сигнал(системный вызовом kill).

Схема установки контрольной точки

Рассмотрим, как можно организовать отладчик.
Действия отладчика:

- считать/записать информацию в память (умеем);
- считать/записать информацию в регистры общего назначения (умеем);
- установка контрольной точки (не умеем);

Чтобы установить контрольную точку по некоторому адресу A необходимо, чтобы подчиненный процесс находился в состоянии остановки.

Последовательность действий, необходимых для установки контрольной точки по адресу A:

- считывание из сегмента кода по адресу A исходную информацию;
- сохранение этой информации в таблице контрольных точек;
- по адресу контрольной точки записывается сегмент кода, команду которая вызовет прерывание и возникновение сигнала;

- дается команда подчиненному процессу(через ptrace) продолжить выполнение с точки остановки;
- кога подчиненный процесс дойдет до точки А, произойдет прерывание, ОС пришлет подчиненному процессу сигнал
- приход этого сигнала будет получен главным процессом $\Rightarrow$ главный процесс выйдет из состояния ожидания;
- проверка: прерывание вызвано ошибкой или приходом на контрольную точку. Проверка идет через таблицу контрольных точек. Если адрес совпадает, значит, мы в контрольной точке.
- если контрольная точка многоразовая, восстановление содержимого ячейки по адресу контрольной точки из таблицы контрольных точек
- Затем переключение работы подчиненного процесса в шаговый режим. После этого управление передается посредством ptrace на адрес контрольной точки.
- Опять восстанавливается контрольная точки из таблицы, снимается шаговый режим, запуск продолжения программы.

К такому адресному отладчику можно добавить мнемонические работы, добавить таблицу имен и адреса имен. Тогда можно писать интерфейс "установить контрольную точку"по мнемоническому адресу.

Рассмотрим работу с программой на языке высокого уровня. Компилятор или редактор связи сформирует контрольные точки. Будет табличка, по которой каждый оператор будет иметь диапазон адресов. Контрольную точку можно будет вставлять в голову оператора.

Для работы с автоматическими переменными компилятор должен сделать табличку для каждого блока локальных переменных. В такой табличке должно быть имя локальной переменной, ее тип, смещение относительно вершины стека.

лекция 17. Операционные системы – Система IPC

Система межпроцессорного взаимодействия IPC. Общая концепция

Ранее обсуждались базовые средства взаимодействия процессов, которые предоставляют UNIX-системы. К ним относятся аппарат передачи/приема сигналов, взаимодействие посредством именованных каналов, взаимодействие посредством неименованных каналов, трассировка. Еще одним средством взаимодействия, которое предоставляется UNIX-системой, является *система межпроцессорного взаимодействия IPC (inter-process communication)*. Эта система применяется для программирования распределенных программных систем, работающих в рамках одного компьютера (работающих под управлением одной ОС).

Концепция взаимодействия посредством IPC. Предполагается, что ОС UNIX предоставляет своим процессам доступ к возможности создания и использованияраспределенных ресурсов.

Система IPC поддерживает три типа разделяемых ресурсов:

- *Очередь сообщений* – распределенный(разделяемый) ресурс, модель send/receive. В такой ресурс можно записывать/выбирать сообщения по некоторой условной стратегии FIFO. Очередь сообщений IPC поддерживает не только неименованный прием и передачу сообщений, но и возможность типизации сообщений.

- *Распределенная память* – этот ресурс позволяет создавать некоторую область оперативной памяти, к которой предоставляется возможность подключения разных процессов.

- *Массив семафоров* – ресурс в виде последовательности одностипных элементов, где каждым из элементов является семафор. Массив семафоров позволяет организовывать синхронизацию.

Система IPC позволяет организовывать взаимодействие произвольных процессов. Проблемой для взаимодействия произвольных процессов является именование. В данном случае нельзя привязаться к PIDам процессов. Возможно взаимодействие посредством именованных каналов, где для именования используются имена специальных файлов (FIFO файлы).

В системе IPC используется модель именования, основанная на *использовании ключей.* При предоставлении возможности взаимодействия группе процессов через

один из разделяемых ресурсов, можно обратиться к ОС посредством специального системного вызова, который вызывает создание этого ресурса с определенными характеристиками. Данный системный вызов ассоциирует вновь созданный разделяемый ресурс с ключом, где ключ – целое положительное значение. Система позволяет программисту назначить ключ. Ресурс будет идентифицироваться по ключу.

Минус: возможность совпадения ключей.

Для избежания создания одинаковых ключей система IPC предоставляет пользователям системный вызов ftok, который генерирует уникальный ключ.

Примечание. При создании/подключении к ресурсам системы IPC следует использовать только ключи, сгенерированные через системный вызов ftok

```
#include <sys/types.h>
#include <sys/ipe.h>
key_t ftok ( char * filename, char proj )
```

Параметры:

- *filename* – указатель на полное имя существующего в файловой системе файла (строка, показывающая путь от корня файловой системы до файла);
- *proj* – добавочный символ

Для генерации ключа используется не только имя файла, но и его свойства, внутренние характеристики.

<ResName>get (key, ..., flags) Для всех ресурсов есть некоторая унифицированная группа системных вызовов, которые позволяют выполнять действия по подключению к существующему ресурсу или создание и подключение к ресурсую.

Рассмотрим структуру имени таких вызовов: в префиксной части находится некотора аббревиатура, связанная с именем конкретного ресурса, а в суффиксной части стит get.

В системном вызове get могут быть разные параметры, связанные с конкретным ресурсом. Каждый из системных вызовов этой группы первым параметром имеет ключ, а последним – параметр флаги.

Через флаги определяются права доступа: чтение, запись, исполнение.

Эти права распределяются на категории типа пользователей:

- пользователь,
- группа, к которой принадлежит пользователь (за исключением самого пользователя),

- все остальные (за исключением группы).

⇒ UNIX-система при работе с правами доступа имеет три категории пользователей,а в каждой категории может быть один из типов доступа.

Флаги создания/подключения:

- *IPC_PRIVATE*. Это запрет подключения к ресурсу других процессов, доступность только породившему процессу. Он позволяет перевести систему взаимодействия из взаимодействия произвольных процессов на взаимодействие родственных процессов.
- *IPC_CREAT*. При обращении к системному вызову get хотим либо создать новый ресурс с переданным ключом, либо подключиться к существующему ресурсу.
- *IPC_EXCL* (+ IPC_CREAT). Позволяет создавать новый ресурс. При обращении к get с этим флагом в случае, если ресурс с данным ключом уже есть, получим ошибку.
- ...

  ⇒ IPC имеет две модели взаимодействия:

  – взаимодействие произвольных процессов;
  – взаимодействие родственных процессов

Очередь сообщений IPC

Одним из трех ресурсов, которые можно создавать средствами IPC и с которыми можно работать является очередь сообщений. Это ресурс, который работает по модели FIFO.

Данный ресурс позволяет представить как очередь FIFO последовательность всевозможных сообщений, которые в него записаны. ИЗ сообщений, записанных в очередь сообщений, можно ассоциировать тип сообщения. Т.е. при записи сообщения в очередь два объекта: само сообщение и тип (характеристика информации).

Эту очередь можно представить как

- одну сквозную очередь (традиционная очередь FIFO)
- объединение подочередей, содержащих сообщения одного типа.

Допустим, можно записывать сообщения типа А и типа В. Если рассматривать эту очередь как нетипизированную, тогда работа с ней будет осуществляться по классической модели FIFO. Если нужно получать сообщения, поступившие в очередь, одного типа, можно организовать доступ к этой подочереди.

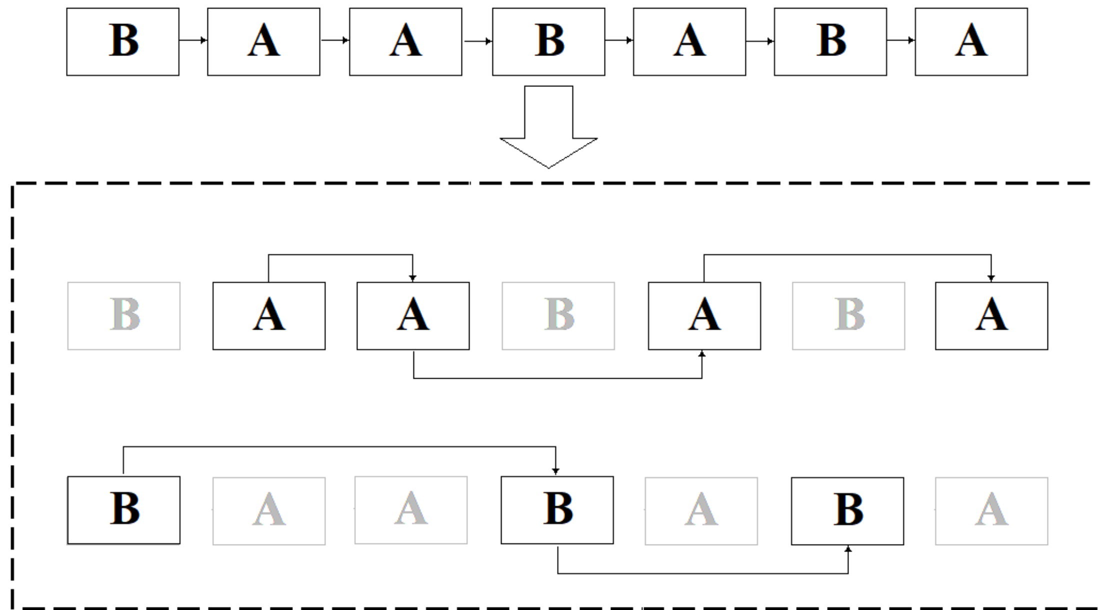

Рис. 17.1. Очередь сообщений

Доступ к очереди сообщений.

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/message.h>
int msgget (key_t key, int msgflag)
```

Системный вызов msgget имеет два параметра:

- key – ключ
- msgflag – флаги, управляющие поведением вызова

В случае успеха вызов возвращает положительный дескриптор очереди, который может в дальнейшем использоваться для операций с ней, в случае неудачи -1, причину отказа смотрим в erno.

ФАКУЛЬТЕТ ВЫЧИСЛИТЕЛЬНОЙ МАТЕМАТИКИ И КИБЕРНЕТИКИ МГУ ИМЕНИ М.В. ЛОМОНОСОВА

Отправка сообщения.

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
 int msgsnd (int msqid, const void *msgp, size_t msgsz, int msgflg)
```

Два системных вызова, которые реализуют модель send/receive : msgsnd. Параметры:

- msqid – дескриптор очереди, полученный в результате вызова msgget()
- msgp – указатель на буфер, в котором находится сообщение. В этой структуре есть два поля:
  - long msgtype – тип сообщения
  - char msgtext[ ] – указатель на массив, в котором находится само сообщение (тело сообщения)
- msgsz - предельный размер сообщения (буфера).
- msgflg - может принимать значения
  - 0 – блокировка, если для посылки сообщения недостаточно системных ресурсов. В случае отсутствия флага IPC_NOWAIT вызывающий процесс будет блокирован (т.е. приостановит работу), если для посылки сообщения недостаточно системных ресурсов, т.е. если полная длина сообщений в очереди будет больше максимально допустимого.
  - IPC_NOWAIT. Если флаг IPC_NOWAIT будет установлен, то в такой ситуации выход из вызова произойдет немедленно, и возвращаемое значение будет равно –1. В случае удачной записи возвращаемое значение вызова равно 0.

В заголовочном файле <sys/msg.h> определена константа MSGMAX, описывающая максимальный размер тела сообщения. При попытке отправить сообщение, у которого число элементов в массиве msgtext превышает это значение, системный вызов вернет –1.

Получение сообщений

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
int msgrcv (int msqid, void *msgp, size_t msgsz, long msgtyp, int msgflg)
```

Первые три аргумента аналогичны аргументам предыдущего вызова. Параметры:

- msqid – дескриптор очереди
- msgp – указатель на буфер, в который хотим получить сообщение
- msgsz – значение размера сообщения, которые хотим прочесть
- msgtyp указывает тип сообщения, которое процесс желает получить.
  - Если значение этого аргумента равно 0, то будет получено сообщение любого типа(работа со сквозной очередью).
  - Если значение аргумента msgtyp больше 0, из очереди будет извлечено сообщение указанного типа.
  - Если же значение аргумента msgtyp отрицательно, то тип принимаемого сообщения определяется как наименьшее значение среди типов, которые меньше модуля msgtyp

Управление очередью сообщений.

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
int msgctl (int msqid, int cmd, struct msgid_ds *buf)
```

Параметры:

- msqid – дескриптор очереди,
- cmd - команда, которую необходимо выполнить. Возможные значения:
  - IPC_STAT – скопировать структуру, описывающую управляющие параметры очереди по адресу, указанному в параметре buf

- IPC_SET – заменить структуру, описывающую управляющие параметры очереди, на структуру, находящуюся по адресу, указанному в параметре buf
- IPC_RMID – удалить очередь. Удалить очередь может только процесс, у которого эффективный идентификатор пользователя совпадает с владельцем или создателем очереди, либо процесс с правами привилегированного пользователя.

Разделяемый ресурс IPC может существовать в системе без процессов, подключенных к этому разделяемому ресурсу.

buf - структура, описывающая управляющие параметры очереди. Тип msgid_ds описан в заголовочном файле <sys/message.h>, и представляет собой структуру, в полях которой хранятся права доступа к очереди, статистика обращений к очереди, ее размер и т.п. Данный вызов используется для получения или изменения процессом управляющих параметров, связанных с очередью и уничтожения очереди

Примеры.

Рассмотрим программу, состоящую из трех процессов: основной процесс, процесс A, процесс B.

1. Основной процесс считывает некоторую строку со стандартного устройства ввода. Если текстовая строка начивается с будквы A, основной процесс передает процессу A, аналогично для процесса B. Если текстовая строка начинается с буквы q, основной процесс передается ее процессу A и процессу B и завершается.

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/message.h>
#include <stdio.h>
struct // Определение буфера для взаимодействия
{
    long mtype; // тип
    char Data[256]; // строка } Message;

int main()
{
    key_t key;
    int msgid;
```


ФАКУЛЬТЕТ ВЫЧИСЛИТЕЛЬНОЙ МАТЕМАТИКИ И КИБЕРНЕТИКИ МГУ ИМЕНИ М.В. ЛОМОНОСОВА

```
    char str[256];
    key=ftok("/usr/mash 's');      // генерация ключа, однозначно определяющего
доступ к ресурсу данного типа
    msgid=msgget (key, 0666 | IPC_CREAT | IPC_EXCL);      // создаем очередь
сообщений , 0666 определяет права доступа, создание нового ресурса
    for(;;)     //запускаем бесконечный цикл
{
        gets(str);     // читаем из стандартного ввода строку
        strcpy(Message.Data, str); // и копируем содержимое строки в буфер
сообщения
        switch(str[0]) {   //разбираемся с первой буквой строки
        case 'a': case 'A':
          Message.mtype=1;   //устанавливаем тип,              msgsnd(msgid,
(struct msgbuf*) (&Message), strlen(str)+1, 0);   //посылаем сообщение в очередь
        break;           case 'b': case 'B':
          Message.mtype=2;
          msgsnd(msgid, (struct msgbuf*) (&Message), strlen(str)+1, 0);
        break;           case q': case 'Q':
          Message.mtype=1;
          msgsnd(msgid, (struct msgbuf*) (&Message), strlen(str)+1, 0);
          Message.mtype=2;
          msgsnd(msgid, (struct msgbuf*) (&Message), strlen(str)+1, 0);
          sleep(10); // нет уведомления о том, что два взаимодействующих
процесса прочли сообщения. Считаем, что если заснем на 10 секунд, процессы
успеют забрать сообщения о завершении (вообще это не очень правильный подход).
msgctl (msgid, IPC_RMID, NULL );   // удаляем этот ресурс
          exit(0);
        default:
          break;
        }
    }
}
```

2. Процесс-приемник А

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/message.h>
```


ФАКУЛЬТЕТ
ВЫЧИСЛИТЕЛЬНОЙ МАТЕМАТИКИ И
КИБЕРНЕТИКИ МГУ ИМЕНИ М.В.
ЛОМОНОСОВА

```
#include <stdio.h>
struct {
    long mtype;
    char Data[256];
} Message;

int main()
{
    key_t key;
    int msgid;
    key=ftok("/usr/mash 's');        // получаем ключ по тем же параметрам
    msgid=msgget(key, 0666 | IPC_CREAT); //создаем очередь сообщений
    for(;;) {        // запускаем вечный цикл
        msgrcv(msgid, (struct msgbuf*) (&Message), 256, 1, 0); //читаем из очереди
сообщений сообщение, которое связано с этим процессом
        printf("% s Message.Data); // выводим этот буфер на стандартное
устройство вывода
        if (Message.Data[0]='q' || Message.Data[0]='Q')        // если первый символ
равен q, то выходим из цикла и завершаем процесс. Если не q, то читаем дальше.
Если в очереди больше нет сообщений подходящего типа, процесс блокируется и
ждет
        break;
}
exit();
}
```

3. Процесс-приемник B Процесс B аналогичен процессу A.

Пример "Клиент-сервер"

Есть процесс-сервер,который будет обрабатывать запросы клиентов. Количество клиентов, взаимодействующих с сервером, может быть произвольно.

Процесс-сервер получает информацию от клиентов через очередь сообщений. Сообщения для сервера приходят с типом 1. Будем считать, что в теле сообщения находится PID процесса-клиента. Сервер может вернуть клиенту ответное сообщение, установив тип сообщения, равным PIDу клиента.

1. Сервер

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
#include <stdlib.h>
#include <string.h>
int main( int argc, char ** argv )
{ struct
    long mestype ;
    char mes [100] ;
  } messageto ;    // определили буфер для отправки сообщения
{ struct
    long mestype ;
    long mes;
  } messagefrom ; // определили буфер для приема сообщения
    key_t key;
    int msgid;
    key=ftok("example 't');       // сгенерировали ключ, считаем, что файл example
находится в текущем каталоге
    msgid=msgget(key, 0666 | IPC_CREAT | IPC_EXCL); //создаем очередь
сообщений
    while (1) {       // запускаем бесконечный цикл. Встали на прием сообщений.
Если в системе клиентов нет, сервер ждет. Как только приходит сообщение, сервер
сразу же начинает готовить сообщение для возврата
        msgrcv(mesid, messagefrom, sizeof (messagefrom) – sizeof (long), 1, 0);
//читаем из сквозной очереди сообщений сообщение, которое связано с этим
процессом
        messageto.mestype = messagefrom.mes;
// из буфера, в который попадает сообщение от клиена, берется соответствующее
поле и записываем его в поле типа сообщения, которое готовится к
отправке(определяем тип)
        strcpy(messageto.mes, "Message for client ");   // копируем определенную
строку в буфер, используемый для отправляемых сообщений
        msgsnd ( mesid, & messageto, sizeof (messageto) – sizeof (long), 0) ;
}
return 0;
}
```

2. Клиент

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
int main( int argc, char ** argv )
{
    struct {
        long mestype ;
        long mes ;
    } messageto ;
    struct {
    long mestype ;
    char mes[100];
  } messagefrom ;
    key_t key;
    int msgid;
    long pid=getpid();    //определили переменную pid, ее инициализировали
PIDом процесса, подключились к ресурсу
    key=ftok("example'r');
    msgid=msgget(key, 0666 );
    messageto.mestype = 1; // опеределяем тип сообщений (для сервера)
    messageto.mes = pid; // определяем в буфер PID
        msgsnd (mesid,& messageto, sizeof (messageto) – sizeof (long), 0) ;
        msgrcv(mesid, & messagefrom, sizeof (messagefrom) – sizeof (long), pid 0) ;
//ждем получение сообщения типа PID. При его получении выводим его.
        printf( " % s ", messagefrom.mes );
return 0;
}
```

Разделяемая память IPC

1. Создание общей памяти. Рассмотрим еще один разделяемый ресурс системы IPC.

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
```


ФАКУЛЬТЕТ ВЫЧИСЛИТЕЛЬНОЙ МАТЕМАТИКИ И КИБЕРНЕТИКИ МГУ ИМЕНИ М.В. ЛОМОНОСОВА

int shmget (key_t key, int size, int shmflg)

В системном вызове shmget передаются следующие параметры:

- key - ключ для доступа к разделяемой памяти;
- size задает размер области памяти, к которой процесс желает получить доступ.
- shmemflg определяет флаги, управляющие поведением вызова.

В случае успешного завершения вызов возвращает положительное число – дескриптор области памяти, в случае неудачи – -1.

2. Доступ к разделяемой памяти.

Чтобы открылся доступ к разделяемой памяти, нужно подключиться к ней и ассоциировать ее с некоторым "адресом"(указателем). Это можно сделать с помощью следующего вызова: #include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
char *shmat(int shmid, char *shmaddr, int shmflg)

Параметры:

- shmid – дескриптор области памяти
- *shmaddr – переменная, через которую указываем "адрес подключения". виртуальный адрес в адресном пространстве, начиная с которого необходимо подсоединить разделяемую память. Если задать значение >0, это значит, что хотим подключить разделяемую память к указанному в параметре адресу(можно попасть на уже существующий адрес).

  Если задать 0, то идет запрос на удобный для системы адрес.
- shmflg - комбинация флагов. В качестве значения этого аргумента может быть указан флаг SHM_RDONLY, который указывает на то, что подсоединяемая область будет использоваться только для чтения.

Системный вызов возвращает адрес, начиная с которого будет отображаться присоединяемая разделяемая память. В случае неудачи вызов возвращает -1.

Отключение от разделяемой памяти

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
int shmdt(char *shmaddr)
```

shmaddr - адрес прикрепленной к процессу памяти, который был получен при вызове shmat().

Данный вызов позволяет отсоединить разделяемую память, ранее присоединенную посредством вызова shmat()

В случае успешного выполнения функция возвращает 0, в случае неудачи – -1.

Управление разделяемой памятью

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
int shmctl(int shmid, int cmd, struct shmid_ds *buf)
```

Параметры:

- shmid - дескриптор области памяти, команда, которую необходимо выполнить, и структура, описывающая управляющие параметры области памяти
- cmd – команда. Возможные значения аргумента cmd:
  - IPC_STAT – скопировать структуру, описывающую управляющие параметры области памяти по адресу, указанному в параметре buf
  - IPC_SET – заменить структуру, описывающую управляющие параметры области памяти, на структуру, находящуюся по адресу, указанному в параметре buf.
  - IPC_RMID – удалить очередь
  - SHM_LOCK, SHM_UNLOCK – блокировать или разблокировать область памяти. Выполнить эту операцию может только процесс с правами привилегированного пользователя. Данный вызов используется для получения или изменения процессом управляющих параметров, связанных с областью разделяемой памяти, наложения и снятия блокировки на нее и ее уничтожения.

Пример. Работа с общей памятью в рамках одного процесса

ФАКУЛЬТЕТ
ВЫЧИСЛИТЕЛЬНОЙ МАТЕМАТИКИ И
КИБЕРНЕТИКИ МГУ ИМЕНИ М.В.
ЛОМОНОСОВА

```
int main(int argc, chr **argv)
{
    key_t key;
    char* shmaddr;
    key=ftok("/tmp/ter",'S');   /*генерация ключа*/
    shmid=shmget ( key, 100,0666|IPC_CREAT|IPC_EXCL );    /*создание области
разделяемой памяти*/
    shmaddr=shmat(shmid,NULL,0); /*подключение к памяти*/
    putm(shmaddr); /*работа с ресурсом*/
..........................................................
    shmctl(shmid,IPC_RMID,NULL); /* уничтожение ресурса */
    exit();
}
```

Массив семафоров

Можно создать ресурс, который представляется в виде совокупности семафоров, организованных в виде массив.

Каждый из семафоров массива может иметь текущее значение, с каждым из этих семафоров можно выполнять семафорные операции. Они являются псевдо атамарными.

Семафорные операции являются аналогами операций Дейкстры.

Семафорные операции над семафорами, составляющими массив семафоров, могут выполняться единовременно.

1. Создание/доступ к семафору

```
#include <sys/types.h>
#include<sys/ipc.h>
#include<sys/sem.h>
int semget (key_t key, int nsems, int semflag)
```

Параметры:

- key - ключ, уникальный идентификатор ресурса;
- nsems - количество семафоров (длина массива семафоров);
- semflag - флаги.

ФАКУЛЬТЕТ ВЫЧИСЛИТЕЛЬНОЙ МАТЕМАТИКИ И КИБЕРНЕТИКИ МГУ ИМЕНИ М.В. ЛОМОНОСОВА

Система создает массив из nsems семафоров и возвращает дескриптор этого ресурса. Через флаги можно определить права доступа и те операции, которые должны выполняться (открытие семафора, проверка, и т.д.). Функция semget() возвращает целочисленный идентификатор созданного разделяемого ресурса, либо -1, если ресурс не удалось создать.

2. Операции над семафором

```
#include <sys/types.h>
#include<sys/ipc.h>
#include<sys/sem.h>
int semop (int semid, struct sembuf *cmd_buf, size_t nops)
```

Параметры системного вызова semop:

- semid – дескриптор ресурса;
- *cmd_buf – указатель на массив из элементов sembuf, в sembuf задается операция над конкретным семафором из массива;
- nops – количество элементов в массиве из операций cmd_buf.

Т.е. возможно передать в систему как запрос на выполнение одной операции над одним семафором, так и запрос на выполнений группы операций.

Рассмотрим структуру буфера семафора.

```
struct sembuf
{    short sem_num;   /*номер семафора в массиве семафоров*/
     short sem_op;   /* операция*/
     short sem_flg;   /*флаги операции*/
}
```

Предположим, что у семафора в текущий момент времени находится в переменной val. Тогда:

- Если sem_op$\neq$ 0 то процесс будет блокирован до тех пор, пока (val+sem_op <0). Т.е. пока значение семафора не изменится каким-то другим процессом.

  val=val+sem_op

- Если sem_op=0 то эта операция блокирует семафор до тех пор, пока его не значение не равно нулю (val $\neq$0). Т.е. операция 0 – ожидание обнуления семафора. Другие процессы могут обнулить значение семафора.

3. Управление массивом семафоров

```
#include <sys/types.h>
#include<sys/ipc.h>
#include<sys/sem.h>
int semetl (int semid, int num, int cmd, union semun arg)
```

Параметры:

- semid – дескриптор массива семфоров;
- num – номер семафора в массиве;
- cmd – операция. Команды:
  - IPC_SET – установить значение семафора
  - IPC_RMID – удалить массив семафоров.
- arg – управляющие параметры

Возвращает значение, соответствовавшее выполнявшейся операции (по умолчанию 0), а в случае неудачи – -1.

ФАКУЛЬТЕТ ВЫЧИСЛИТЕЛЬНОЙ МАТЕМАТИКИ И КИБЕРНЕТИКИ МГУ ИМЕНИ М.В. ЛОМОНОСОВА

Лекция 18. Работа с разделяемой памятью с синхронизацией семафорами

Работа с разделяемой памятью с синхронизацией семафорами

Рассмотрим пример, демонстрирующий возможности синхронизации доступа к разделяемой памяти с использованием семафоров IPC.

Предположим, что первый процесс создает ресурс "разделяемая память"и ресурс "массив семафоров состоящий из одного семафора. Этот процесс будет читать со стандартного устройства ввода некоторую строку и записывать ее в разделяемую память. Второй процесс эту строку из разделяемой памяти будет считывать.

Необходима синхронизация процессов, чтобы второй процесс не начал читать строку до того, как первый процесс закончит писать строку.

Для синхронизации используем массив семафоров из одного семафора. **Первый процесс:**

```
#include <stdio.h> #include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
#include <string.h>
#define NMAX 256

int main ( int arge, char** argv)
{
    key_t key;
    int semid, shmid;
    struct sembuf sops;
    char *shmaddr;
    char str[ NMAX ];
    key = ftok("/usr/ter/exmpl",'S'); // создаем уникальный ключ
    semid = semget(key,1,0666 | IPC_CREAT | IPC_EXCL ); // создаем массив
семафоров из одиного семафор с определенными правами доступа
    shmid = shmget (key, NMAX, 0666 | IPC_CREAT | IPC_EXCL ); /*создаем
разделяемую память на 256 элементов */
    shmaddr = shmat (shmid, NULL, 0);   // подключаемся к разделяемой памяти,
в shaddr - указатель на буфер с разделяемой памятью
    semctl(semid,0, SETVAL, (int) 0);    //инициализируем значение семафора
```

sops.sem_num = 0;   //устанавливаем неизменные параметры.нулевой номер семафора

sops.sem_flg = 0;

do   // запуск бесконечного цикла

{

printf(“Введите с троку:”);

if (fgets (str, NMAX,stdin) == NULL)   strcpy (str, "Q ");   // если со стандартного устройства ввода получаем пустую строку, подставляем Q в нулевой элемент. Пустой ввод – признак завершения.

strcpy (shmaddr, str); // скопировали считанную строчку в разделяемую память

sops.sem_op=3 ;   // устанавливаем поле операции семафора =3

semop (semid, & sops, 1);   // установили значение семафора =3

sops.sem_op=0;   // установление кода операции 0.

semop(semid, & sops, 1);   // ждем пока обнулится значение семафора (идем во второй процесс)   }

while (str[0]!=’Q’) ; // если выходим, отключаемся от ресурса, уничтожаем ресурсы, завершаемся

shmidt (shmaddr);

shmctl (shmid, IPC_RMID, NULL) ;

semctl (semid, 0,IPC_RMID, (int) 0) ;

return 0;

}

Второй процесс: /* здесь нам надо корректно определить существование ресурса, если он есть - подключиться, если нет - сделать что-то еще, но как раз э то го мы делать не будем */ #include <stdio.h>

#include <sys/types.h>

#include <sys/ipc.h>

#include <sys/sem.h>

#include <string.h>

#define NMAX 256

int main (int arge, char** argv)

{

key_t key;

int semid, shmid ;

```
    struct sembuf sops;
    char *shmaddr;
    char str [NMAX];
    ket= ftok ("/usr/ter/exmpl",'S'); // // далее аналогично предыдущему процессу
- инициализации ресурсов
    semid = semget( key, 1, 0666 | IPC_CREAT | IPC_EXCL );
    shmid = shmget( key, NMAX, 0666 | IPC_CREAT | IPC_EXCL );
    shmaddr = shmat(shmid, NULL, 0);
    sops.sem_num = 0;
    sops.sem_flg = 0;
// запускаем цикл. Цикл до тех пор, пока не получим Q
    do {
        printf ("Ждем открытия семафора \ n"); // ожидание положительного
значения семафора        sops.sem_op=-2;   // устанавливаем код операции -2
        semop (semid, & sops, 1);    //выполняем с этим кодом системный вызов
semop
// будем ожидать, пока "значение семафора"+"значение sem_op" не перевалит за 0,
то есть если придет "3", то "3-2=1" . Так как значение уже 3, значит информация
уже записана первым процессом // теперь значение семафора равно 1
strcpy ( str , shmaddr ); //считываем информацию и анализируем
        if ( str[0] == "Q")
          shmdt ( shmaddr );
        sops.sem_op=-1;
        semop (semid, & sops, 1); //выполняем код операции -1, обнуляем семафор
и первый процесс разблокируется
        printf ( "Read from shared memory: % s \ n str ) ;
    } while ( str[0]!='Q' ) ;
    return0;
}
```

Взаимодействие процессов посредством сокетов

Ранее были рассмотрены средства взаимодействия процессов, которые работают в рамках одной вычислительной системы. Аппарат сокетов позволяет организовывать взаимодействие процессов, находящихся в сети.

Появился в системе UNIX BSD 4.2. Затем распространился как стандарт на межпроцессное взаимодействие в рамках компьютерных сетей.

Определение. *Сокет* – ресурс, который позволяет организовывать взаимодействие между процессами.

1. Создание сокета

```
#include <sys/types.h>
#include <sys/socket.h>
int socket (int domain, int type, int protocol);
```

- domain – константа, определяющая коммуникационный домен. Локализация процессов определяется *коммуникационным доменом* . Он определяет то, какие процессы могут организовывать взаимодействие:
  - AF_UNIX – коммуникационный домен, определяющий локальное взаимодействие (например, в рамках UNIX-системы). Если взаимодействуют процессы, находящиеся в рамках одной вычислительной системы.
  - AF_INET – коммуникационный домен, определяющий возможность взаимодействия в рамках сети.

  Т.е. можно создать сокет, который будет позволять организовывать работу как коммуникационный домен одного из типов. Таких коммуникационных доменов может быть много.

- type – тип сокета:
  - SOCK_STREAM – виртуальный канал (надежное взаимодействие)
  - SOCK_DGRAM – датаграммное взаимодействие без установления канала. Более эффективное, но не обеспечивает гарантированной доставки пакета.

- protocol – тип протокола. Эта характеристика определяет выбор сетевого протокола, который будет использоваться про организации взаимодействия через сокет:
  - 0 – автоматический выбор системой наиболее подходящего протокола
  - IPPROTO_TCP используют, если тип сокета -– виртуальный канал,
  - IPPROTO_UDP – если датаграммный сокет

После передачи этих трех характеристик в системном вызове socket получаем дескриптор созданного сокета.

2. Связывание сокета Чтобы организовывать взаимодействие между процессами, использующими сокеты, создавшие процесс сокеты можно именовать. Система

предполагает использование системного вызова bind, предназначенного для именования сокетов.

```
#include <sys/types.h>
#include <sys/socket.h>
int bind(int sockfd, struct sockaddr * myaddr, int sizeaddr);
```

Параметры:

- sockfd – дескриптор сокета, который вернула функция socket
- myaddr – указатель на структуру, содержащую адрес, который хотим поставить в соответствие сокеты
- sizeaddr – размер той структуры в байтах, на которую указывает второй аргумент (myaddr)

Имя сокета или адрес сокета зависят от коммуникационного домена. Если хотим работать коммуникационном домене UNIX, то используем в качестве имени полное имя некоторого файла(похоже на именованный канал). Если работать коммуникационном домене INET, имя стоится из IP адреса компьютера, на котором работаем процесс и номера порта.

В случае успешного связывания bind возвращает 0, а в случае ошибки – -1 с кодом причины в erno.

Предварительное установление соединения

Есть несколько моделей использования сокетов. Рассмотрим модель с *предварительным установлением соединения.* Тип сокета – виртуальный канал или датаграмма.

В модели клиент-сервер два процесса : процесс-сервер и процесс-клиент.

Процесс-сервер не знает, какие клиенты к нему обращаются. Клиенты точно знают свой сервер → сервер должен быть именован.

Сервер

- Создаем сокет.
- Ассоциируем с сокетом некоторое имя, через которое клиенты смогут обращаться.
- Если connect пришел до того, как сервер обратился к системному вызову listen, клиент получит ошибку.
- Системный вызов listen устанавливает характеристики очереди ожидания.
- В процессе-сервере соединение подтверждается через системный вызов accept. Подтверждение соединения – процесс создания нового сокета, который создается и связывается с клиентом.

  После установления этого системного вызова, connect от клиента смогут проходить. Организовывается синхронная передача данных.

  Если вся очередь занята, поступает отказ от прихода пакета. Если в listen есть свободное место, устанавливается соединение.

Примечание. Создавать все сокеты в рамках одного серверного процесса неправильно, потому что забьется таблица открытых файлов в процессе, что вызовет отказы. Надо сформировать сыновние процессы, каждый из которых будет связываться со своим клиентом.

Клиент

- Создаем свой сокет, через который будет осуществляться взаимодействие с сервером.
- Можно не связывать данный сокет с именем, так как модель клиент-сервер организована на ответе сервера на запрос клиента.
- Обращение к системному вызову connect(). Это системный вызов, в котором указывается имя сокета сервера и запрашивается возможность установки соединения.

ФАКУЛЬТЕТ ВЫЧИСЛИТЕЛЬНОЙ МАТЕМАТИКИ И КИБЕРНЕТИКИ МГУ ИМЕНИ М.В. ЛОМОНОСОВА

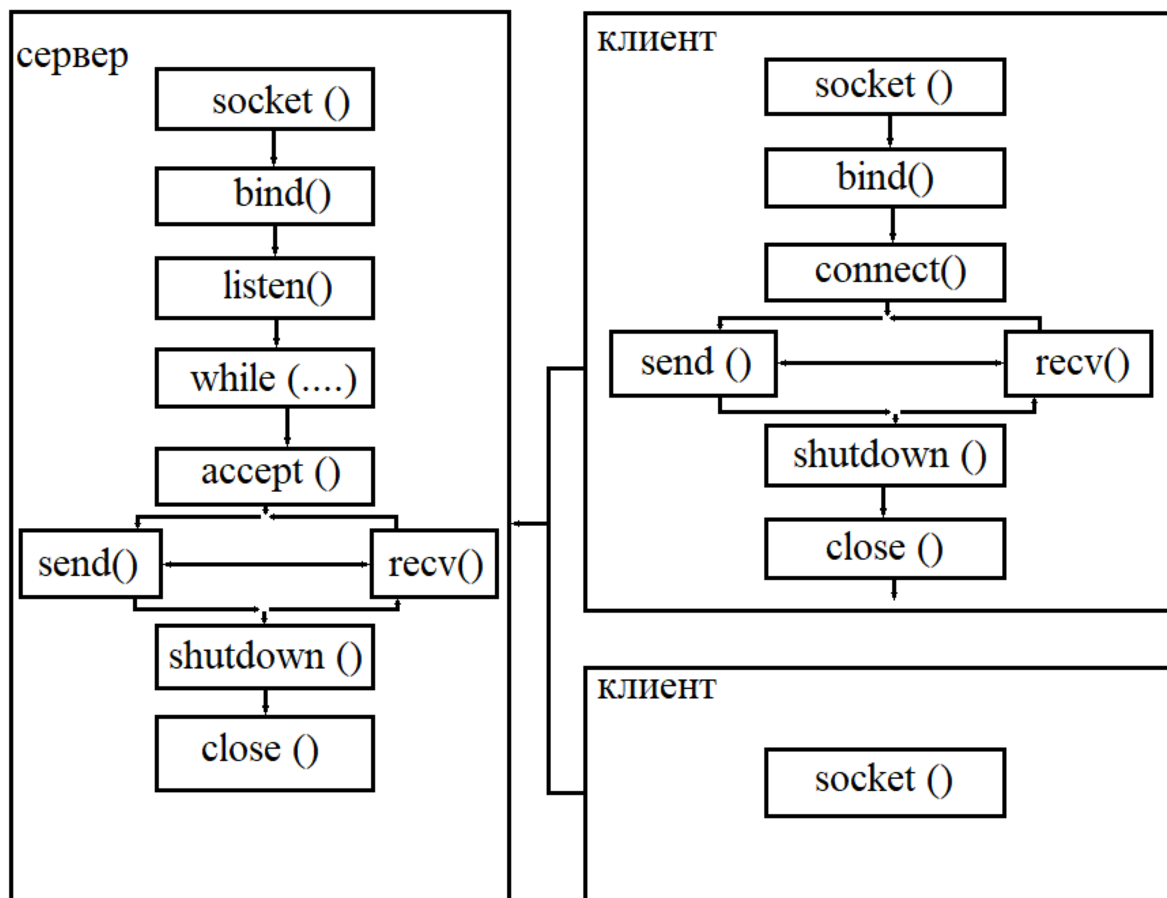

Рис. 18.1. Сокеты с установлением предварительного соединения

Сокеты без предварительного соединения

Тип сокета – детаграмма. В этой модели могут взаимодействовать произвольные процессы.

Рассмотрим схему взаимодействия.

- В процессе создается сокет.
- Этот сокет именуется.
- Можно использовать именованную отправку send/receive, в параметрах которой указывается имя сокета-получателя и имя сокета-отправителя.

Итог: Две модели использования сокетов: сокеты с предварительным установлением соединения, когда устанавливается виртуальный канал, через который осуществляется обмен информацией и сокеты без предварительного установления соединения.

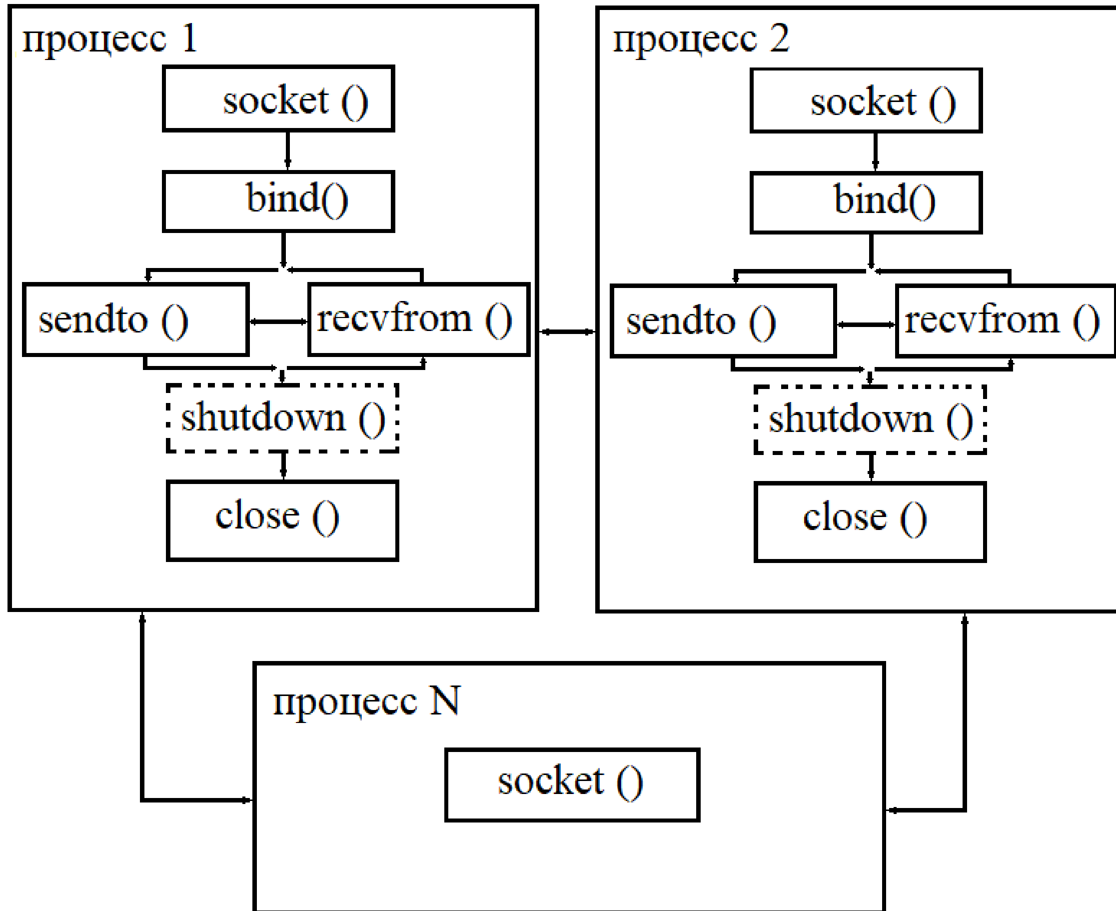

Рис. 18.2. Сокеты без установления предварительного соединения

Управление оперативной памятью

Это одна из важнейших функций операционной системы, которая обеспечивает распределение и контроль за использованием ресурса вычислительной системы, который называется оперативная памяти. В эту задачу входит распределение физической памяти между процессами, контроль за использованием памяти и реализация той или иной модели виртуальной памяти. Эта часть ОС решает целый спектр задач.

Основные задачи:

1. Контроль состояния каждой единицы памяти (система должна отслеживать, какая единица свободна/распределена, обеспечение аппаратурой компьютера и ОС - таблицы).

2. Стратегия распределения памяти (выбор правил, по которым принимаются решения, кому, когда и сколько памяти должно быть выделено).

ФАКУЛЬТЕТ
ВЫЧИСЛИТЕЛЬНОЙ МАТЕМАТИКИ И
КИБЕРНЕТИКИ МГУ ИМЕНИ М.В.
ЛОМОНОСОВА

3. Выделение памяти (выбор конкретной области, которая должна быть выделена).

4. Стратегия освобождения памяти (процесс освобождает, ОС "забирает" окончательно или временно. Здесь же выбор стратегии). Вся память всегда занята, поэтому здесь принимается решение, у кого забрать память, чтобы ее выделить другому процессу.

Функция ОС по управлению оперативной памятью состоит из двух составляющих: аппаратная составляющая (те аппаратные интерфейсы, которые позволяют осуществлять управление оперативной памятью) и программная реализация.

Далее рассмотрим стратегии и методы управления.

1. Одиночное непрерывное распределение.
2. Распределение разделами.
3. Распределение перемещаемыми разделами.
4. Страничное распределение.
5. Сегментное распределение.
6. Сегментно-страничное распределение.

Рассмотрим стратегии управления по следующему плану:

1. Основные концепции.
2. Необходимые аппаратные средства.
3. Основные алгоритмы.
4. Достоинства, недостатки.

Одиночное непрерывное распределение

ФАКУЛЬТЕТ ВЫЧИСЛИТЕЛЬНОЙ МАТЕМАТИКИ И КИБЕРНЕТИКИ МГУ ИМЕНИ М.В. ЛОМОНОСОВА

Предполагается, что имеется адресное пространство физической оперативной памяти. Часть этого адресного пространства выделена под ОС, часть может использоваться для выполнения одного процесса.

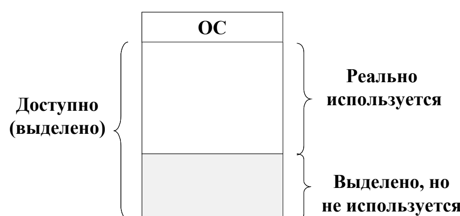

Рис. 18.3. Организация адресного пространства

Необходимые аппаратные средства:

- Регистр границ + режим ОС / режим пользователя. Тот регистр, в который устанавливается адрес границы ОС. По значению этого регистра все то, что больше – адресное пространство процесса, все то, что меньше – адресное пространство ОС.
- Если ЦП в режиме пользователя попытается обратиться в область ОС, то возникает прерывание.

Недостатки:

1. Размер обрабатываемого процесса ограничен размерами доступной для процесса области.
2. Часть памяти не используется
3. Процессом/заданием память занимается все время выполнения.

Не предполагается мультипроцессирвание.

Распределение неперемещаемыми разделами

Есть выделенная область, в которой находится ОС. Осальное пространство разделено НА неперемещаемые разделами. Каждый раздел предопределенного фиксированного размера. В каждый раздел может быть загружен один процесс. Эта модель предусматривает мультипрограммную/мультипроцессную обработку.

ФАКУЛЬТЕТ ВЫЧИСЛИТЕЛЬНОЙ МАТЕМАТИКИ И КИБЕРНЕТИКИ МГУ ИМЕНИ М.В. ЛОМОНОСОВА

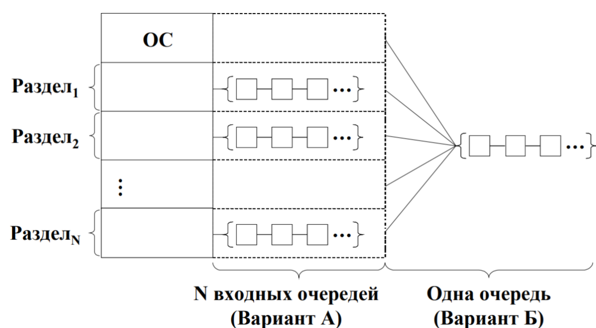

Рис. 18.4. Организация адресного пространства для модели с неперемещаемыми разделами

Необходимые аппаратные средства:

- Два регистра границ, которые в каждый момент времени будут содержать адрес начала и адрес конца того раздела, в котором находится исполняемый процесс.
- Ключи защиты (PSW)

Алгоритмы.

Модель статического определения разделов.

А. Для каждого раздела формируется своя очередь процессов. ОС оценивает необходимую для процесса оперативную память и соразмерно этой оценки помещает это процесс в определенную очередь. Каждая очередь связана со своим разделом. Процесс размещается в разделе минимального размера, достаточного для размещения данного процесса. В случае отсутствия процессов в каких-то подочередях – неравномерная загрузка разделов, неэффективность использования памяти.

Б. Одна входная очередь процессов.

1. Освобождение раздела $\Rightarrow$ поиск (в начале очереди) первого процесса, который может разместиться в разделе.
Проблема: несоответствие размеров, большие разделы $\leftrightarrow$ маленькие процессы.
2. Освобождение раздела $\Rightarrow$ поиск процесса максимального размера, который поместится в раздел.
Проблема: дискриминация “маленьких” процессов.
3. Оптимизация варианта 2. Организация очереди с характеристикой "счетчик дискриминации". Каждый процесс имеет счетчик дискриминации. Если значение счетчика процесса $\geqslant$ K, то обход его в очереди невозможен. Ищем баланс между "справедливостью"и эффективностью.

ФАКУЛЬТЕТ ВЫЧИСЛИТЕЛЬНОЙ МАТЕМАТИКИ И КИБЕРНЕТИКИ МГУ ИМЕНИ М.В. ЛОМОНОСОВА

Достоинства:

1. Простые средства аппаратной поддержки. Два регистра границ, которые переустанавливаются при смене процессов.
2. Простое средство организации мультипрограммирования.
3. Простые алгоритмы.

Недостатки:

1. Фрагментация. Проблема несоответствия размера раздела размеру процесса.
2. Ограничение размерами физической памяти.
3. Весь процесс размещается в памяти – возможно неэффективное использование.

Распределение перемещаемыми разделами

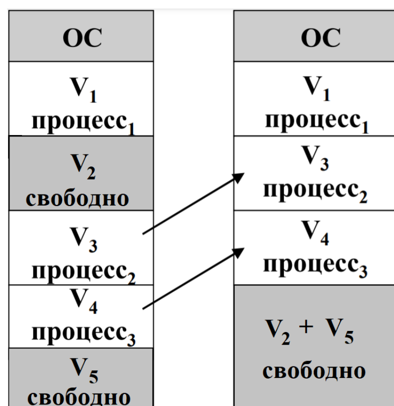

Рис. 18.5. Организация виртуальной памяти

Система загружается процессами. Разделы, которые заняты процессами, периодически заканчиваются и освобождаются. Сдвигая процессы, хотим высвобождать один свободный независимый фрагмент. Решение – использование системы адресации с базированием.
Внутри процесса используются относительные адреса (относительно начала процесса).

Необходимые аппаратные средства:

1. Регистры границ + регистр базы
2. Ключи + регистр базы

Алгоритмы: Аналогично предыдущему
Достоинства: Ликвидация фрагментации
Недостатки: Ограничение размером физической памяти и затраты на перекомпоновку

Лекция 19. Управление оперативной памятью

Обзор прошлой лекции

Функция управления оперативной памятью во многом аппаратно-программная. Ее функции, возможности системы, компоненты ОС, которые ее реализуют, во многом зависят от аппаратной поддержки.

Модели управления оперативной памятью рассматриваются по следующему плану:

1. Основные концепции.

2. Необходимые аппаратные средства.

3. Основные алгоритмы.

4. Достоинства, недостатки.

Одиночное непрерывное распределение – простейшая модель. Все адресное пространство оперативной памяти разделяется на две области: область, выделенная под программы ОС и область для загрузки и исполнения оного процесса.

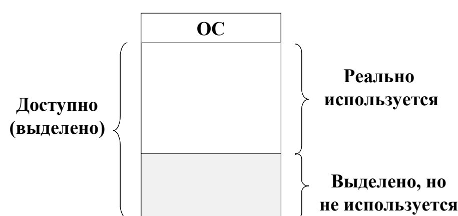

Рис. 19.1. Организация памяти

Для реализации такой модели необходила возможность установки регистра границ. Если ЦП в режиме пользователя попытается обратиться в область ОС, возникает прерывание.

Распределение неперемещаемыми разделами. Уже возможна обработка группы пользовательских процессов. Выделяется адресное пространство под ОС. Оставшееся пространство разделяется на фиксированное количество разделов некоторого предопределенного раздела. В каждом из разделов может быть загружен свой процесс. Эти процессы могут обрабатываться в режиме мультипрограммирования. Необходимы средства защиты, чтобы не было недетерминированного результата.

ФАКУЛЬТЕТ
ВЫЧИСЛИТЕЛЬНОЙ МАТЕМАТИКИ И
КИБЕРНЕТИКИ МГУ ИМЕНИ М.В.
ЛОМОНОСОВА

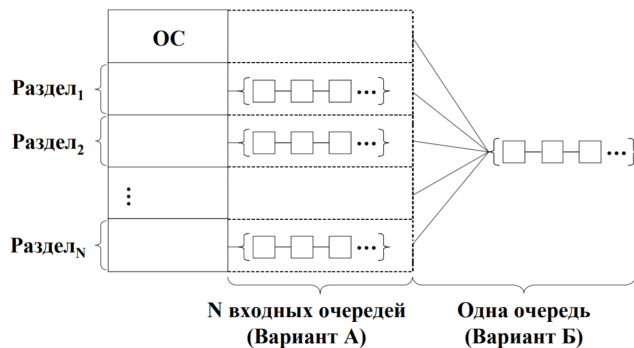

Рис. 19.2. Организация памяти

Аппаратные средства:

- Два регистра границ и возможность процессора работать как в режиме ОС, так и в пользовательском режиме.
- Если исполнительный адрес не попадает в диапазон значений, который находится в регистрах границ, происходит прерывание.

Алгоритмы:

Модель статического определения разделов.

[A] Для каждого раздела формируется своя очередь процессов, которые ожидают обработку. Может произойти ситуация, когда для определенных разделов очередь существенная, а для других разделов очередь уже завершилась.

[B] Одна входная очередь процессов.

Рассмотрим варианты действий в ответ на освобождение раздела:

1. поиск первого процесса, который помещается в освободившийся процесс. $\Rightarrow$ Возможна ситуация выбора маленького процесса для большого раздела
2. поиск процесса максимального размера, который поместится в освободившийся процесс. $\Rightarrow$ Дискриминация маленьких процессов.
3. Оптимизация варианта 2. Для каждого процесса при его размещении в очереди определим некоторую константу – предельное значение количества дискримимнаций. При каждой дискриминации значение уменьшается на 1. При значении 0 этот процесс будет размещен в раздел при любых обстоятельствах.

ФАКУЛЬТЕТ ВЫЧИСЛИТЕЛЬНОЙ МАТЕМАТИКИ И КИБЕРНЕТИКИ МГУ ИМЕНИ М.В. ЛОМОНОСОВА

Достоинства:

- модель позволяет организовывать мультипрограммирование
- простые средстсва аппаратной поддержки
- простые алгоритмы

Недостатки:

- обрабатываемый процесс ограничен размером доступной оперативной памяти
- фрагментация, образуются свободные неиспользуемые куски памяти
- весь процесс весь период обработки находится в оперативной памяти

Распределение перемешаемыми разделами.

Появляется модель виртуальной памяти – модель управления памятью, которая потенциально позволяет переносить загрузку процесса из одной области в другую. одним из решений проблемы перемещаемости является аппарат виртуальной памяти и аппарат базирования адресов. Внутри программы используются не абсолютные адреса, а относительные адреса (смещение от некоторых точек, которые помещаются в специальные базовые регистры), в данном случае смещение относительно начала загрузки.

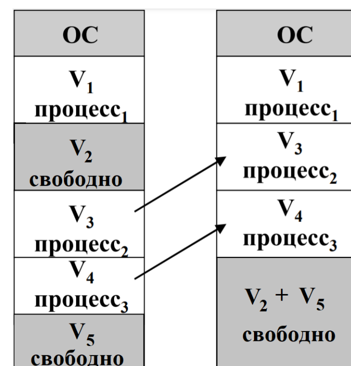

Рис. 19.3. Организация памяти

Процессы загружаются в оперативную память, они попеременно выполняются заканчиваются. При загрузке новых процессов образуются фрагменты. Когда ОС понимает, что система деградирует из-за фрагментации, приостанавливается выполнение пользовательских процессов, они перемещаются по оперативной памяти с целью объединения фрагментов в один общий.

Необходимые аппаратные средства: регистры границ, которые обеспечивают защиту памяти, и регистр базы.

Достоинства: мультипроцессная обработка и борьба с фрагментацией.

Недостатки: не подходит для мультипроцессного режима с интерактивными процессами, т.е. ограничение на пакетные системы; большие накладные расходы на перекомпоновку; процесс ограничен размером памяти; обрабатываемый процесс ведь находится в оперативной памяти.

Страничное распределение

Примечание. Виртуальное адресное пространство – модель адресации, которая используется внутри каждого из процессов. Размер виртуального адресного пространства определяется разрядностью процессора.

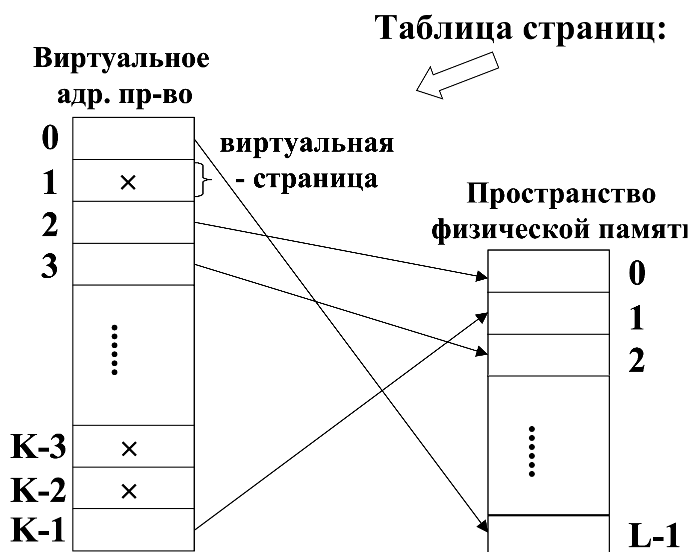

Рис. 19.4. Таблица страниц

И виртуальное адресное пространство, и пространство физической памяти разделены на страницы фиксированного размера. Размер – степень 2. Страничная организация памяти позволяет проводить отображение виртуальных адресов в физические адреса. Виртуальный адрес представляется как номер виртуальной страницы и смещение относительно начала этой виртуальной страницы, а физический адрес представляется как номер физической страницы и смещение относительно начала этой страницы.

Отображение – преобразование номера виртуальной страницы в номер физической страницы, в которой размещена соответствующая виртуальная страница.

Для этого отображения используется таблица страниц. Размер таблица страниц – количество виртуальных страниц в виртуальном адресном пространстве. i-я строчка таблицы страниц соответствует информации об i-й виртуальной страницы выполняемого в данный момент процесса. Содержимым этой строки является либо номер физической страницы, в которой размещена i-я виртуальная страница, либо информация о том, что такой страницы в памяти нет.

Проблемы: таблица страниц огромная, при смене процессов необходимо менять таблицу страниц. Таблицу процесса, который исполнялся ранее, необходимо сохранить.

Необходимые аппаратные средства:

ФАКУЛЬТЕТ ВЫЧИСЛИТЕЛЬНОЙ МАТЕМАТИКИ И КИБЕРНЕТИКИ МГУ ИМЕНИ М.В. ЛОМОНОСОВА

1. Полностью аппаратная таблица страниц (стоимость, полная перегрузка при смене контекстов, скорость преобразования). Нереализуемый подход.

2. Таблица страниц в ОЗУ + Регистр начала таблицы страниц в памяти (простота, управление смены контекстов, медленное преобразование).

   Размещение таблицы в оперативной памяти. Можно предусмотреть в процессоре специальный регистр, который в каждый момент времени будет содержать адрес начала таблицы страниц исполняемого процесса. При смене процессов не надо будет переписывать таблицы, а просто переустанавливать значение этого регистра.

3. Гибридное решение. Комбинация использования аппаратных средств хранения информации таблицы страниц с программными алгоритмами.

Буфер быстрого преобразования адресов (TLB)

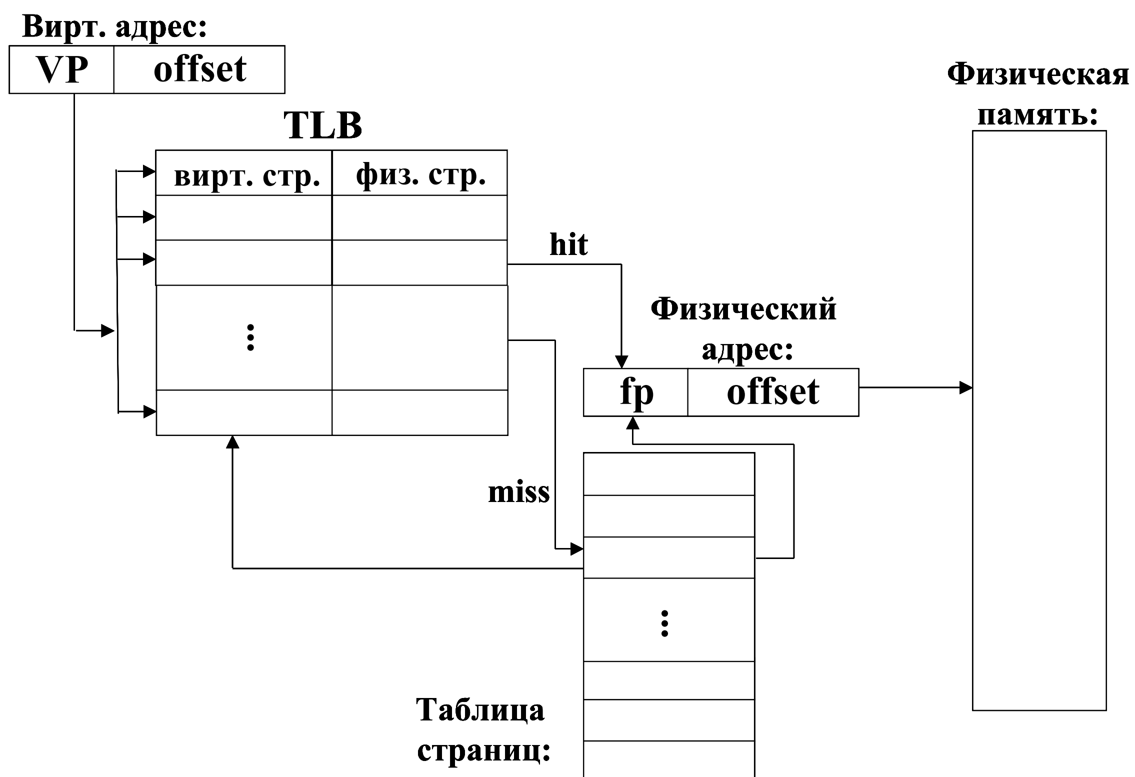

Рис. 19.5. TLB

Размер этой таблицы может быть 8-64 строки, т.е. ее размер аппаратно реализуем внутри процессора.

ФАКУЛЬТЕТ ВЫЧИСЛИТЕЛЬНОЙ МАТЕМАТИКИ И КИБЕРНЕТИКИ МГУ ИМЕНИ М.В. ЛОМОНОСОВА

Структура TLB таблицы. Каждая строка содержит два значения: номер виртуальной и физической страниц.

Виртуальный *исполнительный адрес* – это страничный адрес, в котором некоторое количество разрядов выделено под номер виртуальной страницы, другие разряды выделены под смещение внутри страницы.

Устройство управления процессора вытаскивает из виртуального значение поля, в котором находится номер виртуальной страницы. В процессоре имеется аппаратный блок сопоставления полученного номера виртуальной страницы с содержимым таблицы TLB.

Если такой поиск завершается успешно, это означает, что в таблице есть строка, в которой находится номер физической страницы для искомой виртуальной страницы. Тогда номер виртуальной страницы заменяется на номер физической страницы из строки TLB. Т.е. преобразование произошло аппаратно.

Если же в таблице нет строки, в которой первое поле совпадает со значением номера виртуальной страницы исполнительного адреса, происходит обращение к таблице страниц обрабатываемого процесса, которая находится в оперативной памяти. Далее обновляется таблица TLB (одна строчка выкидывается и туда ставится новая пара: номер виртуальной страницы, который взяли из исполнительного виртуального адреса, и номер физической страницы, который взяли из таблицы страниц).

Выбор строчки TLB, которая будет стерта может быть случайным, либо наименее интенсивно используемую (если в таблице есть колонка с характеристикой интенсивности использования строк)

Итог. Данная аппаратно-программная схема позволяет, используя TLB как кэш, который позволяет минимизировать реальные обращения к неэффективному ресурсу(оперативной памяти), повысить эффективность. При смене процессов все еще просиходит долгая замена таблицы.

Иерархическая организация таблицы страниц

Проблема – размер таблицы страниц.

Одним из эффективных решений представления таблиц в памяти являются многоуровневые таблицы страниц.

Двухуровневая организация. Виртуальный исполнительный адрес состоит из виртуальной страницы(VP) и смещения по этой странице.

Двухуровневая организация позволяет разделить поле "номер виртуальной страницы"на два подполя.

Инофрмация таблицы страниц представляется следующим образом: есть внешняя

ФАКУЛЬТЕТ ВЫЧИСЛИТЕЛЬНОЙ МАТЕМАТИКИ И КИБЕРНЕТИКИ МГУ ИМЕНИ М.В. ЛОМОНОСОВА

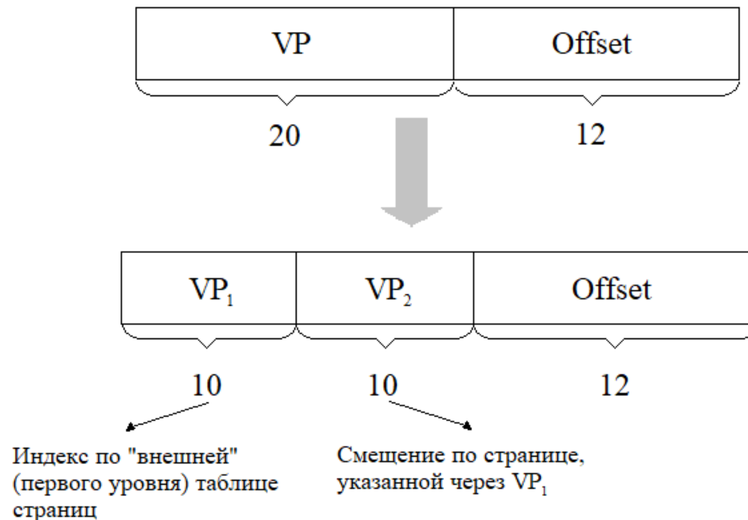

Рис. 19.6. Двухуровневая организация

таблица страниц, размер которой 2^10 и таблицы второго уровня (их может быть 2^10 штуки).

При преобразовании виртуального адреса в физический, считаем, что внешняя таблица страниц вся находится в оперативной памяти. Индексируемся по внешней таблице страниц, по значению первого разделенного поля. В этой строчке либо информация о том, что таблицы второго уровня для данного адреса в памяти нет(ее надо загрузить), либо записан адрес начала соответствующей таблицы страниц второго уровня. По этой таблице индексируемся по значению второго разделенного поля. В этой строчке получим информацию о физической странице. Преобразование виртуальный страничный адрес в физический страничный адрес.

$\Rightarrow$ В памяти всегда находится таблица первого уровня. Есть множество таблиц второго уровня, большинство которых размещено не в оперативной памяти. Т.е. нет необходимости держать в памяти всю таблицу. За счет локализации, которая имеется при формировании потока запросов за окмандами и операндами, перезагрузки будут происходить не так часто.

$\Rightarrow$ При смене процессов уже не надо перемещать таблицу из миллиона строк. Надо переместить только внешнюю таблицу из 2^10 строк. Механизм второго уровня – меканизм КЭШа, модель применения алгоритмов кэширования.

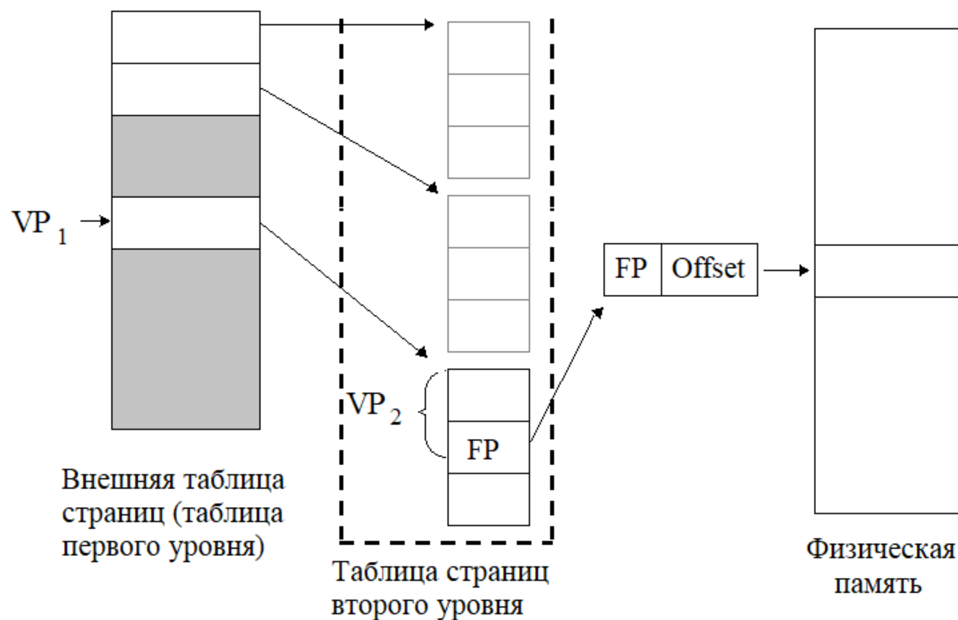

Рис. 19.7. Двухуровневая организация

Лекция 20. Страничное распределение памяти

Использование хэш-таблицы

Следующая модель организации – использование хэширования. Модель основывается на использовании функции расстановки (хэш-функции). Функция расстановки отображает аргумент в некоторое ограниченное множество значений. Значения, полученные в результате обращения к функции расстановки, фактически определяют номер строки хэш-таблицы. Хэш-таблица – таблица ограниченного размера, её размер накладывает ограничение на функцию расстановки. Чем меньше размер хэш-таблицы, тем более вероятна ситуация коллизии – для двух разных аргументов получаем одно и то же значение функции расстановки. В этом случае обычно используется список, сформированный из пар виртуальная страница-физическая страница. В этом списке функция расстановки даёт для всех значений виртуальных страниц одно и то же значение. То есть, при преобразовании мы берем некоторый номер виртуальной страницы, получаем номер записи в хэш-таблице, если при этом наблюдается коллизия, проходим по списку с раскрытием совпадений. Такая модель может быть весьма эффективной. Её качество зависит от размера хэш-таблицы.

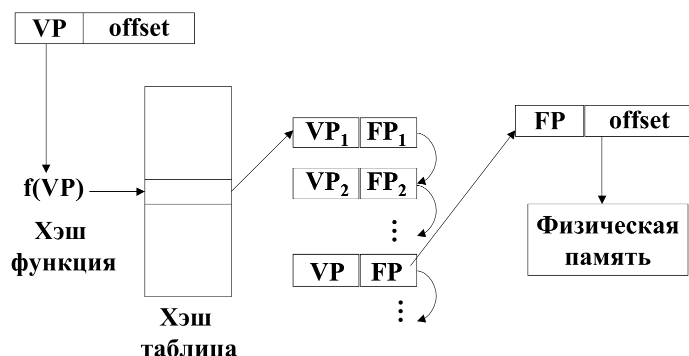

Рис. 20.1. Использование хэш-функциии

Инвертированные таблицы страниц

В исполнительный виртуальный адрес добавляется ещё одно поле – идентификатор процесса (PID). Теперь исполнительный виртуальный адрес представляет собой тройку – PID, номер виртуальной страницы и смещение по странице.

Инвертированная таблица страниц устроена следующим образом. Каждая запись таблицы страниц содержит пару значений – PID и номер виртуальной страницы. Номер записи есть номер физической страницы, в которой находится соответствующая виртуальная страница процесса. Соответственно, преобразование виртуального адреса в физический происходит следующим образом. Получаем исполнительный виртуальный адрес, вырезаем из него пару – PID-номер виртуальной страницы, далее ищем по таблице страниц строку, в которой будут совпадать PID и номер виртуальной страницы. Если такая строка найдена, то номер данной строки есть номер физической страницы, в которой находится виртуальная страницы данного процесса.
i

Такая организация дает возможность существования единой таблицы. То есть, у нас нет таблиц страниц процессов, мы имеем единую инвертированную таблицу, которая по сути даёт информацию о том, в какой физической странице оперативной памяти находится такая виртуальная страница некоторого процесса. Таким образом, мы избавились от проблемы смены таблиц страниц при смене процессов. Остаётся другая проблема – проблема поиска. Прямой поиск является довольно затратной операцией.

Размер инвертированной таблицы страниц равен количеству физических страниц, которые подключены к компьютеру.

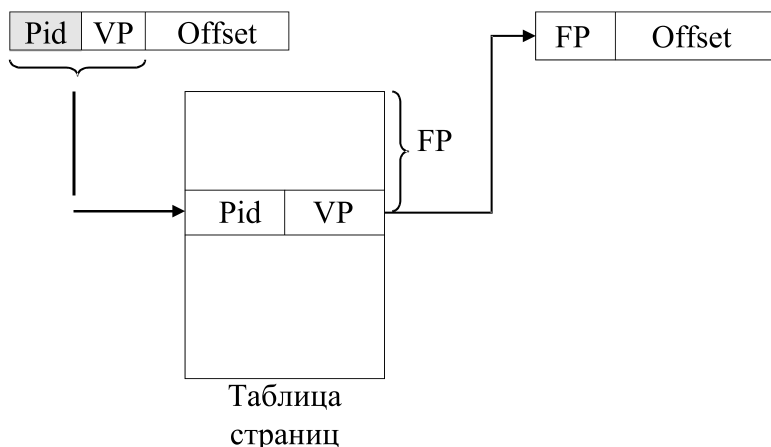

Рис. 20.2. Инвертированные таблицы страниц

subsectionАлгоритмы замещения страниц Рассмотрим проблему замещения страниц. Пусть нужно загрузить некоторую страницу из вне в физическую оперативную память. Для этой загрузки нужно какую-то из страниц, которые находятся в оперативной памяти, выкинуть. Для выполнения данной операции существует несколько модельных алгоритмов. Эти алгоритмы используют некоторые признаки, которые ассоциируются с каждой из страниц. Так, R – **признак обращения**, то есть признак, который указывает на то, что в страницу что-то записывалось или из неё производилось чтение. Признак M – **признак модификации** указывает на то, что страница изменялась. Значения данных признаков устанавливаются аппаратно (соответственно при чтении или при записи), операционная система может обнулять эти признаки.

Рассмотрим алгоритм **NRU** (Not Recently Used – не использовавшийся в последнее время). Его действие состоит в следующем:

1. При запуске процесса M и R для всех страниц процесса обнуляется
2. Через некоторые, предопределённые промежутки времени по прерыванию по таймеру, операционная система обнуляет все биты R
3. При возникновении ситуации принятия решения о замещении мы анализируем признаки M и R для всех страниц. Соответственно, мы выделяем четыре класса:
   - Класс 0: $\begin{cases} M = 0 \\ R = 0 \end{cases}$

ФАКУЛЬТЕТ ВЫЧИСЛИТЕЛЬНОЙ МАТЕМАТИКИ И КИБЕРНЕТИКИ МГУ ИМЕНИ М.В. ЛОМОНОСОВА

- Класс 1: $\begin{cases} M = 1 \\ R = 0 \end{cases}$
- Класс 2: $\begin{cases} M = 0 \\ R = 1 \end{cases}$
- Класс 3: $\begin{cases} M = 1 \\ R = 1 \end{cases}$

4. Выбираем случайную страницу из непустого класса с минимальным номером. То есть, наиболее приоритетным для нас является признак доступа.

Рассмотрим следующий алгоритм – **FIFO**. В этом случае мы откачиваем ту страницу, которая находилась в памяти дольше всего. Здесь признаки не нужны, операционная система формирует для каждой страницы время. тогда, в ситуации страничного прерывания по временам пребывания выбирается самая старая страница.

У данного алгоритма есть проблема: не обязательно страница, которая является "старой не используется. То есть, мы может ошибочно откачать страницу, которая интенсивно используется.

Модификация алгоритма (алгоритм вторая попытка):

1. Выбирается самая старая страница. Если $R = 0$, то она заменяется
2. Если $R = 1$, то R обнуляется, обновляется время загрузки страницы в память (то есть переносится в конец очереди)

Простейшая модификация данного алгоритма – **алгоритм часы**. В этом случае все странички размещаются по круговому списку, некоторый маркер передвигается по часовой стрелке. При возникновении прерывания совершается одно из действий:

1. Если $R = 0$, то выгрузка страницы и стрелка на позицию вправо
2. Если $R = 1$, то R обнуляется, стрелка на позицию вправо и на п.1

Следующий алгоритм используется для поиска редко используемых страниц – **алгоритм NFU** (Not Frequently Used). В этом случае мы для каждой страницы i определим счётчик $Count_i$. Далее, по таймеру к счётчику прибавляется соответствующее значение признака R. То есть, счётчик фактически будет аккумулировать

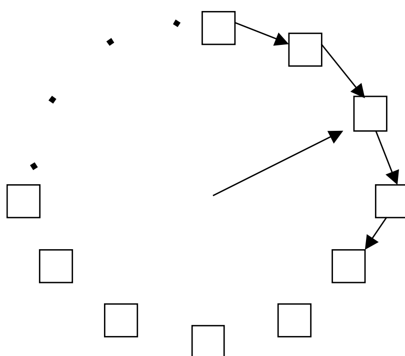

Рис. 20.3. Алгоритм часы

интенсивность работы со страничкой. Далее, когда происходит страничное прерывание, мы выбираем страницу с минимальным значением счётчика.

Данный алгоритм имеет следующие недостатки:

1. помнит старую активность
2. при большой активности возможно переполнение счётчика

Данный алгоритм можно модифицировать – будем не скалывать значения счетчика, а сдвигать на один разряд и логически добавлять в счетчик соответствующий код. Модификация NFU – **алгоритм старения**:

1. значение счётчика сдвигается на один разряд вправо
2. значение R добавляется в крайний левый разряд счётчика

Сегментная организация памяти

Исторически в компьютере сначала появилась сегментная память, а потом – страничная. Вспомним, что одним из недостатков модели страничной памяти является тот факт, что данная модель не соответствует логической структуре памяти процесса. В процессе существует адресное пространство кода – место, где находятся машинные команды. Также в процессе есть адресное пространство, в котором находятся константы и статические переменные. При этом в процессе присутствует стек. Можно перечислить множество конструкций, входящих в структуру памяти процесса. То есть, получаем, что структура памяти процесса неоднородна. Ее сложно представить в виде одного неоднородного непрерывного диапазона адресов. Страничная

ФАКУЛЬТЕТ ВЫЧИСЛИТЕЛЬНОЙ МАТЕМАТИКИ И КИБЕРНЕТИКИ МГУ ИМЕНИ М.В. ЛОМОНОСОВА

организация памяти никак не решает эту проблему. У нас есть одно виртуальное адресное пространство, и на него отображается логическая структура памяти каждого процессов. Для разделения логических компонент памяти используется сегментная организация памяти.

Основные концепции:

1. виртуальное адресное пространство процесса представляется в виде совокупности сегментов
2. каждый сегмент имеет свою виртуальную адресацию
3. виртуальный адрес представляется в виде пары номер сегмента - смещение относительно начала сегмента

Для реализации сегментной памяти используется следующее:

1. в процессоре имеет аппаратная таблица сегментов
2. эта таблица имеет количество строке, равно предельному количеству сегментов, которые обрабатываются и (или) реализуются в данном компьютере
3. каждая строка этой таблицы содержит пару значений – адрес начала сегмента в физической памяти (база) и размер сегмента

Таблицы используются следующим образом. Мы имеем исполнительный виртуальный адрес, который представляется в виде пары: номер сегмента и смещение относительно начала сегмента. Процессов автоматически выбирает номер сегмента, и индексирует по таблице сегментов. Получает строку, в которой находится описание заданного сегмента. Далее, проверяем, не выходит ли наш виртуальный адрес за пределы сегмента (смещение больше или меньше размера сегмента). Если смещение больше размера сегмента, то происходит прерывание – мы пытаемся обратиться к запрещённой памяти вне заданного сегмента. Если смещение меньше или равно размера сегмента, мы добавляем к базовому адресу смещение и получаем физический адрес.

Недостатки сегментной организации памяти:

1. весь сегмент должен находиться в оперативной памяти, мы не можем загрузить какую-то его часть
2. сегментная память возвращает актуальность решения проблемы фрагментации (возможна фрагментация)

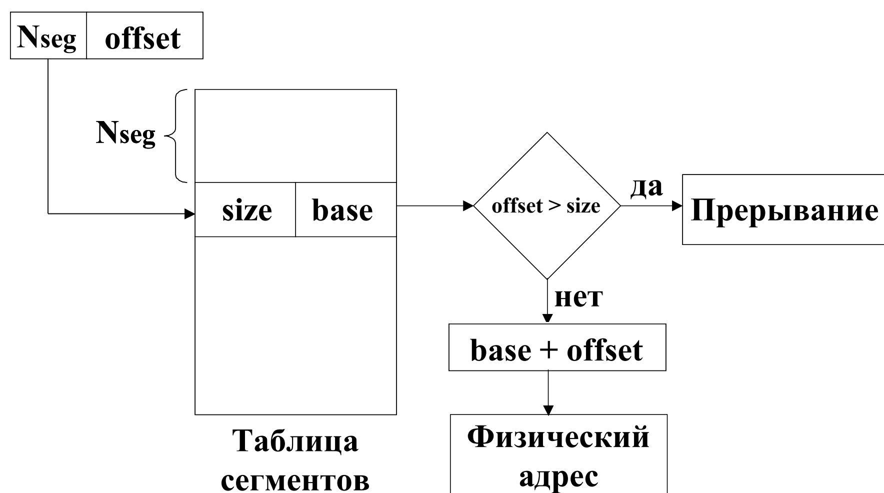

Рис. 20.4. Реализация сегментной памяти

3. весь сегмент находится в памяти на протяжении всей обработки процесса

4. накладные расходы при свопинге (свопинг – процесс перемещения информации о программе из внешней памяти в оперативную и обратно)

Сегментно-страничная организация памяти

В случае сегментно-страничной организации памяти виртуально-адресное пространство процесса представляется в виде сегментов, где каждый из сегментов имеет страничную организацию. Это означает, что виртуальный адрес состоит из трех компонент: номер сегмента, номер страницы в сегменте и смещение по странице. Таки образом, сперва мы индексируемся по номеру сегмента, из этого номера сегмента получаем таблицу страниц сегмента и далее действуем как в случае сегментной памяти внутри указанного сегмента.

Сегментно-страничная память достаточно универсальна. Можно свести ее адресацию к чисто страничной памяти (можно выделить для процесса не группу сегментов, а один сегмент).

Файловые системы

Обсудим **файловую систему** как компонент операционной системы, представляющий собой организованную структуру данных, хранящуюся на внешних запоминающих устройствах и обеспечивающих именованный доступ к данным. Таким

образом, два основных тезиса данного определения – хранение данных и именованный доступ.

На компьютерах первого поколения были доступны внешние запоминающие устройства (изначально это была магнитная лента). С помощью магнитной ленты можно было организовывать как оперативное, так и долговременное хранение данных. Это устройство было неэффективным: магнитная лента – устройство последовательного доступа, то есть устройство, которое предполагает организацию данных порциями переменного размера. Соответственно, чтобы прочесть некоторую запись, нужно было рассмотреть все предыдущие от начала ленты и считать информацию. Если лент несколько, то программисту необходимо знать, какая информация на какой ленте находится (то есть нужно знать координаты для записи информации и самостоятельно решать задачу защиты от несанкционированного доступа).

После появляются жесткие магнитные диски – первые устройства прямого доступа. Эти устройства позволяют организовывать многопользовательскую работу. С точки зрения организации и доступа к данным всегда нужно было помнить, начиная с какого сектора, с какой дорожки начинается запись информации.

Появление файловых систем решило обозначенную выше проблему. Оно позволило организовать на внешних запоминающих устройствах хранение именованных наборов данных, а также организовать поименный доступ к содержимому этих данных. Теперь достаточно знать только имя файла, чтобы добраться до содержимого файла, прочесть, модифицировать или удалить содержимое.

Структурная организация файлов

1. файл, как **последовательность байтов** – данные сохранялись просто как последовательность байтов, интерпретация зависела от программы

2. файл как **последовательность записей переменной длины**. Эта модель появилась вследствие того, что появились дисковые устройства и возникла необходимость отображать содержимое ленточной информации (в виде записи переменной длины) на устройствах.

3. файл как **последовательность записей постоянной длины**. Эта модель появилась для отображения колоды перфокарт, где запись постоянной длины

ФАКУЛЬТЕТ ВЫЧИСЛИТЕЛЬНОЙ МАТЕМАТИКИ И КИБЕРНЕТИКИ МГУ ИМЕНИ М.В. ЛОМОНОСОВА

– это предельная по объёму информация, которая могла различаться на перфокартах.

4. иерархическая организация файлов (дерево). записи находятся в узлах дерева (возможны записи перченной длины).

Сейчас основной моделью представления файлов является модель файл как последовательность байтов. Далее мы можем интерпретировать модель.

Для файлов как для объектов файловой системы присущи некоторые характеристики, описывающие свойства этого файла, режимы его использования и прочее. данные характеристики называются **атрибутами файла**. Перечислим их:

- имя
- права доступа
- персонификация (создатель, владелец)
- тип файла
- размер записи
- размер файла
- указатель чтения (записи)
- время создания
- время последней модификации
- время последнего обращения
- предельный размер файла
- ...

Для каждой файловой системы будет свой набор характеристик.

Права доступа к файлу – информация, которая регламентирует возможности доступа к файлу для разных пользователей и (или) категорий пользователей. Персонификация достаточно жестко связана с правами доступа. Например, если мы берём некоторый файл и запускаем его как процесс, то права этого процесса будут определяться той информацией, которая была записана или хранилась в персонификации

этого файла.

От типа файла может зависеть интерпретация его содержимого, то есть в разных файловых системах могут поддерживаться различные типы файлов. В зависимости от этого типа может по-разному происходить обработка и интерпретация содержимого.

К атрибутам может относиться информация о размерах записи файла блока. Почти во всех файловых системах присутствует понятие указатель чтения (записи). Это может быть один или несколько указателей.

Основные сценарии работы с файлом

Практически во всех файловых системах есть предопределённые сценарии работы с файлами, то есть последовательность шагов, которые нужно и можно выполнить, чтобы можно было работать с файлом. В подавляющем большинстве случаев этот сценарий можно разбить на несколько этапов.

Первый этап – **начало**: открытие файла (регистрация в системе возможности работы процесса с содержимым файла.) На этом этапе процесс запрашивает у операционной системы разрешение на работу с конкретным файлом. Если разрешение получено, то фактически формируется сеанс работы с содержимым этого файла.

Следующий этап – **работа с содержимым файла, с атрибутами файла**. Работа с атрибутами – изменение значений атрибутов.
третий этап – **завершение** (закрытие файла): системе подается информация о том, что процесс закрывает сеанс работы с файлом. Единовременно (как в одном процессе, так и в системе в целом) может быть несколько сеансов работы с одним и тем же файлом.

Операция закрытия файла – это операция возврата тех ресурсов операционной системы, которая она выделила для организации работы с файлом. Пример таких ресурсов, которые выделяет операционная система и по завершении закрывает – ресурсы, связанные с буферизацией обменов.

Для каждого сеанса с открытым файлом операционная система формирует системную структуру данных, в котором находится актуальная информация об от-

крытом файле. эта системная структура данных называется **файловым дескриптором**.

Типовые программные интерфейсы работы с файлами

Каждая файловая система предоставляет пользователю переопределённый набор интерфейсов для обеспечения работы с файлами. Сценарий заполняется интерфейсами, которые в подавляющем большинстве реализуются как системные вызовы:

- **open** – открытие (создание) файла
- **close** – закрытие
- **read(write)** – читать, писать (относительно положения чтения (записи))
- **delete** – удалить файл из файловой системы
- **seek** – позиционирование указателя чтение (запись)
- **rename** – переименование файла
- **read (write attributes)** – чтение, модификация атрибутов файла

Лекция 21. Файловые системы.

Файловые системы. Структурная организация.

Файловая система – ключевой компонент любой операционной системы. Файловая система – это пример компонента операционной системы, который реализует виртуальные ресурсы, то есть виртуальные ресурсы как файлы и все его реквизиты.

Ранее мы обсудили, что файловая система есть комплекс программ, который позволяет организовывать хранение, доступ и использование именованных наборов данных. В случае работы с именами работа с информацией значительно упрощается, так как упрощается доступ к ней.

Ранее мы рассмотрели структурную организацию данных:

1. файл, как **последовательность байтов** – данные сохранялись просто как последовательность байтов, интерпретация зависела от программы
2. файл как **последовательность записей переменной длины**. Эта модель появилась вследствие того, что появились дисковые устройства и возникла необходимость отображать содержимое ленточной информации (в виде записи переменной длины) на устройствах.
3. файл как **последовательность записей постоянной длины**. Эта модель появилась для отображения колоды перфокарт, где запись постоянной длины – это предельная по объёму информация, которая могла различаться на перфокартах.
4. иерархическая организация файлов (дерево). записи находятся в узлах дерева (возможны записи перченной длины).

Последовательность записей фиксированной длины появилась вследствие появления перфокарт (каждая перфокарта вмещала 80 символов). Запись переменной длины подразумевала помимо содержательной части разделители – маркеры, которые указывают на то, что запись закончилась или началась.Файлы, имеющие организацию памяти постоянной длины не требуют подобных разделителей. Могут быть более сложные организации файлов – иерархические (файл организуется в виде дерева и каждая запись этого дерева, находящаяся в узле, состоит из поля данных и поля-ключа).

С другой стороны, любой файл можно представлять как последовательность байтов, а далее, на следующем уровне, интерпретировать её в той или иной структуре.

Атрибуты файла и основные сценарии работы

Во всех файловых системах с каждым из файлов связывается совокупность системных данных, которая описывает актуальное состояние каждого из файлов. Эта совокупность данных называется **атрибуты файла**. Состав атрибутов зависит от той или иной операционной системы. Есть выделенная позиция – имя файла. Наиболее характерные атрибуты файла, которые бывают в большинстве файловых систем:

- имя
- права доступа
- персонификация (создатель, владелец)
- тип файла
- размер записи
- размер файла
- указатель чтения (записи)
- время создания
- время последней модификации
- время последнего обращения
- предельный размер файла
- ...

Любая файловая система предусматривает некоторые предопределенные, зафиксированные сценарии работы с файлами. Практически во всех файловых системах эти сценарии предполагают, что перед началом работы с содержимым файла идёт запрос на разрешение работы с этим файлом и фактически создание некоторого виртуального сеанса работы с файлом.

Обычно сеанс работы с файлом создаются посредством так называемого открытия файла. В подавляющем большинстве систем это системный вызов **open**. Соответственно, при вызове open выполняется проверка – существует ли указанный

файл. Если все нормально, то в результате открытия создаются системные структуры данных, которые позволяют организовывать работу с содержимым этого файла, позволяют оптимизировать работу с содержимым этого файла. К таким структурам данных, в частности, относится указатель чтения (записи). Эта информация должна быть в начальный момент сеанса настроена на некоторую начальную позицию и после каждого обращения к файлу этот указатель (указатели) меняет свой значение.

Вычислительная система представляется в виде совокупности взаимодействующих компонент, где каждая из компонент имеет свои характеристики производительности и эффективности. В случае работы в системе такие характеристики должны быть согласованы. Здесь возникает вопрос оптимизации.Аналогично тому, как происходит сглаживание дисбаланса между оперативной памятью и центральным процессором, мы наблюдаем дисбаланс между оперативной памятью и скоростью внешних запоминающих устройств (то есть устройств, на которых размещается внешняя файловая система). Для задач сглаживания в каждом сеансе работы предусматривается выделение системных ресурсов, которые позволяют сглаживать дисбаланс работы внешнего запоминающего устройства и оперативной памяти.

Отметим, что содержимое системной структуры данных отражает актуальное состояние сеанса. Обычно, с одним и тем же файлом может быть ассоциировано два и более сеансов связи – один и тот же файл может быть открыт в системе многократно.

Следующий этап сценария – работа с содержимым файла и с атрибутами. Почти все файловые системы предоставляют системные вызовы, которые позволяют организовывать работу с атрибутами файлов. То есть, мы можем изменять имя файла, указатель чтения (записи), права доступа. также, работая с атрибутами, мы можем добавить к файлу дополнительные имена. Многие файловые системы позволяют для файла (как для совокупности именованных данных) присваивать два и более имен.

Закрытие файла – закрытие сеанса работы с файлом. Соответственно, этап закрытия – это возврат выделенных операционной системой ресурсов обратно в операционную систему. Что, и как выделяется и возвращается – зависит от конкретной операционной системы.

Типовые программные интерфейсы работы с файлами

Каждая файловая система предоставляет для работы с файлами унифицированный программный интерфейс – множество системных вызовов, которые обеспечивают обращение и использование файловой системы. Перечислим эти системные вызовы:

- **open** – открытие (создание) файла
- **close** – закрытие
- **read(write)** – читать, писать (относительно положения чтения (записи))
- **delete** – удалить файл из файловой системы
- **seek** – позиционирование указателя чтение (запись)
- **rename** – переименование файла
- **read (write attributes)** – чтение, модификация атрибутов файла

Модельная организация

Компонент, который присутствует практически во всех файловых системах – это каталог. **Каталог** – компонент файловой системы, содержащий информацию о содержащихся в файловой системе файлах. Каталоги являются специальным видом файла. То есть, каталог – это системная структура данных, через которую мы можем добраться к содержимому существующих файлов.

В некоторых файловых системах каталог есть внутренняя системная структура данных, в некоторых системах каталог – это файла специального типа.

Существует достаточно большое количество моделей организация файловых систем. Простейшая модель – одноуровневая, то есть когда все файлы представляются в виде одно броневого списка. В одноуровневой файловой системе не допускается коллизия имён. Данная система может быть использована для различных управляющих систем.

На этой основе можно получать модели с ограниченным количеством уровнем, например, двухуровневую.Здесь появляется общий, корневой каталог и каталоги следующих уровней. Каждый каталог второго уровня может содержать перечень своих файлов. Соответственно, увеличиваются возможности: возможна коллизия имен

файлов, которые находятся в разных каталогах. Такая модель может быть использована для простейшей многопользовательской системы.

Третья модель – **иерархические файловые системы**. Это файловая система, которая предоставляется в виде дерева. В дереве есть утри категории: один корень, узлы, не являющиеся листьями и узлы, являющиеся листьями. Соответственно, узлы, являющиеся листьями, это файлы. Корневой каталог и узлы, не являющиеся листьями, это каталоги.

Таким образом, мы имеем организацию по файлам такую, что в каждом каталоге не допускается коллизия имен. То есть файловая система содержит информацию о файлах в виде иерархической, древообразной структуры. В целом повторяющихся имён файла модем быть произвольное количество, но такая иерархическая структура позволяет организовывать уникальное именование имён. Исходя из древообразной организации и ограничения, что в каждом каталоге не может быть коллизии имён, можно утверждать, что если мы выпишем пусть от корня файловой системы до любого файла (пусть как последовательность имён), то эта строка в древообразной системе будет уникальной. Соответственно, можно говорить, что такая модель файловой системы оперирует со следующими понятиями:

- имя файла
- полное имя файла
- относительное имя
- домашний каталог
- текущий каталог

Здесь **имя файла** – это информация о регистрации файла в своем каталоге. В целом по такой файловой системе может быть многократная коллизия имён файлов.

Полное имя файла – это путь от корневого каталога до имени файла, с которым мы хотим работать. Полное имя уникально. Эта строка однозначно определяет конкретный файл.

Возьмем некоторый каталог и рассмотрим поддерево, которое исходит из этого каталога. Тогда относительно этого каталога у нас есть редуцированные цепочки –

относительное имя.

Иерархические файловые системы оперируют с двумя понятиями: текущий каталог (каталог, с которым в данный момент настроена работать система) и домашний каталог (каталог, который по умолчанию становится текущим при входе пользователя в систему). Логично сделать для каждого пользователя свой уникальный домашний каталог.

Структура системного диска

Рассмотрим подходы к практической реализации файловой системы. Структуру системного диска можно представить следующим образом. Нулевой блок – это блок программного загрузчика, там находится код программного загрузчика той операционной системы, которая связана с этим системным устройством. Из этого блока аппаратный загрузчик считывает загрузчик Windows или Unix системы. Далее возникает некоторое технологическое соревнование между объемом внешний запоминающих устройств и адресацией памяти. В какие-то моменты мы можем представить в адресуемой единице памяти номер любого блока любого дискового устройства. Однако, если количество блоков превосходит разрядность машины, мы не можем адресовать некоторый блок. Чтобы решить эту проблему, используется система разделов. Системно разделяется пространство диска на некоторые области, которые называются разделами. Это разбиение находился в первом блоке системного диска Таблица разделов – это совокупность пар: начало раздела - конец раздела. Соответственно, в каждом разделе может находиться содержательная информация, то есть каждый раздел рассматривается системой как виртуальный диск. Мы можем установить на некоторый специальный регистр внешнего запоминающего устройства, что мы работаем с разделом в диапазоне. Далее, мы можем производить обмены в терминах относительных блоков (блоков относительно начала раздела).

Какой-то из разделов (или несколько разделов) может быть выбран как системный. В какала из этих системных разделов может находиться своя операционная система. В этом случае основной программный загрузчик будет не загрузчиком конкретной операционной системы, а мультисистемным загрузчиком. То есть, аппаратный загрузчик будет запускать программный загрузчик, а программный загрузчик, зная структуру системного диска и координаты системных разделов, может предложить загрузку той или иной операционной системы.

Рассмотрим внутреннюю структуру системного раздела. Его структура в некотором роже идентична описанной. Нулевой блок – загрузчик операционной системы. Далее идёт совокупность блоков (обычно это фиксированное пространство), в которой находится системная информация, то есть системные структуры данных, предназначенные для организации функционирования файловой системы и операционной системы. После пространства системной информации находится область, в которой находятся данные файлов. В этой области расположены компоненты, принадлежащие файлам (то есть блоки, содержащие информацию тех или иных файлов) и свободное пространство (незанятые блоки).

Отметим, что имеет место некоторая иерархия ресурсов, начинающаяся от физических ресурсов и продолжающаяся и заканчивающаяся виртуальными ресурсами. Например, рассмотрим блок диска. С каждым дисковым устройством ассоциируется физический блок, он имеет некоторое размещение и характеристики. Следующий уровень – блок виртуального диска. Он унифицирует дисковое устройство, он может представляться своим размером, который не обязательно совпадает с размером физического диска. Далее может быть блок файловой системы, то есть мы можем настроить файловую систему так, чтобы она работала с блоками определённого размера.

Мы можем ввести ещё один уровень виртуальности блоков. При открытии файла будем работать с логическими блоками заданного размера. В этом случае обмены будут происходить порциями блока (то есть порциями заданного размера).

Модели реализации файлов. Непрерывные файлы.

Простейшая модель реализации файлов – организация файлов виде непрерывных областей. То есть, если есть некоторая последовательность файлов, то для первого файла выделяется некоторая область необходимого размера; все блоки файла находятся в подряд идущих блоках дискового устройства. Когда файл заканчивается, мы аналогично начинаем следующий файл.

У данной медли есть очевидные **достоинства**:

- простота реализации
- высокая производительность

Чтобы реализовать данный метод, достаточно иметь в каталоге имя файла и два числа: номер начального блока и количество блоков в файле (длина файлов в блоках). При этом практически отсутствуют расходы.

Однако, есть и ряд **недостатков** данного метода:

- фрагментация свободного пространства
- возможность увеличения размера существующего файла

Фрагментация пространства предполагает, что возможно существования свободных областей таких размеров, что их недостаточно, для того, чтобы разместить в них какой-то из файлов. Соответственно, возникает процесс деградации. Данная проблема решается с помощью компрессии (перенос всех файлов и размещение их без свободных фрагментов). При этом компрессия – долговременная и дорогостоящая операция.

Второй недостаток связан с тем, что объем файлов, с которыми мы работаем, может варьироваться. Тогда, если нужно увеличить размер существующего файла, поступают следующим образом. Первый способ: берём существующий файл, находим для него фрагмент свободного пространства на диске, достаточный для хранения этого файла и блок для расширения. Переносим файл в блок и расширяем. Тогда на месте расширенного файла возникает свободный фрагмент, что приводит к увеличению фрагментации. Описанная операция неэффективна. Второй способ решения проблемы: при создании файла (то есть при выделении под файл пространства) мы всегда резервируем под него некоторый запас. Размер этого резерва можно статистически подсчитать.

Файлы, имеющие организацию связанного списка

Рассмотрим модель использования файла как списка блоков. Эта модель предназначена для ухода от использования непрерывных областей дисков.
Идея: в каждом блоке имеется фиксированное пространство, которое не используется для хранения содержательной информации файла, а используется для указания на следующий блок. Таким образом, файл состоит из совокупности блоков, связанных списком. В каждом файле есть ссылка на следующий блок. Если следующего блока нет, эта ссылка может иметь некоторое предопределённое значение. В каталоге должны находиться имя файла и номер начального блока. Соответственно, при

открытии файла мы получаем номер начального блока. Читая эти ссылочные элементы, мы можем добраться до любого блока файла.

Достоинства метода:

- отсутствие фрагментации свободного пространства (за исключением блочной фрагментации)
- простота реализации
- эффектный последовательный доступ

Данный подход решает проблему фрагментации по блокам в файловой системе. Здесь уже не будет фрагментов блоков, которые мы не можем использовать.

Недостатки метода:

- сложность (неэффективность) организации прямого доступа
- фрагментация файла по диску
- наличие ссылки в блоке файла (ситуация чтения двух блоков при необходимости чтения данных объёмом один блок)

Невозможность эффективной организации прямого доступа заключается в том, что чтение любого блока требует предварительного чтения всех предыдущих блоков.

Такая организация файла приводит к тому, что со временем содержимое файла будет сильно фрагментировано по пространству диска. То есть, один блок будет находиться в начале дискового пространства, второй может находится в середине, третий – в начале, четвертый – в конце. тогда, если мы хотим последовательно читать содержимое файла, головки механических дисковых систем будут перемещаться по всему дисковому пространству. Работа в таком режиме очень быстро приводит к выходу из строя диска.

Нарушается логическая структура данных, связанная с тем, что обычно обмен осуществляется в терминах блоков содержательных данных. При такой организации в файловой системе для чтения одного блока данных требуется два обмена.

Таблица размещения файловой системы FAT

Рассмотрим следующую модель организации файлов – модель таблицы размещения файловой системы.

Идея: файловая система операционной системы формирует таблицу фиксированного размера, которая называется **FAT-таблица.** Размеры FAT-таблицы соответствуют размеру пространства, в котором находятся блоки файлов в файловой системе (количество блоков файлов, которые может содержать файловая система в предельно случае). i-я строчка этой таблицы содержит информацию о статусе использования i-го блока файловой системы. Для каждого файла в такой файловой системе в каталоге будет содержаться имя файла и номер начального блока. Далее, по номеру начального блока можно обратиться к FAT-таблице и прочесть содержимое записи, которое соответствует начальному блоку. Если у файла есть еще блоки, то в этой записи будет номер следующего блока. То есть по FAT-таблице организован список блоков для каждого файла. Строчка последнего блока файлов содержит нулевую ссылку.

Предполагается, что FAT-таблица находится в оперативной памяти. Это означает, что если нужно прочесть в каком-то файле i-й блок, то через каталог получаем номер стартового блока и обращаемся в FAT-таблицу. Далее, через FAT-таблицу получаем список блоков этого файла.

Описанная модель похожа на предыдущую. Однако есть отличие: у нас нет накладных расходов, связанных с непроизводительными обменами. Мы перемещаемся по оперативной памяти. Работа с данными в оперативной памяти (если мы оцениваем работу с внешними запоминающими устройствами) не образует накладных расходов. То есть, накладной расход при работе с внешним устройством – это непроизводительные обмены.

Достоинства модели:

- возможность использования всего блока для хранения данный файла
- оптимизация прямого доступа (при полном или частичном размещении таблицы в ОЗУ)

Видно, что данная модель позволяет уйти от проблемы логической структуры реальной организации данных на диске.

Недостатки модели:

- желательно размещение всей таблицы в ОЗУ

Недостаток модели аналогичен проблеме страничной организации памяти. Пока диски были относительно небольшие, FAT-таблица помещалась в оперативную память. В современных условиях FAT-таблица становится довольно большой, вследствие чего становится нерациональным держать ее целиком на оперативной памяти.

Лекция 22. Основные концепции файловых систем.

Использование индексных дескрипторов (узлов)

Рассмотрим следующую модель – использование индексных узлов. **Индексный узел (дескриптор)** – системная структура данных, содержащая информацию о размещении блоков конкретного файла в файловой системе.

Идея: для каждого файла система формирует специальную структуру данных, которая содержит информацию о размещении блоков файла. Соответственно, эта системная информация размещается в пространстве, выделенном в разделе, предназначенном для хранения параметров настройки системной информации. То есть, упрощенно, индексный дескриптор – это совокупность каких-то характеристик файла и перечень блоков файла (то есть номер блока, в котором находится нулевой блок файла, первый блок и т.д.). При открытии файла файловая система находит для заданного файла его индексный дескриптор, и если соблюдаются все права доступа, то открывается сеанс работы с файлом, а индексный дескриптор помещается в оперативную память операционной системы. Далее каждый обмен проходит через информацию индексного дескриптора. То есть в оперативной памяти, принадлежащей операционной системе будут храниться все открытые индексные дескрипторы.

Таким образом, мы ушли от необходимости размещения всей таблицы. При этом появляются некоторые ограничения на индексный дескриптор. Если мы хотим работать с индексным дескриптором предопределённого размера, то скорее всего мы тем самым наложим ограничения на количество блоков, которое может быть у файла. Если мы работаем с индексным дескриптором произвольного размера, то в таком индексном дескрипторе мы можем указать информацию любого файла любого размера. Но в этом случае мы возвращаемся к проблеме FAT-таблицы (таблица слишком большая, чтобы держать её в оперативной памяти). Существует два возможных решения данной проблемы. Можно сразу наложить ограничение на размер файлов, с которыми мы работаем. Однако это неприемлемое решение в современных условиях. Второе решение: создать иерархическую организацию индексного дескриптора, то есть дескриптор будет состоять из двух частей – одна часть ограниченного (фиксированного) размера, в ней будет находиться предопределённая информация. Вторая часть дескриптора будет динамической. В этом случае организация индексного дескриптора и программы работы с ним будут более сложными. При этом динамическая часть дескриптора будет храниться во внешней памяти, то есть будут появляться наклад-

ные расходы, которые влияют на производительность.

Подведем итоги. **Достоинства модели:**

- нет необходимости в размещении в ОЗУ информации всей FAT-таблица о всех файлах системы, в памяти размещаются атрибуты, связанные только с открытыми файлами.

Недостатки модели:

- размер файла и размер индексного узла (в общем случае прийти к размерам таблицы размещения)

Решение:

- ограничение размера файла
- иерархическая организация индексных узлов

Организация каталогов

Системе удобнее работать с таблицами, содержащими записи фиксированного размера. В этом случае мы можем оценивать все характеристики используемых программ. В связи с этим каталог может организовываться как записи фиксированного размера, в котором два типа полей – имя и атрибуты. Объем атрибутов файла может быть достаточно большим.

Альтернативный вариант: каталог содержит записи фиксированного размера и два поля – имя и ссылка на атрибуты. Атрибуты могут храниться на внешнем устройстве.

Обе описанные модели обсуждаются при условиях, когда длина имени файла ограничена. Это не всегда удобно. Поэтому, если нам необходимо работает с длинными именами, мы можем префиксную часть держать в поле имени, а суффиксную – в атрибутах.

Соотношение имени и содержимого файла

Существует две модели соотношения имени и содержимого файла.

1. Содержимому любого файла соответствует единственное имя файла (то есть существует взаимно-однозначное соответствие между именем и содержимым файла). В качестве примера можно привести иерархическое содержание.

2.Содержимому файла может соответствовать два и более имен файла. Эти имена могут использоваться в различных стратегиях. В такой мультиименной системе может быть две реализации. Первая реализация – это так называемая **жесткая связь** между именем и содержимым. В этом случае имя является одним из атрибутов файла, причём атрибутов этого типа у файла модем быть произвольное (в пределах существующего ограничения) количество. Каждое из имён равноценно, то есть мы можем работать с содержимым файла равноценно, используя все возможные функции, через любое из этих имён. Если мы вернемся к иерархической системе, то в дереве у будут храниться имена (из разных каталогов можно ссылаться на одно и то же содержимое). Таким образом, в случае жёсткой связи системный вызов "удалить файл"эквивалентен вызову "удалить имя файла". Этот вызов удаляет имя файла. Если у файла больше нет имён, то удаляется и его содержимое. Отметим, что в случае жесткой связи помимо совокупности имён в атрибутах должен быть считчик этих имён.

Второй тип многоименного именования – так называемая **символическая связь**. Ее идея заключается в следующем. Имеется файл специального типа, содержимое которого есть полный путь на некоторой существующий в файловой системе файл. Мы можем работать с содержимым оригинального файла через файл-ссылку при использовании системных вызовов обмена. Однако, когда мы удаляем этот файл, содержимое файла, на который он ссылался, остается неизменным в любом случае. Соответственно, если мы удаляем файл, на который ссылается файл-ссылка, то образуется подвешенная ссылка (неработающая ссылка).

Координация использования пространства внешней памяти

Проблема – размер блока файловой системы.

"Большой блок":

- эффективность обмена
- существенная внутренняя фрагментация (неэффективное использование пространства ВП)

"Маленький блок":

- эффективное использование пространства ВП
- фрагментация данных по диску

Проблема – определение оптимального размера блока.

Ресурсы системы

Рассмотрим вопросы организации информации об имеющихся и используемых в системе ресурсах. Сперва рассмотрим **учет свободных блоков файловой системы**. Согласно организации структуры системного раздела, в разделе имеется пространство, которое изначально было выделено для размещения в нем блоков файлов. В этом пространстве часть блоков занята файлами (номера этих блоков будут встречаться в индексных дескрипторах). Часть блоков остаются свободными. Для учета и координации этого системного ресурса необходимо научиться работать с информацией о свободных блоках. Простейшая модель предполагает организацию списка свободных блоков в виде связного списка блоков. В каждом Блоке перечислены номера свободных блоков, причем в каждом таком блоке есть ссылка на следующий блок. Если свободных блоков больше, чем размер блоков записи, то существует ссылка на следующий блок (и так далее до последнего). Эти блоки размещаются в пространстве блоков, которое выделено для хранения блоков файлов. Этот ресурс никак не вступает в противоречия с ресурсом, выделенном для хранения блоков файлами.

В качестве альтернативы представления свободных блоков есть решение, состоящее в **использовании битового массива**. Пусть есть некоторый массив с любыми переменными. Переименуем в этом массиве все разряды. Таким образом получаем битовый массив. Информация более компактна. Более того, данная модель решает проблему фрагментации блоков по устройству.

Итог: состояние любого блока определяется содержимым бита с номером каждого блока. Если блок свободен, бит равен 1, занят – 0.

Квотирование пространства внешней памяти

Пусть есть некоторый сервер. На нем работают операционная и файловая системы. С этим сервером работает 350 человек. Пользователи в разные моменты времени работаю с сервером, используя файловую систему. В этом случае некоторые

ФАКУЛЬТЕТ ВЫЧИСЛИТЕЛЬНОЙ МАТЕМАТИКИ И КИБЕРНЕТИКИ МГУ ИМЕНИ М.В. ЛОМОНОСОВА

ресурсы файловой системы будут ограничены. Пространство в разделе, выделенное на системную информацию, ограничено. Тогда количество файлов будет ограниченным. Более того, размер файла ограничен организацией файловой системы. Также будут ограничены блоки файлов. Соответственно, возможна ситуация, в которой один из пользователей по какой-то причине занял весь свободный ресурс сервера. Таким образом, для систем массового доступа необходимо средство регламентации использования ресурсов. Таким средством является средство бюджетирования или квотирования ресурсов. Для каждого из пользователей выделяется некоторый объем ресурса, который он может использовать. Если пользователь выходит за пределы этого объема, то возможна блокировка пользователя. Существует несколько моделей таких систем бюджетирования.

Рассмотрим систему, которая использует так называемые многоуровневые лимиты. **Идея**: для каждого пользователя (или для группы пользователей)определяется две квоты. **Жестка квота** – ограничение на используемый ресурс, которое невозможно обойти. Если мы хотим получить ресурс свыше жесткой квоты, то запрос будет блокирован дол тех пор, пока мы не откажемся от запроса или пока не сократим используемые ресурсы. Вторая характеристика – **гибкая квота** – это значение, которое меньше жесткого лимита, и которое в определённых случаях можно превосходить. Для работы с гибким лимитом используется так называемые **счётчик штрафов**. Далее в разных системах существуют разные способов работы со счетчиком.

Надежность файловой системы

Файловая система обеспечивает работу с данными. Поэтому обеспечение надежности файловой системы – важная задача. Наращение надёжности приводит к потере информации. Возможность потери информации может происходить по двум причинам:

- ошибки в программно-аппаратном обеспечении.

потери по неумышленному действию пользователя

Риски потерь информации можно разделить на две группы: потеря пользовательской информации (содержимое файлов) и потеря системной информации (теряется информация, размещенная в области системных данных). Обо ущерба, естественно, нежелательны, однако утеря системной и информации более существенна. Соответственно, файловая система должна быть надёжной. Надёжность – минимизация

ФАКУЛЬТЕТ ВЫЧИСЛИТЕЛЬНОЙ МАТЕМАТИКИ И КИБЕРНЕТИКИ МГУ ИМЕНИ М.В. ЛОМОНОСОВА

вероятности безвозвратной потери информации. В качестве решения возможно резервное копирование или архивирование.

Резервное копирование можно обеспечить различными способами. Первый способ – выборочное копирование (копируем только некоторые категории файлы, например, категории, относящиеся к типу содержимого). Объектные модули копировать не нужно, так как объектные модули из каких-то внешних программных систем, то их можно восстановить из дистрибутива. Объектные модули, полученные в результате компиляции программ, то их можно спасти путем компиляции (если исходные тексты сохранились).

Возможно инкрементное архивирование. Создается серия копий. Нулевая копия (мастер-копия) содержит копии всех объектов, которые мы хотим сохранить. С мастер-копией ассоциируется время и дата создания. После этого по необходимому расписанию запускается создание инкрементных копий. В день создания инкрементной копии мы выбираем для копирования только те файлы, которые были созданы или модифицированы после создания последней копии. Объем и уровень потерь зависит от частоты архивирования.

Использование компрессии подразумевает использование алгоритмов сжатия, что в свою очередь приводит к большой чувствительности к потери информации. Если из сжатого архива потерялся бит, то скорее всего весь архив будет утерян.

Подведем итог. **Резервное копирование (архивирование):**

- копируются не все файлы файловой системы (избирательность архивирования по типам файлов)
- инкрементное архивирование (резервное копирование) – единожды создаются "полная"копия, все последующие включают только обновлённые файлы
- использование компрессии при архивировании (риск потери всего архива из-за ошибки в чтении(записи) сжатых данных)
- проблема архивирования "на ходу"(во время копирования происходят изменения файлов, создание, удаление каталогов и т.д.)
- распределённое хранение резервных копий

Архивирование можно организовывать различными способами. Существует две стратегии архивирования. **Физическое архивирование** предполагает копирование блоков (а не файлов).

Физическая архивация:

- "один в один"
- интеллектуальная физическая архивация (копируются только использованные блоки файловой системы)
- проблема обработки дефектных блоков

Логическая архивация – копирование файлов (а не блоков), модифицированных после заданной даты

Проверка целостности системной информации

Проблема – при аппаратных или программных сбоях возможна потеря информации:

- потеря модифицированных данных в "обычных"файлах
- потеря системной информации (содержимое каталогов, списков системных блоков, индексные узлы и т.д.)

Современные операционные системы и их файловые системы обладают средствами то автоматического тестирования содержания своей системной информации и автоматического или автоматизированного восстановления. Существуют различные модели организации восстановления и контроля.

Рассмотрим модель проверки целостности файловой системы с точки зрения непротиворечивости блока файлов. **Модельная стратегия контроля:**

1. формируются две таблицы:
   - таблица занятых блоков
   - таблица свободных блоков

   Размеры таблиц соответствуют размеру файловой системы – число записей равно числу блоков ФС. Изначально все записи таблиц обнуляются.

2. Анализируется список свободных блоков. Для каждого номера свободного блока увеличивается на 1 соответствующая ему запись в таблице свободных блоков.

3. Анализируются все индексные узлы. Для каждого блока, встретившегося в индексном узле, увеличивается его счетчик на 1 в таблице занятых блоков.

4. Анализ содержимого таблиц и корреляция ситуаций. Если поэлементное сложение значений этих таблиц равно 1, то считаем, что данные целостные (блок или занят или свободен). В противном случае возможна ситуация, когда блок отсутствует и в списке занятых, и в списке свободных блоков. В этом случае мы принимаем решение, что было нарушение в списке свободных блоков файловой системы. Тогда этот номер добавляется в список свободных.

Возможна ситуация, когда в табличке свободных блоков для какого-то блока указано значение, большее 1. Это означает, что список свободных блоков некорректный. В этом случае запускается процесс пересоздания списка свободных блоков. Список свободных блоков пересоздается следующим образом: создаются альтернативная таблица – таблица занятых.

Возможна ситуация, в которой в какой-то записи таблицы занятых блоков значение больше единицы. Это означает, что один блок принадлежит одновременно более, чем одному файлу. То есть, информация в индексных узлах некорректна. Исправление: находим те файлы, которые содержат этот общий блок, например $Name_1$ и $Name_2$. Тогда

1. Копируем $Name_1$ в $Name_1'$

2. Копируем $Name_2$ в $Name_2'$

3. Удаляем $Name_1$ и $Name_2$

4. Запускаем переопределение списка свободных блоков

5. Обратное переименование файлов и фиксация факта их возможной проблемности

Лекция 23. Файловая система Unix.

Организация файловой системы Unix

Рассмотрим примеры конкретной реализации файловой с Файловая система Unix – первая широко распространенная иерархическая файловая система. До нее все распространённые системы были одноуровневыми и двухуровневыми. Концепция файлов в UNIX состоит из двух посулов: унификация использования файловых интерфейсов для работы со всеми легально зарегистрированными внешними устройствами (легально зарегистрированное устройство представляется в виде фактически, в UNIX системе нет команд в том смысле, в котором команда понималась до появления UNIX. Раньше все системы имели встроенную реализацию команд, которые были доступны для работы пользователей (я ядре был состав команд, которые нельзя было изменить). В UNIX интерпретатор команд – выделенная отдельная компонента, которую можно варьировать. Более того, реализация команд – это просто исполняемые файлы. Если нам не нравится наличие какой-то команды в системе, мы можем просто удалить этот файл из системы.

С точки зрения организации **файл операционной системы Unix** – это специальным образом именованный набор данных, размешенный в системе. Отметим, что файловая система поддерживает различные типы файлов(тип – набор значений и набор операций). Разные операционные системы поддерживают разный состав типов файлов. Перечислим **виды файлов**:

- обычный файл (regular file)
- каталог (directory)
- специальный файл устройств (special device file)
- именованный канал (FIFO)
- ссылка (link)
- сокет (socket)

Обычный файл – файл как последовательность байтов, который можно интерпретировать, обращаясь к тем или иным системным вызовам работы с этим файлом. То есть мы можем интерпретировать его как текстовый файл (запись переменной длины) или файл с фиксированной структурой записи.Таким образом, в основе такой файл представляет собой последовательность байтов; при работе этот файл будет

по-разному интерпретироваться в зависимости от того, какими средствами мы будем пользоваться при работе с его содержимым.

Каталог предназначен для организации связи между именем файла и его содержимым. Заметим, что в UNIX-системе у одного и то же содержимого может быть произвольное количество имён. Эти имена равнозначны.

Файл устройств – это специальный файл, в задачу которого входит установление соответствия между именем устройства и зарегистрированным в системе драйвера. То есть все те имена устройств, которые мы используем в работе, это имена файлов устройств.

Именованный канал (FIFO) – файл с фиксированной стратегией доступа. Из этой стратегии логически вытекает ограничение – мы не можем перемещать указатель чтения и записи произвольным образом. Он перемещается только посредством обращения к read или write.

Файл-ссылка – это мягкие ссылки, то есть содержимым файла является полное имя какого-то из файлов в файловой системе. Посредством использования ссылки можно добраться до содержимого того файла, на который эта ссылка ссылается.

Права доступа

UNIX-система (а соответственно и его файловая система) поддерживает три **категории пользователей**:

- пользователь
- группа (группа, которой принадлежит пользователь за исключением этого пользователя)
- все пользователи группы (за исключением группы, которой принадлежит пользователь)

Существует три категории **прав доступа**:

- на чтение
- на запись

- на исполнение

Отметим, что тот факт, что для файла разрешено исполнение, не означает, что он может быть запущен в качестве процесса. Внутри исполняемого файла в его главной части находится магическое число. Это число (код) при запуске проверяется системой. Если это число отсутствует, то процесс запустить не получится.

Для разных типов файлов указанные категории могут интерпретироваться по-разному. Например, для регулярного файла категории остаются теми же – читать из файла, писать в файл, исполнять файл. Если речь идёт о каталоге, то разрешение на чтение есть возможность открытия файла-каталога и чтение информации их него. Право на запись в каталог подразумевает, что мы можем удалять и создавать в данном каталоге. Исполнение – в систему может поступать от пользователя запрос на поиск в каталоге.

Логическая структура каталогов

С точностью до именования и некоторых минимальных перестановок логическая структура UNIX-системы представляется следующим образом. Пусть есть корневой каталог файловой системы. В нем могут находиться произвольные файла и файлы с предопределенным содержимым. Первый файл, который находится в корневом каталоге – это файл, в котором находится исполняемый код системы. Обычно в названии этого файла присутствует аббревиатура названия операционной системы. Программный загрузчик операционной системы UNIX знает, что в корневом каталоге файловой системы находится файл с таким именем и при старте системы (при работе программного загрузчика операционной системы) загрузчик залезает в структуру корневого каталога, находит указанный файл и загружает его в оперативную память (начинается процесс загрузки ядра). Если перемножать или ужалить этот файл, все будет работать до того момента, как потребуется перезагрузка.

В корневой файловой системе находятся каталоги с предопределёнными именами и предопределённым наполнением. BIN – каталог с исполняемыми файлами, реализующими команды общего назначения (то есть пользовательские команды). Следующий каталог – ETC – содержит файлы, содержащие системные таблицы, которые используются для сконфигурирования системы и задания тех или иных параметров работы. Также в этом каталоге размещаются команды, предназначенные для организации работы с системной информацией.

Следующий каталог – TMP – каталог, предназначенный для размещения "мусора"(временные файлы операционной системы). Данные временные файлы сохраняются на период работы системы. После перезагрузки системы временные файлы в этом каталоге могут быть удалены.

MNT – каталог, предназначен для установления связи файловой системы с файловыми системами, находящимися на других устройствах. Под другими устройствами подразумевается как другие дисковые разделы, так и реальные устройства.

DEV – каталог устройств – предназначен для хранения файлов устройств, то есть тех файлов, которые подтверждают регистрацию возможности работы системы с тем или иным устройством.

Каталог USR содержит информацию, предназначенную для обеспечения работы пользователей. В этом каталоге находятся следующие каталоги. Каталог LIB – каталог для размещения прикладных систем, предназначенных для организации работы пользователей. Каталог INCLUDE содержит файлы, которые используются как файлы прототипов и используем в командах препроцессора языка C в командах include. Каталог BIN содержит файлы, реализующие локализацию системы для конкретного пользователя. То есть в этом каталоге размешаются реализации команд, которые являются не штатными (которые поступают вместе с системой), а которые были сделаны или модифицированы конкретной организацией или пользователем. В каталоге USER обычно размещаются каталоги, которые ассоциируются с домашними каталогами зарегистрированных пользователи. В разных реализациях UNIX-системы могут быть вариации (не обязательно сами каталоги будут находиться в указанном каталоге USER, они могут находиться на примонтированном устройстве).

Пусть есть некоторый файл в файловой системе. Он исполняемый. То есть мы с использованием этого файла создали процесс, который начал работать. При этом процесс может быть не весь загружен в оперативную память. Тогда, если удалить указанный файл, то ничего страшного не произойдёт, так как при создании процесса содержимое исполнительного файла копируется в системные структуры данных. То есть, в общем случае можно взять любой файл, запустить его как процесс, а после этого – удалить. Процесс будет работать нормально.

В Unix предусмотрена оптимизация для организации работы с файловой системой. В строчке passwd имеется поле, которое определяет порядок рассмотрения ката-

ФАКУЛЬТЕТ ВЫЧИСЛИТЕЛЬНОЙ МАТЕМАТИКИ И КИБЕРНЕТИКИ МГУ ИМЕНИ М.В. ЛОМОНОСОВА

логов при поиске исполняемых файлов. То есть, определяется приоритет, в котором будут рассматриваться файлы. Более того, интерпретатор команд при входе пользователя в свой аккаунт берет все каталоги, которые указаны в параметре pass, выбирает из них все исполняемые файлы в соответствии с приоритетом и создаёт единую хэш-таблицу. Далее, когда набирается имя name, интерпретатор анализирует, есть ли у его такие встроенные команды. Если нет, то он обращается к хэш-таблице, которая обеспечивает быстрый ответ, есть ли такой файл среди тех каталогов, которые мы хотим рассматривать как местоположение команд, или нет. Если такой файл есть, мы переходим к нему.

Модель версии system V

Рассмотрим модель файловой системы System V с точки зрения организации данных.

Системный раздел системного устройства операционной системы System V имеет следующую структуру. Все пространство блоков этого раздела разделено на три области.

Суперблок файловой системы содержит оперативную информацию о текущем состоянии файловой системы (то есть информация, которая позволяет организовывать работу с ресурсами файловой системы), а также данные о параметрах настройки.

Кроме того, в суперблоке размещается информация о допустимом количестве индексных дескрипторов, которые могут быть созданы в файловой системе. Соответственно, параметр размер файловой системы в блоках и количество индексных дескрипторов однозначно определяют структуру области (с какого места начинается область индексных дескрипторов, до какого места она располагается, с какого места начинаются блоки файлов).

Индексный дескриптор – специальная структура данных ФС, которая ставится во взаимно однозначное соответствие с каждым файлом. Область индексных дескрипторов – последовательность индексных дескрипторов, именованных по номеру индексного дескриптора.

Блоки – свободные, занятые под системную информацию, занятые файлами. То

есть область блоков – это область, в которой размещаются блоки файлов (то есть содержимое файлов) и системная информация.

Рассмотрим суперблок подробнее. Суперблок – структура фиксированного размера, которая при работе системы всегда находится в оперативной памяти. В суперблоке находятся параметры настройки файловой системы. Более того, в суперблоке имеются структуры данных, которые предназначены для координации работы с ресурсами файловой системы. Ресурсы – это блоки файлов и индексные дескрипторы. Первый ресурс, который находится в суперблоке – массив номеров свободных блоков.Это массив некоторого фиксированного размера, в котором перечислены свободные блоки файловой системы. Если этого массива не хватает, чтобы перечислить все свободные блоки, то один из элементов этого массива содержит в себе номер блока (которые расположен в области блоков), в котором находится продолжение списка свободных блоков. Если одного блока недостаточно, то снова один из элементов этого блока ссылается на другой блок, в котором список продолжается и так далее. Таким образом, используем связный список свободных блоков. Первый элемента этого списка – массив, находящийся в суперблоке, который, в свою очередь всегда находится в оперативной памяти. Указанный массив нужен для кеширования запросов к медленной внешней памяти, когда нам нужно работать с номерами свободных блоков.

Работа с массивами номеров свободных индексных дескрипторов

Также в суперблоке находится массив свободных индексных дескрипторов. Это массив предопределённого фиксированного размера, в котором находится перечень свободных индексных дескрипторов.

Когда нужен индексный дескриптор, забираем номер этого дескриптора из массива, а его место обнулфю. Когда дескриптор освобождается, записываем его в указанный массив.

Есть некоторое отличие от работы с массивом свободных блоков. Мы работаем со списком свободных индексных дескрипторов в количестве, равном размеру этого массива. То есть, если у нас освобождаются индексные дескрипторы, то при каждом освободившемся пытаемся записать номер в массив свободных индексных дескрипторов. Если в этом массиве нет свободного места, то мы "забываем"об этом свободном индексном дескрипторе. Обратно – когда забираем свободный индексный дескрип-

тор, чтобы создать файлы, обнуляем соответствующие позиции в массиве свободных индексных дескрипторов. Если при очередном запросе в массиве не находится номер (все обнулено), то далее смотрим на параметры, находящиеся в суперблоке. В суперблоке есть два значения – предельное количество индексных дескрипторов и количество занятых индексных дескрипторов. Если эти числа говорят о том, что ещё есть индексные дескрипторы, то операционная система запускает процесс формирования массива свободных индексных дескрипторов.

Освобождение ИД

- есть свободное место – номер → элемент массива
- нет свободного места – номер "забывается"

Запрос ИД

- поиск в массиве
    - массив пусто – обновление массива
    - массив не пустой – ОК

Индексные дескрипторы

Индексный дескриптор – это структура фиксированного размера, в которой размещается информация об актуальном содержимом файла. Внутренняя структура индексных дескрипторов отличается для разных типов файлов. Рассмотрим некоторые часто используемые типы.

В регулярных файлах в индексном дескрипторе указываются параметры: тип файла, параметры о владельце файла и правах доступа к файлу, количество ссылок на индексный дескриптор из каталога файловой системы. Последний параметр показывает количество имён, которые жестка ассоциированы с данным индексным дескриптором. Это поле показывает, занят индексный дескриптор, или свободен. Если в поле 0 – данный индексный дескриптор свободен. Если не ноль, то дескриптор занят, а значение поля есть количество имён, которые размещены по каталогам файловой системы, которые, в свою очередь, ассоциированы с данным индексным дескриптором.

В индексном дескрипторе находится информация о длине файла. Кроме того, в индексном дескрипторе размещается информация о блоках, которые принадлежат

данному файлу. Соответственно, адресациям блоков устроена следующим образом. Это массив из тринадцати элементов. В общем случае каждый элемент – это значение, в котором может размещаться номер блока. Для примера будем считать, что блок файловой системы содержит 512 байтов. Тогда адресация блоков в индексном дескрипторе устроена следующим образом. В индексном дескрипторе находится массив из тринадцати элементов. Первые десять из них – номера первых десяти блоков файла. Одиннадцатый элемент содержит номер блока, в котором размещены следующие блоки файла. В данном примере в таком блоке может размещаться $\frac{512}{4} = 128$ номеров блоков. Если файл превосходит это количество, то используется двенадцатый элемент. Этот двенадцатый элемент используется для организации двойной косвенности. В двенадцатом элементе находится номер блока, в котором находятся номера ещё 128 блоков, в которых (в каждом из этих 128 блоков) 128 блоков файлов. Последний, тринадцатый элемент, реализует трехуровневую косвенность – в нем находится номер блока, в котором находится список из 128 блоков, в которых ссылки на блоки. То есть на третьем уровне имеем номера блоков файлов.

Минимальное количество накладных расходов при работе с блоками файлов в этой архитектуре равно нулю, а максимальное – три. Для данного примера предельное количество блоков, которое может быть в файле, равно $10 + 128 + 128^2 + 128^3$. Соответственно, предельный размер в байтах равен $(10 + 128 + 128^2 + 128^3) \cdot 512$.

Файл каталог

В System V была простейшая структура файлов каталогов – запись фиксированного размера. В записи два поля фиксированного размера. Первое поле: номер индексного дескриптора. Второе поле : имя файла. В операционной системе System V было существенное ограничение на длину используемого имени файла.

Структура каталога имеет следующие особенности. Первые две записи предопределенного назначения. Первая запись – в поле имени находится имя ".". В поле "номер индексного дескриптора"находится индексный дескриптор самого каталога. Вторая запись каждого каталога: в поле имени ".. в поле "номер индексного дескриптора"индексный дескриптор родительского каталога. Для корневого каталога обе эти записи имеют одинаковый номер индексного дескриптора – 1.

ФАКУЛЬТЕТ
ВЫЧИСЛИТЕЛЬНОЙ МАТЕМАТИКИ И
КИБЕРНЕТИКИ МГУ ИМЕНИ М.В.
ЛОМОНОСОВА

Достоинства и недостатки файловой системы System V

Достоинства файловой системы System V

- Оптимизация в работе со списками номеров свободных индексных дескрипторов и блоков
- Организация косвенной адресации блоков файлов

Отметим, что индексный дескриптор – это структура, которая хранится в оперативной памяти для всех открытых файлов. То есть операционная система всегда может оперативно работать с информацией индексных дескрипторов открытых файлов. В индексном дескрипторе организована косвенная организация блоков. Однако у данного решения есть минус – на каждый обмен с дальними блоками нарастают накладные расходы в виде дополнительных обменов.

Недостатки файловой системы System V

- концентрация важной информации в суперблоке
- проблема надежности
- фрагментация файла по диску
- ограничения на возможную длину имени файла

Из-за того, что важная информация хранится на суперблоке, необходимо регулярно сбрасывать информацию суперблока на диск. В противном случае возможна серьёзная потеря данных.

Пробьлема надежности: ткая архитектура модет способствовать разрушению файловой системы (так как есть дополнительная регулярная нагрузка на обоасть диска, в которй находится суперблок).

Список свободных способствует тому, что блоки файлов могут размещаться в произвольном порядке и этот порядок может быть фрагментирован по всему пространству, выделенному на хранение блоков.

Возможное решение: файловая система FFS BSD.

- Стратегия размещения

- Внутренняя организация блоков
- Выделение пространства для файла
- Структура каталога FFS

Лекция 24. Управление внешними устройствами.

Управление внешними устройствами

Рассмотрим историческое развитие проблем организации работы с внешними устройствами. У нас есть центральный процессор, оперативное запоминающее устройство и внешнее устройство. Имеет смыл обсуждать проблему взаимодействия с внешними запоминающими устройствами (то есть с стройками, которые обеспечивают передачу информации из оперативного запоминающего устройства на внешнее или наоборот). Для решения этой задачи возможные разные мелели.

1. **Непосредственное управление**внешними устройствами центральном процессором
2. **Синхронное управление** внешними устройствами использованием контроллеров внешних устройств
3. **Асинхронное управление** внешними устройствами с использованием **контроллеров внешних устройств**
4. Использование **контроллера прямого доступа** к памяти (DMA) при обмене
5. Управление внешними устройствами с использованием **процессора или канала ввода (вывода)**

Первая модель отпросится к самым первым компьютерам, ее суть заключается в том, что в системе команд процессора имелся ряд специальных команд, обеспечивающих обращение и взаимодействие с внешними запоминающими устройствами. Посредством исполнения этих команд можно было обеспечить управление этим устройством. Чем сложнее устройство, тем сложнее состав команд. Обмен совершался между оперативной памятью и внешним запоминающим устройством. Соответственно, кроме команд управления внешним запоминающим устройством в программе должны были присутствовать команды, которые брали те данные, которые мы хотим записать на внешнее запоминающее устройство, и переносили на специальные буфера, которые ассоциированы с внешним запоминающим устройством. После выполнения соответствующих команд с этих буферов происходила запись. Обратно – при считывании – данные попадали в буфер, после чего они переносились программой в оперативную память. То есть, непосредственный обмен связан с выполнением команд, которые занимались управлением внешним устройством и команд, которые

занимались переносом данных из оперативной памяти во внешнее устройство и обратно.

Понятно, что в случае первой модели обмены происходили синхронно. Если мы подаем команду, которая требует достаточной длительности выполнения, процессор блокируется и ожидает завершения действия. При этом наблюдались максимальные накладные расходы: процессор должен был выполнять код, обеспечивающий управление устройством и процессор должен был реализовывать код, который должен был обеспечивать перемещение данных, участвующих в обмене от устройства в оперативную память и обратно.

Во второй модели появляются контроллеры внешних запоминающих устройств. Контроллер – уродство, которое стояло между процессором и внешним запоминающим устройством и могло выполнять какие-то минимальные действия. В этом случае в рамках этого контроллера мы могли реализовать минимальную обработку запросов по обмену. Тогда нагрузка на процессор с точки зрения команд по управлению внешним устройством, которые должны были выполняться, сократилось. При этом управление было синхронным. То есть, если поступал заказ на действие по управлению обменом, то процессор блокировался и ожидал завершения отработки этого заказа.

Следующая модель связана с появлением асинхронного управления. То есть, с помощью специальной программы процессора мы можем задать заказ на выполнение какого-то действия, и после этого не будет происходить блокировка процессора, то есть можно продолжить выполнение последующих команд. Соответственно, информация о том, что заказ выполнен, передавалась посредством прерывания. При этом оставалась проблеме переноса данных из оперативной памяти в регистр, связанной с устройством и обратно. Это порождает накладные расходы.

Четвертая модель связана с появлением контроллеров, которые называются DMA – контроллеры прямого доступа к оперативной памяти. Эти контроллеры подключались к процессору, устройству и оперативной памяти. При заказе на обмен через контроллер результатом обмена было выполнение чтения или записи и осуществление соответствующего чтения или записи переноса данных в оперативную память или из оперативной памяти.

Пятая модель – появление и развитие процессоров или каналов ввода (вывода).

DMA контроллеры получили развитие и в их функции были добавлены возможности по обработке и коррекции ошибок кешированию информации и прочее.

Программное управление внешними устройствами

В основе архитектуры программного управление находится аппаратура. На вышестоящих уровнях находятся программы обработки прерываний, на основе которых строятся программы драйверов физических устройств, на основе которых можно строить драйверы логических устройств.

Цели и задачи, которые решаются компонентами операционной системы, занимающимися управлениями внешними устройствами:

- унификация программных интерфейсов доступа к внешним устройствам (унификация наименования, абстрагирование от свойств конкретных устройств)
- обеспечение конкретной модели синхронизации при выполнении обмена (синхронный, асинхронный обмен)
- обработка возникающих ошибок (индикация ошибки, локализация ошибки, попытка исправления ситуации)
- буферизация обмена
- обеспечение стратегии доступа к устройству (распределительный доступ, монопольный доступ)
- планирование выполнения операций обмена

Планирование дисковых обменов

Пусть имеется некоторый жёсткий диск, у которого имеется позиционирование головки в некоторый точке. Пусть есть очередь запросов к дорожкам. Как можно обработать этот набор запросов? Простейшая модель, позволяющая решить ланную задачу – FIFO, то есть в порядке поступления.

Есть класс алгоритмов, который называется **жадные алгоритмы**. **Жадный алгоритм** – итерационный алгоритм, который на каждом шаге пытается выбрать наиболее оптимальное решение. При этом жадный алгоритм не всегда самый оптимальный. Shortest Service Time First – "жадный"алгоритм – на каждом шаге поиск

обмена с минимальным перемещением.

Другие примеры планирования обменов – алгоритмы, основанные на приоритетах, то есть такие, что цепочка заказов на обмен разбирается на подцепочки, так или иначе связанные с какими-то приоритетами. Приоритеты могут быть типа задач, типа пользователей и другие. При этом есть проблема с корректностью выбора приоритетов и раздачей приоритетов.

Также существует алгоритм сканирования. Идея заключается в том, что мы сначала движемся в одну сторону до упора устройства, потом в другую сторону до упора устройства. Путь перемещения головки в данном случае детерминированный, но возможна ситуация, когда процесс зависает на какой-то одной дорожке, с которой производится интенсивная работа.

Альтернативой является многошаговое сканирование. То есть очередь запросов на обмен разделяется на подочереди по какому-то признаку. Далее определяется приоритет этих очередей. В соответствии с приоритетом мы обрабатываем эти подочереди в некотором циклическом процессе.То есть, когда мы берет подочередь на обработку, мы блокируем её обновление.

RAID системы

Важным компонентом является управление дисковыми системами(блок-ориентированными системами). Одной из прорывных технологий в этой области являются RAID-системы.

RAID (Redundant Array of Independent (Inexpersive) Disks)– избыточный массив независимых (недорогих) дисков.

RAID система – это технология, обеспечивающая объединение набора физических дисковых устройств, рассматриваемых операционной системой, как единое дисковое устройство (данные распределяются по физическим устройствам, образуется избыточная информация, используемая для контроля и восстановления информации).

Обмен с RAID-системой осуществляется в терминах полос. Полоса – блок для RAID-системы, то есть это порция данных фиксированного размера, которая осуществляет обмен с RAID-устройством. RAID-устройства могут использоваться раз-

личных целей, например для организации или создания устройств большого или свехбольшого обмена.

Все устройства, которые функционируют в RAID, считаются независимыми, поэтому обмены можно запускать на каждом из устройств параллельно(напоминает расслоение памяти). Это позволяет увеличивать скорость выполнения обмена.

За счет возможной избыточности хранения информации может обеспечивать надёжность хранения информации, то есть возможность восстановления информации как в оперативном режиме, так и в режиме, требующем определённого времени.

Функция RAID 0 – объем и скорость доступа. В RAID конфигурации 0 объем дисковых пространств, составляющих RAID объединяется, а за счет того, что каждое устройство может работать параллельно и независимо, скорость увеличивается. Без избыточной информации.

RAID 1 – система зеркалирования. RAID 1 создает два комплекта устройств. Информация, которая записывается на один комплект, дублируется на второй. 100% избыточность, нет временных потерь. Наиболее часть используется в системах, в которых требудется надежность функционирования. Как и RAID 0 является независимым устройством. Могут реализовываться программно посредством драйверов ОС.

RAID 2 и RAID 3 – специализированные системы, обладающие синхронизированными головками. Полосы короткие, имеют системы избыточности, обеспечивающие восстановление информации.

RAID 2 – системы, построенные на использовании кодов Хэмминга, может исправлять одинарные ошибки и обнаруживать двойные.

RAID 3 – система с четностью с чередующимися битами. На несколько содержательных устройств приходится одно устройство для хранения контрольной информации. В случае гибели одного из содержательных устройств, можно восстановить его содержание по контрольной информации.

Недостаток RAID 2 и RAID 3: диски с контрольной информацией получают максимальную нагрузку, что приводит к быстрому выходу из строя устройства. Последующие RAID предназначены для упрощения этой схемы.

RAID 4 – уход от специализированных дисковых устройств, уход от синхронных

головок. Осталась проблема перегрузки контрольного диска.

RAID 5 – распределенная четность. Циклическое распределение контрольных блоков по всем устройствам.

RAID 6 – двойное резервирование контрольных блоков. Циклическое распределение четности с использованием двух схем контроля: N+2 дисков.

Работа с внешними устройствами в операционной системе UNIX

1. файлы устройств, драйверы
   (a) файлы устройств
   (b) системные таблицы драйверов устройств
   (c) ситуации, вызывающие обращение к функциям драйвера
   (d) включение (удаление) драйверов в сситеме
2. организация обмена данных с файлами
3. буферизация при блок-ориентированном обмене
4. борьба со сбоями

Файлы устройств, драйверы Все устройства, с которыми легально работает система, регистрируются за счет регистрации своих драйверов и специальных файлов устройств, ассоциированных с драйверами.

Файл устройств – специализированный тип файла, не имеющий блоков. Вся информация файла устройств находится в индексном дескрипторе.

1. Иерархия драйверов
2. Два типа файлов устройств :
   (a) Байт-ориентированные устройства – это устройства, обмен с которыми может осуществляться произвольными порциями байтов. Пример: оперативная память.
   (b) Блок-ориентированные устройства – устройства, которые обеспечивают взаимодействие порциями фиксированного размера, блоками. Пример: диск.

   Одно и то же физическое устройство может представляться и как байт-ориентированное, и как блок-ориентированное устройство. Так может оперативная память.

Файлы устройств

- Содержимое файлов устройств размещается исключительно в соответствующем индексном дескрипторе.
- В индексном дескрипторе файла устройств кроме регистрационной информации может размещаться информация, относящаяся к типу данных. Она состоих из трех элементов. Структура ИД файла устройства:
  - "Старший номер"(major number) устройства. Это число говорит о том, что данное устройство ассоциируется с драйвером, зарегистрированным в системе под этим номером.
  - Тип устройства: байт-ориентированное или блок-ориентированное устройство. В системе две таблицы для регистрации драйверов соответветствующего типа. Старший номер и тип устройства определяют, с какой таблицей связываться, где старший номер – номер записи в этой таблице.
  - "Младший номер"(minor number) устройства
- Системные таблицы драйверов устройств:
  - bdevsw
  - cdevsw

Определение. *Коммутатор устройства* – каждая запись в таблице.

Файлы устройств

Системные таблицы драйверов устройств

- Запись таблицы – **коммутатор устройства**

  Определение. *Коммутатор* – структура фиксированного размера, которая содержит перечень точек входа в соответствующий драйвер.

  Эти точки входа – это набор системных вызовов, которые предоставляются для взаимодействия с файловой системой.

- Типовой набор точек входа в драйвер:
  - $\beta open()$, $\beta Close()$
  - $\beta read()$, $\beta write()$
  - $\beta ioctl()$

ФАКУЛЬТЕТ ВЫЧИСЛИТЕЛЬНОЙ МАТЕМАТИКИ И КИБЕРНЕТИКИ МГУ ИМЕНИ М.В. ЛОМОНОСОВА

– $\beta intr()$
– $\beta strategy()$

Ситуации, вызывающие обращение к функциям драйвера

- старт системы, определение ядром состава доступных устройств
- обработка запроса ввода/вывода
- обработка прерывания, связанного с данным устройством
- выполнение специальных команд управления

Организация обмена данными с файлами

Для поддержания структуры в системе имеется три таблицы:

- Таблица индексных дескрипторов открытых файлов (размещается в памяти ядра ОС)
- Таблица файлов (размещается в памяти ОС)
- Таблица открытых файлов. Создается системой на ресурсах процесса. Размер таблицы – параметр настройки ОС. Он регламентирует предельное количество файловых драйверов или файловых дескрипторов, которые может иметь процесс единовременно(предельное количество открытых файлов)

Пример. Пусть есть некий процесс, у которого есть таблица открытых файлов. В этой таблице имеется ссылка на запись в таблице файлов. В таблице файлов кроме всего прочего размещается указатель чтения/записи и счетчик ссылок извне на эту запись. Изначально там 1. Процесс, который открыл файл name, сделал fork $\to$ появился клон исходного процесса, в котором будет такая же запись для открытого файла, которая будет ссылаться на ту же запись в таблице файлов. В индекс добавляется единица, ибо на эту запись ссылаются из двух таблиц файлов. При первом открытии (open), эта цепочка продолжается дальше. Сделали open name, разместили его в таблице открытых файлов, нашли место и разместили его в таблице файлов, ищем место в таблице индексных дескрипторов открытых файлов. Ищем запись с индексным дескриптором файла name. Если ее находим, устанавливаем ссылку в таблице файлов, а счетчик кратности увеличивается на 1. Если после fork запустили еще один процесс, для него будет сформирована отдельная новая запись в таблице файлов.

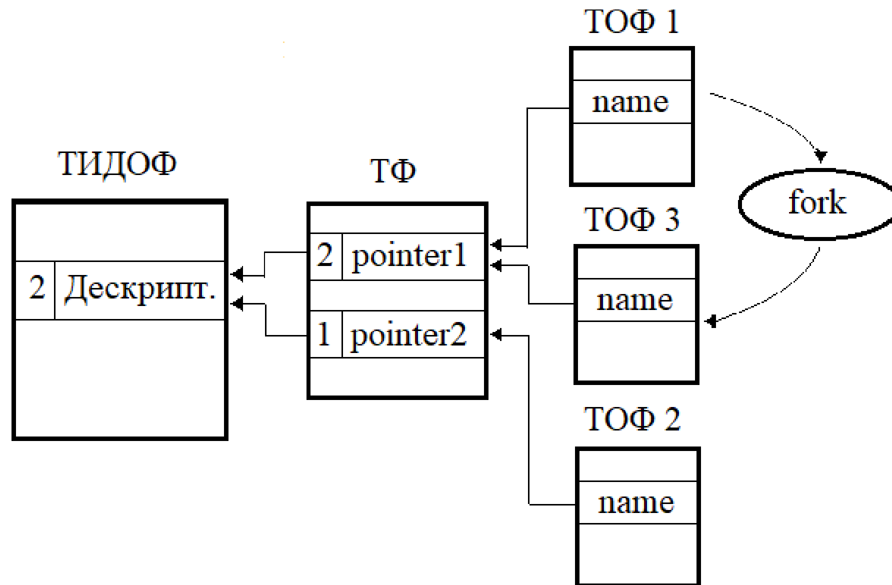

Рис. 24.1. Пример

Вывод. Работа с файлами осуществляется по двум моделям:

1. Модель работы с файлами, открытыми через open, в этом случае у него свой указатель чтения/записи.

2. Работа с файлами, которые получились в процесса в результате наследования. у них общий указатель чтения/записи.

Буферизация при блок-ориентированном обмене Это КЭШ, который организован в терминах блоков и КЭШ, который организован в оперативной памяти операционной системы. Его задача – минимизация реальных обращений, борьба со сбоями, модель проверки целостности.

ФАКУЛЬТЕТ
ВЫЧИСЛИТЕЛЬНОЙ
МАТЕМАТИКИ И
КИБЕРНЕТИКИ
МГУ ИМЕНИ
М.В. ЛОМОНОСОВА

teach-in
ЛЕКЦИИ УЧЕНЫХ МГУ