

МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ УКРАИНЫ

Национальный технический университет  
“Харьковский политехнический институт”

А.С.Дервянко, М.Н.Солощук

## **ОПЕРАЦИОННЫЕ СИСТЕМЫ**

### **Часть I**

**Построение и функционирование операционных систем**

Учебное пособие

Харьков 2002

ББК 32 - 973.018

УДК 681.3.06

Рецензенти: **Г.І.Загарій**, д-р техн. наук, проф., завідувач кафедри “Автоматика і комп’ютерні системи управління” Української державної академії залізничного транспорту; **З.В.Дудар**, канд. техн. наук, проф., завідувач кафедри “Програмне забезпечення ЕОМ” Харківського національного університету радіоелектроніки.

Интеллектуальная собственность НТУ “ХПИ”. Все права защищены.

**Операционные системы:** Деревянко А.С., Солощук М.Н. Учебное пособие. - Харьков: НТУ “ХПИ”, 2002. - ...с.

Представлена концепция операционной системы как набора программных модулей, выполняющих планирование аппаратных и программных ресурсов. Рассмотрены дисциплины и алгоритмы планирования для различных ресурсов при различных задачах, стоящих перед системами. Предназначено для студентов и специалистов направлений “Компьютерные науки” и “Компьютерная инженерия”

## ISBN

Подано концепцію операційної системи як набору програмних модулів, які виконують планування апаратних та програмних ресурсів. Розглянуто дисципліни та алгоритми планування для різних ресурсів при різних задачах, що стоять перед системами. Призначається для студентів та спеціалістів напрямків “Комп’ютерні науки” та “Комп’ютерна інженерія”

Илл. - 55, библиогр. - 46 назв.

© А.С. Деревянко, М.Н. Солощук, 2002

© Национальный технический университет “ХПИ”, 2002

# **Содержание**

## **Введение**

### **Глава 1. Основные понятия**

- 1.1. Операционная система с точки зрения системного программиста
  - 1.2. Классификация и предварительный обзор операционных систем
  - 1.3. Точка зрения пользователя
  - 1.4. Аппаратная архитектура и поддержка ОС
  - 1.5. Ядро и процессы
  - 1.6. Архитектурные концепции операционных систем
- Контрольные вопросы

### **Глава 2. Планирование процессов**

- 2.1. Дисциплины планирования – требования, показатели, классификация
  - 2.2. Базовые дисциплины планирования
  - 2.3. Планирование процессов в реальных системах
  - 2.4. Другие уровни планирования
- Контрольные вопросы

### **Глава 3. Управление памятью**

- 3.1. Виртуальная и реальная память
  - 3.2. Фиксированные разделы.
  - 3.3. Односегментная модель
  - 3.4. Многосегментная модель
  - 3.5. Страничная модель
  - 3.6. Сегментно-страничная модель
  - 3.7. Плоская модель
  - 3.8. Одноуровневая модель
- Контрольные вопросы

### **Глава 4. Порождение программ и процессов**

- 4.1. Компиляция
  - 4.2. компоновка и загрузка
  - 4.3. Цикл жизни процесса
  - 4.4. Нити
- Контрольные вопросы

### **Глава 5. Монопольно используемые ресурсы**

- 5.1. Свойства ресурсов и их представление
- 5.2. Обедаяющие философы
- 5.3. Тупики: предупреждение, обнаружение, развязка
- 5.4. Бесконечное откладывание

Контрольные вопросы

## Глава 6. Управление вводом-выводом

- 6.1. Виртуализация устройств и структура драйвера
- 6.2. Интерфейсы устройств
- 6.3. Управление устройствами
- 6.4. Примеры драйверов устройств
- 6.5. Поток и многоуровневые драйверы
- 6.6. Интерфейс процесса
- 6.7. Буферизация

Контрольные вопросы

## Глава 7. Файловые системы

- 7.1. Иерархическая модель файловой системы
- 7.2. Логическая организация файлов. Интерфейсы
- 7.3. Логическая файловая система. Каталоги
- 7.4. Логическая файловая система. Системные вызовы
- 7.5. Базовая файловая система
- 7.6. Физическая структура файлов
- 7.7. Пример
- 7.8. Целостность данных и файловой системы
- 7.9. Загружаемая файловая система

Контрольные вопросы

## Глава 8. Параллельное выполнение процессов

- 8.1. Постановка проблемы
- 8.2. Взаимное исключение запретом прерываний
- 8.3. Взаимное исключение через общие переменные
- 8.4. Команда testAndSet и блокировки
- 8.5. Семафоры
- 8.6. "Производители-потребители"
- 8.7. Конструкции критических секций в языках программирования
- 8.8. Мониторы
- 8.9. "Читатели-писатели" и групповые мониторы
- 8.10. Примитивы синхронизации в языках программирования
- 8.11. Рандеву

Контрольные вопросы

## Глава 9. Системные средства взаимодействия процессов

- 9.1. Скобки критических секций
- 9.2. Виртуальные прерывания или сигналы
- 9.3. Модель виртуальных коммуникационных портов
- 9.4. Общие области памяти
- 9.5. Семафоры
- 9.6. Программные каналы
- 9.7. Очереди сообщений

Контрольные вопросы

## Глава 10. Защита ресурсов

- 10.1. Общие требования безопасности
- 10.2. Объектно-ориентированная модель доступа и механизмы защиты
- 10.3. Представление прав доступа
- 10.4. Дополнительные возможности

Контрольные вопросы

## Глава 11. Интерфейс пользователя

- 11.1. Командный язык и командный процессор
- 11.2. Командные файлы и язык процедур
- 11.3. Проблема идентификации адресата
- 11.4. WIMP-интерфейс

Контрольные вопросы

Заключение

Список литературы

## Введение

Предлагаемое вниманию читателей учебное пособие написано по материалам курсов "Системное программное обеспечение" и "Системное программирование и операционные системы", читаемых студентам направлений "Компьютерные науки" и "Компьютерная инженерия" Национального политехнического университета "ХПИ", а также слушателям Межотраслевого института повышения квалификации при НТУ "ХПИ". Изложение этих курсов сопровождается неизменным интересом слушателей и неизменной нехваткой учебной литературы. Дело в том, что курсы базируются на общих концепциях, сложившихся в начале 70-х годов. Произошедшая в середине 80-х "персональная революция" создала ошибочное впечатление об устарелости этих концепций и обусловила перерыв в издании учебной литературы, рассматривающей их. Так, последнее известное нам "раннее" издание такого рода на русском языке датировано 1989 годом [26]. Однако последующее совершенствование средств вычислительной техники и ее программного обеспечения показало, что эти концепции отнюдь не устарели, но продолжают применяться и развиваться. Старые издания не могут удовлетворить растущего интереса студентов и специалистов к этой теме, во-первых, потому, что они уже стали библиографической редкостью, а во-вторых, потому, что в них, естественно, не рассматриваются современные версии ОС и те (пусть и немногие) новые концепции, которые появились в последние годы. Два же известных нам современных издания [7, 22] посвящены рассматриваемой нами теме лишь отчасти, содержат далеко не полные обзоры и также не могут удовлетворить спрос в полной мере. Мы

надеемся, что предлагаемое издание в какой-то мере уменьшит этот информационный дефицит.

В первой части данного учебного пособия мы не привязываемся к какой-либо конкретной ОС, рассматривая лишь общие принципы построения и функционирования ОС. Вторая же часть посвящена тому, как рассмотренные принципы реализованы в конкретных современных системах.

Системные вызовы мы по возможности именовали в соответствии с традициями, сложившимися в ОС Unix и зафиксированными в стандарте POSIX, однако, с легкостью отступали от этих традиций там, где это нам представлялось необходимым. В описании алгоритмов и данных мы ориентировались на язык программирования C, однако, опять-таки отступали от синтаксических правил языка там, где строгое следование им вело, по нашему мнению, к излишней конкретизации.

В тексте книги шрифтом Courier New выделены фрагменты, представляющие собой лексические конструкции языков программирования или командных языков и элементы формальной математической записи.

Авторы посвящают эту работу памяти своего друга и коллеги почетного доктора НТУ "ХПИ" Хартмуда Штира (IBM development Laboratory, Boblingen, Germany).

# Глава 1. Основные понятия

## 1.1. Операционная система с точки зрения системного программиста

Операционная система (ОС) есть набор программ, которые распределяют ресурсы процессам.
--

Приведенная выше формулировка является ключевой для понимания всего курса. Прежде, чем мы ее раскроем, дадим определение входящих в нее терминов. Ресурс – "средство системы обработки данных, которое может быть выделено процессу обработки данных на определенный интервал времени" [8]. Иными словами: ресурс – это все те аппаратные и программные средства и данные, которые необходимы для выполнения программы. Ресурсы можно подразделить на первичные и вторичные. К первой группе относятся те ресурсы, которые обеспечиваются аппаратными средствами, например: процессор, память – оперативная и внешняя, устройства и каналы ввода-вывода и т.п.; ко второй – ресурсы, порождаемые ОС, например, системные коды и структуры данных, файлы, семафоры, очереди и т.п. В последнее время в связи с развитием распределенных вычислений и распределенного хранения данных все большее значение приобретают такие ресурсы как данные и сообщения.

В [12] приведено около десяти определений термина "процесс", из которых автор выбирает: "программа в стадии выполнения". Это определение близко к тому, что интуитивно понимают под "процессом" программисты, но оно не является строгим. Более строгое определение



процесса, которое дает терминологический стандарт, представляется нам гораздо более удачным, поэтому ниже мы приводим его полностью.

"Процесс обработки данных – система действий, реализующая определенную функцию в системе обработки информации и оформленная так, что управляющая программа данной системы может перераспределять ресурсы этой системы в целях обеспечения мультипрограммирования.

Примечания:

1. Процесс характеризуется состояниями, которые определяются наличием тех или иных ресурсов в распоряжении процесса и, следовательно, возможностью фактически выполнять действия, относящиеся к процессу.

2. Перераспределение ресурсов, выполняемое управляющей программой, влияет на продолжительность процесса обработки данных, но не на его конечный результат.

3. Процесс оформляют с помощью специальных структур управляющих данных, которыми манипулирует управляющий механизм.

4. В конкретных системах обработки информации встречаются разновидности процессов, которые различаются способом оформления и составом ресурсов, назначаемых процессу и отнимаемых у него, и допускается вводить специальные названия для таких разновидностей, как, например, задача в операционной системе ОС ЕС ЭВМ" [8].

(В соответствии со сложившейся в литературе традицией мы часто будем употреблять термин "задача" как синоним термина "процесс".)

На примечания к определению процесса мы обратим внимание позже, а пока сосредоточимся на основной его части. С точки зрения ОС процесс – это "юридическое лицо", которое получает в свое распоряжение ресурсы. Процесс может иметь сложную структуру, но его составные части либо оформляются как отдельные процессы и тогда предстают перед ОС как независимые от процесса-родителя "юридические лица", либо

используют ресурсы от имени всего процесса и тогда они "невидимы" для ОС. (Промежуточный случай – нити – мы рассматриваем в главе 4)

Такой взгляд на разработку и анализ ОС сложился в конце 60-х – начале 70-х годов, в значительной степени под влиянием ОС Unix [9, 33], в которой принцип процессов и ресурсов реализован наиболее последовательно и изящно. Большое количество изданий, посвященных ОС и отражающих как эмпирический (например, [12, 17-19, 36]), так и аналитический (например, [1, 2, 16]) подходы, разделяет именно такой взгляд. Следование принципу процессов – ресурсов позволяет структурировать изучение ОС в виде таблицы, приведенной на рисунке 1.1. Столбцами этой таблицы являются классы ресурсов, которыми управляют ОС, а строками – конкретные ОС.

	Процессорное время	Память	Монопольные ресурсы	Ввод-вывод	Внешняя память и файлы	Взаимодействие процессов	Взаимодействие с пользователем	и т.д.
MS DOS	✓	✓	✓	✓	✓	✓	✓	✓
Windows 3.x	✓	✓	✓	✓	✓	✓	✓	✓
Windows 9x	✓	✓	✓	✓	✓	✓	✓	✓
OS/2	✓	✓	✓	✓	✓	✓	✓	✓
Windows NT	✓	✓						

Рисунок 1.1 Операционные системы и ресурсы

В идеале исчерпывающее изложение курсов "Системное программное обеспечение ЭВМ" и "Операционные системы" должно привести к заполнению всех клеток этой таблицы, но в данном учебном курсе мы сосредоточили внимание на изучении "структуры записи" (строки) этой таблицы. Владение этой структурой позволит специалисту

самостоятельно заполнить пробелы в таблице и при необходимости дополнить таблицу новыми строками. В связи с конкурентной борьбой на рынке программных продуктов описания современных ОС, появляющиеся в печати, по большей части акцентируют внимание на тех свойствах, которые придают системе "товарный вид", хотя и необязательно определяют фундаментальные возможности и эффективность системы. Понимание таких возможностей вооружает специалиста инструментом для сравнительного анализа различных ОС по общим объективным критериям.

Попытку "эскизного" заполнения таблицы на рисунке 1.1 мы делаем во второй части этой книги.

## **1.2. Классификация и предварительный обзор операционных систем**

В изданиях, упомянутых выше, классификация совмещается с историческим обзором, показывающим, как со временем увеличивались ресурсы вычислительных систем и соответственно усложнялись функции управления ими. Наряду с этим, нам представляется интересным провести классификацию также и в одном (сегодняшнем) временном срезе.

Мы будем классифицировать ОС по количеству пользователей и количеству задач (процессов), одновременно управляемых системой. Чем вызывается стремление увеличить эти показатели?

Доводы за многопользовательский режим составляют две группы. Во-первых, возьмем на себя смелость утверждать, что сегодня персональный компьютер возможен только как игрушка.

Профессиональный программист или пользователь ЭВМ не может сегодня работать на персональном компьютере – он может (и должен) работать на сколь угодно интеллектуальном персональном терминале в глобальной компьютерной сети. Естественно, что стоимость обработки данных в такой сети может быть существенно снижена при концентрации программ и данных, относящихся к одному, например, проекту или предприятию, в одном узле этой сети с обеспечением доступа к ним всех пользователей этой информации. Во-вторых, в 70-е годы состояние средств вычислительной техники и их программного обеспечения позволило специалистам вывести правило о том, что при линейном возрастании стоимости вычислительной системы ее возможности возрастают в квадрате [12]. В середине 80-х годов это правило было нарушено из-за значительного снижения стоимости ПЭВМ за счет их массового производства, но в настоящее время технологии производства компонентов больших ЭВМ (мейнфреймов) по стоимостному показателю сравнялись с ПЭВМ [31, 38, 40], и это правило вновь становится актуальным. Но более мощную вычислительную систему один пользователь будет просто не в состоянии загрузить – отсюда и необходимость в многопользовательском режиме.

Многозадачность (синоним: мультипрограммирование – "режим работы, предусматривающий поочередное выполнение двух или более программ одним процессором" [8]) при ее возникновении была обусловлена стремлением наиболее полно использовать ресурсы. При работе системы в пакетном режиме целью, к которой стремится ОС, является повышение пропускной способности – обслуживание как можно большего числа заданий в единицу времени. Поскольку аппаратная архитектура большинства вычислительных систем допускает параллельное функционирование центрального процессора и каналов ввода-вывода, очевидным представляется программное решение, использующее это

распараллеливание: один процесс выполняется на центральном процессоре, в то время как другой (другие) работает с каналом (каналами) ввода-вывода. С заменой перфокарточных и перфоленточных устройств ввода терминалами стал активно развиваться интерактивный режим.

Понятие "задание" (job) сменяется понятием "сеанс" (session). В отличие от задания, в котором исходные данные готовились до начала выполнения программы и вводились в ЭВМ вместе с программой, в сеансе эти данные вводятся уже в ходе выполнения, зачастую они просто не могут быть подготовлены заранее. Пока в одном сеансе происходит подготовка и ввод данных, система может обслуживать другие сеансы. Поскольку ввод данных, выполняемый оператором или пользователем, – процесс очень медленный, уровень мультипрограммирования (количество параллельно выполняемых процессов) в такой системе значительно повышается. При управлении ресурсами в интерактивном режиме на передний план выдвигается цель справедливого обслуживания: обеспечение минимальной дисперсии времени ответа системы на ввод данных пользователем и приемлемого времени ожидания ответа.

Разновидностью интерактивного режима можно считать вычисления в режиме клиент/сервер [34]. В этом режиме управление каким-либо ресурсом (например, базой данных) осуществляется отдельным процессом (возможно, и отдельным компьютером в сети) – сервером. Приложения-клиенты для получения доступа к ресурсу обращаются к серверу. При любой обработке данных имеются три основных уровня манипулирования данными, как показано на рисунке 1.2:

- хранение данных;
- бизнес-логика, т.е. выборка и обработка данных для нужд прикладной задачи;
- представление данных и результатов обработки конечному пользователю.

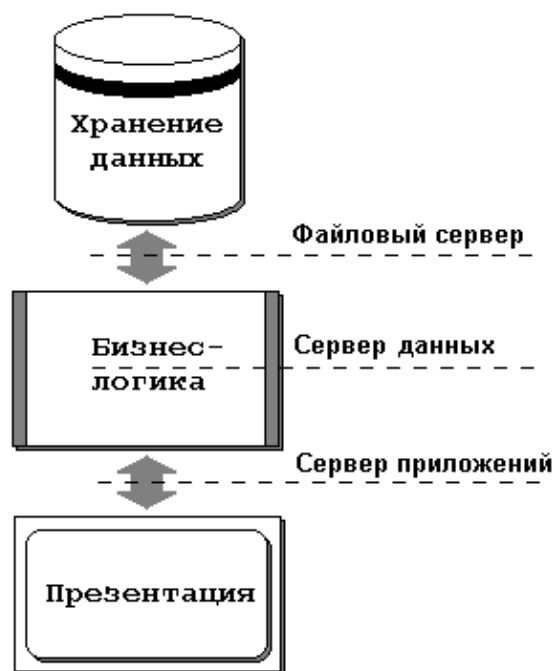


Рисунок 1.2 Уровни обработки и модели клиент/серверных вычислений

В вычислительных системах, построенных по персональной идеологии, все три функции в полной мере сосредоточены на одном компьютере.

При построении неперсональных систем выполняется перераспределение функций между компьютерами в сети.

Распределение функций манипулирования данными между клиентом и сервером может быть различным. Различные варианты распределения функций между сервером и клиентами образуют различные варианты архитектуры клиент/сервер (см. рисунок 1.2):

- если сервер выполняет только хранение данных, и при необходимости вся единица хранения данных (файл) пересылается клиенту, и всю дальнейшую работу с данными выполняет клиент, то это вариант файлового сервера;
- если на сервер возлагается выполнение одной из самых трудоемких функций логики приложения – выборки необходимых для обработки данных, то это вариант сервера данных;

- если вся логика приложений выполняется на сервере, а в клиентскую часть передаются лишь результаты обработки, то это вариант сервера приложений.

В любом из этих вариантов клиентские ОС работают в интерактивном режиме, обслуживая пользователей-операторов, а ОС сервера – тоже в интерактивном режиме, но пользователями для нее являются приложения-клиенты. Отличия режима клиент/сервер от обычного интерактивного скорее количественные, чем качественные: ОС сервера выполняет несколько более длинные последовательности процессорных команд без обращения к операциям ввода-вывода и несколько реже получает внешние прерывания. Поэтому дисциплины управления ресурсами в интерактивных и клиент/сервер ОС различаются не структурами алгоритмов, а их параметрами.

Сходные задачи стоят и перед системами реального времени, как правило, работающими в непосредственной связи (on-line) с объектом управления и выполняющими некоторые операции по управлению либо периодически, либо по требованию. Но в отличие от интерактивных или клиент/серверных ОС, для систем реального времени основной целью является обеспечение гарантированного времени ответа, ни в коем случае не превышающего некоторого критического значения.

Наконец, современная (и перспективная) модель вычислений предполагает разнесение всех трех уровней клиент/серверной архитектуры – клиент, сервер приложений, сервер данных – по разным ЭВМ. Функции клиента сводятся к презентации информации для конечного пользователя. Сервер приложений обеспечивает разнообразные вычислительные возможности; сервер данных, прежде всего, – хранение и выборку данных, хотя может выполнять и значительную часть их обработки. В условиях развития глобальных коммуникаций каждый клиент может получать обслуживание от многих серверов приложений, а каждый сервер

приложений – получать данные из многих источников, как показано на рисунке 1.3.

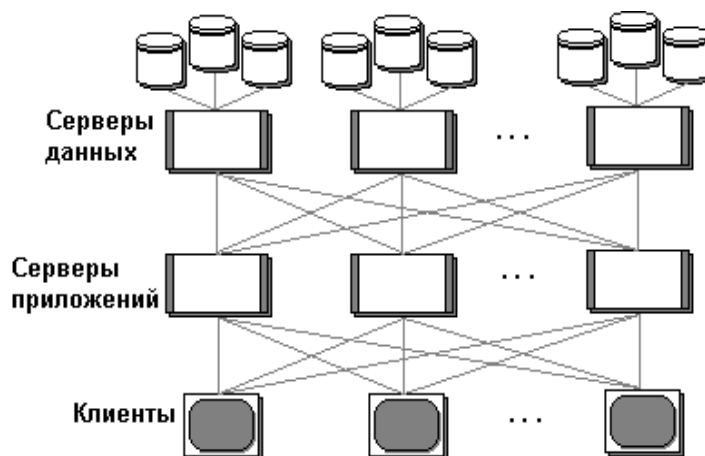


Рисунок 1.3 Трехуровневая архитектура клиент/сервер

Прогнозируется (см., например, [44]), что в ближайшие годы ПЭВМ должны будут существенно "потесниться" в роли клиента, уступив значительную часть этого ареала устройствам с ограниченными вычислительными возможностями (так называемым "тонким" клиентам), в том числе, и мобильным. Вычисления, таким образом, становятся все более сервер-центрическими, распределяясь между серверами приложений и серверами баз данных. При работе с мобильными клиентами и удаленными источниками данных получение обслуживания клиента у сервера приложений, а сервера приложений – у сервера данных может происходить и без установления непосредственной связи между клиентом и сервером, а состоять из послыки клиентом сообщения – запроса на обслуживание и получения им ответного сообщения с результатами выполнения запроса. В этом случае мы как бы возвращаемся к пакетному режиму, хотя и с иными характеристиками пакетов-заданий.

Хотя описанное нами развитие методов обработки данных происходит во времени, новый подход никогда полностью не отменяет предыдущие. В настоящее время в эксплуатации находятся



вычислительные системы с самым разным объемом ресурсов и с применением самых разных методов обработки информации.

Целью настоящего издания не является исчерпывающий обзор ОС, однако в тексте мы часто будем приводить примеры организации тех или иных функций в конкретных системах. В сумме эти примеры, рассредоточенные по разным главам, могут составить не исчерпывающее, но довольно полное представление о нескольких ОС. Поэтому в приводимой ниже классификации мы дадим вводную характеристику тем ОС, которые составляют наш "банк примеров". Некоторые приводимые нами характеристики ОС, возможно, будут непонятны начинающему читателю, их объяснение вы найдете в следующих главах настоящего пособия.

Простейшим является класс однозадачных однопользовательских систем. Их аппаратной платформой является IBM PC (XT, AT), ОС – **MS DOS**. Поскольку ресурсы такой системы весьма ограничены, ее рассмотрение не представляет для целей данного пособия существенного интереса.

Класс многозадачных однопользовательских систем начинается с тандема MS DOS + Windows, но настоящими ОС этого класса являются OS/2 и Windows 9x. Эти ОС работают на аппаратной платформе не ниже процессора Intel 80386, ресурсы, поддерживаемые такими ОС, – более мощные, управление ими усложняется. Вместе с тем в функции системы не входит защита ресурсов от других пользователей: в однопользовательской системе "украсть" ресурсы можно только у самого себя.

**Windows 1.x - 3.x** представляет собой надстройку над MS DOS, обеспечивающую управление виртуальной памятью (сегментную или сегментно-страничную – в зависимости от процессора – модель) и кооперативную многозадачность.

Операционные системы **OS/2** и **Windows 95/98/ME** – системы многозадачные, однопользовательские. (Хотя OS/2 позиционируется на рынке как серверная система, ядро ее продолжает оставаться однопользовательским.) Они обеспечивают вытесняющую многозадачность и работу с нитями, а также богатый набор средств взаимодействия процессов. В них используется 32-разрядная (плоская) модель памяти

Многозадачные многопользовательские системы в настоящее время эксплуатируются на ЭВМ, работающих в многопользовательском интерактивном режиме или выполняющих функции серверов в сетях, их современные аппаратные платформы – на базе серверов Intel-Pentium и RISC-процессоров. Управление ресурсами здесь усложняется не только из-за простого возрастания их объема, но и из-за изменения задач. Система исходит из "презумпции нечестности" пользователей – предположения о том, что любой процесс будет стремиться захватить как можно больше ресурсов в ущерб процессам других пользователей. ОС должна обеспечить справедливое распределение ресурсов между пользователями и их учет (возможно, для оплаты). Важной составляющей таких ОС является также обеспечение безопасности: защита программ и данных пользователя от их чтения или изменения или уничтожения другими пользователями.

Первым примером ОС такого класса, естественно, должна быть названа ОС **Unix**, которая существует и развивается с 1968 г. ОС Unix оказала огромное влияние на развитие концепций построения ОС, породила множество клонов (BSD Unix, Solaris, AIX, Linux и т.д.) и является основой стандартов на ОС.

**Windows NT** (начиная с версии 5 – Windows 2000) является полностью 32-разрядной ОС с объектно-ориентированной структурой и строится на базе микроядра.

Семейство вычислительных систем **AS/400** является результатом длительного эволюционного развития, начавшегося с IBM System/38 (1978 г.). По ряду идей и решений эволюционный ряд System/38 - AS/400 является лидером в развитии ОС. Среди особенностей, делающих эту систему интересным примером для нас, следует назвать: объектно-ориентированную ее структуру и архитектуру на базе микроядра, одноуровневую модель памяти, мощные средства защиты. Системное программное обеспечение AS/400 двухуровневое: нижний уровень выполняется Лицензионным Внутренним Кодом (LIC – Licensed Internal Code) и обеспечивает аппаратную независимость верхнего уровня, который составляет собственно ОС OS/400. AS/400 отличается значительной степенью системной интеграции и высоким уровнем системных интерфейсов.

Наконец, последний рассматриваемый нами класс – гигаресурсные (термин введен нами) системы. Являясь также многозадачными и многопользовательскими, они отличаются от предыдущего класса тем, что ресурсы, управляемые ими, на несколько порядков больше. Их аппаратной платформой являются мейнфреймы System/390 или ESA (Enterprise System Architecture) фирмы IBM, представляющие собой эволюционное развитие ряда System/360 – System/370. Современные мейнфреймы отличаются большим объемом возможностей, реализованных на аппаратном уровне, таких как мультипроцессорная обработка, средства создания системных комплексов, объединяющих несколько мейнфреймов, средства логического разделения ресурсов вычислительной системы, высокоэффективная архитектура каналов ввода-вывода, и т.д. Современные ОС ESA – VSE/ESA, VM/ESA, OS/390 представляют собой развитие работавших на System/360, System/370.

- **VSE/ESA** ориентирована на использование в конечных и промежуточных узлах сетей. Она функционирует на наименее мощных моделях мейнфреймов. VSE эффективно выполняет пакетную обработку и обработку транзакций в реальном времени.
- **VM/ESA** – гибкая интерактивная ОС, поддерживающая одновременное функционирование нескольких "гостевых" ОС на одной вычислительной системе.
- **OS/390** (в последней версии – z/OS) – основная ОС для применения на наиболее мощных аппаратных средствах. Она обеспечивает наиболее эффективное управление ресурсами при пакетном и интерактивном режимах и обработке в реальном времени, возможно совмещение любых режимов. Обеспечивает также комплексирование вычислительных систем, динамическую реконфигурацию ввода-вывода, расширенные средства управления производительностью.

### 1.3. Точка зрения пользователя

ОС есть набор программ, которые скрывают от пользователя детали управления оборудованием (hardware) и обеспечивают ему более удобную среду

Этот принцип иллюстрируется рисунком 1.4.



Рисунок 1.4 Операционная система, процессы, оборудование

Как видно из рисунка 1.4, ОС играет роль "прослойки" между процессами пользователей и оборудованием системы. (Под оборудованием понимаются, как правило, внешние устройства, но можно трактовать этот термин и шире, включая в него все первичные ресурсы). Процессы пользователей не имеют непосредственного доступа к оборудованию и, говоря шире, к системным ресурсам. Если процессу необходимо выполнить операцию с системным ресурсом, в том числе и с оборудованием, процесс выдает системный вызов. ОС интерпретирует системный вызов, проверяет его корректность, возможно, помещает в очередь запросов и выполняет его. Если выполнение вызова связано с операциями на оборудовании, ОС формирует и выдает на оборудование требуемые управляющие воздействия. Оборудование, выполнив операцию, заданную управляющими воздействиями, сигнализирует об этом прерыванием. Прерывание поступает в ядро ОС, которое анализирует его и формирует отклик для процесса, выдавшего системный вызов. Если выполнение системного вызова не требует операций на оборудовании, отклик может быть сформирован немедленно.

Управляющие воздействия и прерывания составляют интерфейс оборудования, системные вызовы и отклики на них – интерфейс процессов. В качестве синонима интерфейса процессов мы в соответствии со сложившейся в последнее время традицией часто будем употреблять

аббревиатуру API (Application Programm Interface – интерфейс прикладной программы).

Отделение процессов пользователя от оборудования преследует две цели.

Во-первых – безопасность. Если пользователь не имеет прямого доступа к оборудованию и вообще к системным ресурсам, то он не может вывести их из строя или монопольно использовать в ущерб другим пользователям. Обеспечение этой цели нуждается в аппаратной поддержке, рассматриваемой в следующем разделе.

Во-вторых – обеспечение абстрагирования пользователя от деталей управления оборудованием. Вывод на диск, например, требует сложного программирования контролера дискового устройства, однако, все пользователи используют для этих целей простое обращение к драйверу устройства. Более того, в большинстве систем имеются библиотеки системных вызовов, обеспечивающие API для языков высокого уровня (прежде всего – для языка C). Можно также говорить о том, что ОС интегрирует ресурсы: из ресурсов низкого (физического) уровня она конструирует более сложные ресурсы, которые, с одной стороны, сложнее (по функциональным возможностям), а, с другой, проще (по управлению) низкоуровневых.

## **1.4. Аппаратная архитектура и поддержка ОС**

Существует несколько различных определений того, что следует считать аппаратной архитектурой ЭВМ, каждое из таких определений "работает" для определенного класса задач. Мы как программисты воспользуемся таким определением:

Аппаратной архитектурой называются те компоненты
--

вычислительной системы, через которые программное обеспечение взаимодействует с аппаратурой.

Таким образом, в аппаратную архитектуру попадают не все компоненты компьютера, а только программно доступные – те, состоянием и действием которых программа может управлять или с которых программа может считать информацию. В состав этих средств входят:

- система команд процессора;
- регистры процессора;
- память;
- система ввода-вывода;
- система прерываний.

Аппаратную поддержку управления памятью и вводом-выводом мы рассматриваем отдельно (в главах 3 и 6 соответственно).

**Система команд процессора** обеспечивает выполнение программой действий по обработке данных. Большинство команд в системе команд процессора имеет прикладное назначение, однако некоторые команды из набора команд процессора предназначены для организации управления вычислительным процессом и, таким образом, непосредственно поддерживают функционирование ОС. Такие команды в современных системах являются привилегированными – это, например, команды ввода-вывода и изменения состояния системы. Современные ОС рассчитаны на наличие в вычислительной системе двух (как минимум) режимов функционирования процессора – привилегированного режима (режим ядра в терминологии Unix) и непривилегированного режима (режим процесса в Unix). Если программа, выполняющаяся в режиме ядра, может выполнять любые команды, то для программы, выполняющейся в режиме процесса, привилегированные команды запрещены. Попытка программы выполнить

привилегированную команду в режиме процесса вызывает исключение (см. ниже). В системе ESA, например, таких основных состояний два (есть еще ряд промежуточных) [20, 45], они называются "супервизор" и "задача", такие же названия они имеют в процессоре Power PC. В процессорах Intel-Pentium аналогичную роль играют уровни привилегий, они же – кольца защиты [32], причем из четырех аппаратно обеспечиваемых уровней привилегий в современных ОС используются два или три. Возможность для пользователя разрабатывать модули, работающие в режиме ядра, обычно строго регламентируется ОС. Хорошо защищенная ОС должна безоговорочно пресекать попытки процесса перейти в состояние ядра.

В число **регистров процессора** входят регистры общего назначения, которые в основном используются для манипулирования с прикладными данными, но также и специальные регистры, такие как регистр адреса команды, регистр флагов-признаков, регистр режима процессора и т.п. Содержимое регистра режима процессора определяет привилегированное или непривилегированное состояние процессора, команды, изменяющие содержимое этого регистра, обязательно являются привилегированными. В различных архитектурах специальные регистры могут либо представлять собой отдельные аппаратные компоненты, либо интегрироваться в более сложные аппаратные структуры.

Содержимое специальных аппаратных регистров процессора (обязательно включая регистр адреса команды) составляет вектор состояния программы/процесса. В большинстве процессорных архитектур вектор состояния может быть загружен в соответствующие регистры или считан из них в память одной или несколькими командами. Так, в процессорах Intel-Pentium имеется структура данных, называемая TSS (Task State Segment – сегмент состояния задачи),



содержимое которой играет роль вектора состояния. При выполнении команд JMP или CALL, адресующих дескриптор TSS, процессор среди прочих действий сохраняет содержимое регистров в TSS текущей задачи и загружает регистры из TSS новой задачи [32]. В процессоре S/390 [20] имеется 8-байтная структура PSW (Program Status Word – слово состояния программы), содержащая значительную часть информации вектора состояния (кроме содержимого регистров общего назначения), и имеются две команды – LPSW и SPSW – для загрузки и запоминания PSW соответственно.

**Прерывание** состоит в прекращении выполнения текущей программы и передаче управления на другую программу – программу обработки прерывания. При этом сохраняется возможность возврата в прерванную программу, в ту точку, в которой ее выполнение было прервано. При всем разнообразии аппаратных архитектур выполнение прерывания в них происходит примерно по одному сценарию:

- сохраняется вектор состояния прерванной программы (в стеке или в специально предназначенной для этого области оперативной памяти);
- в регистры процессора загружается некоторый вектор состояния, заранее "заготовленный";
- в "заготовленном" векторе состояния регистр адреса команды содержит адрес программы обработки прерывания, таким образом, управление передается на программу обработки прерывания;
- как правило, программа обработки прерывания сохраняет содержимое регистров общего назначения, а затем выполняет действия, предусмотренные для данного прерывания;
- после выполнения своих действий программа обработки прерывания восстанавливает содержимое регистров общего назначения прерванной

программы, а затем восстанавливает ее запомненный ранее вектор состояния;

- прерванная программа продолжает свое выполнение с точки прерывания, даже "не заметив", что было принято и обработано прерывание.

Различаются прерывания трех типов: внешние, программные и исключения.

Внешние прерывания поступают от источников, внешних по отношению к процессору. Такими источниками являются внешние устройства, другие процессоры и т.д. При помощи такого прерывания внешний источник сигнализирует о каком-либо изменении своего состояния, требующем реакции системы. Внешние прерывания являются важнейшим компонентом управления вводом-выводом. Внешнее прерывание является асинхронным, то есть оно поступает в непредсказуемые моменты и невозможно предугадать, какой участок программного кода будет прерван внешним прерыванием. Команды процессора обладают свойством атомарности в отношении внешних прерываний: внешнее прерывание не может быть принято, пока не закончится выполнение текущей команды. При сохранении вектора состояния в нем запоминается адрес той команды, которая должна выполняться после команды, во время выполнения которой произошло внешнее прерывание.

Программное прерывание вызывается специальной командой процессора (в Intel-Pentium мнемоника этой команды – INT, в S/390 – SVC). Выполняется программное прерывание так же, как и внешнее, но, в отличие от внешних, программные прерывания являются синхронными, так как они вызываются самой программой. Программные прерывания являются средством обращения процесса к ОС, механизмом системного вызова. Обычные команды передачи управления – типа команд CALL или

JMP – изменяют регистр адреса команды, но не весь вектор состояния. Прерывание же позволяет изменить весь вектор состояния, то есть не только передать управление на другую программу, но и перевести процессор из непривилегированного режима в привилегированный.

Прерывания, называемые исключениями (exception) или ловушками (trap), вызываются ошибочными ситуациями при выполнении команды. В отличие от внешних или программных прерываний, исключения прерывают выполнение команды на середине. Вектор состояния, запоминаемый при выполнении исключения, таков, что его восстановление приводит к повторному выполнению команды, вызвавшей исключение. Исключение, например, генерируется при неправильном коде команды, при попытке выполнить привилегированную команду в не привилегированном режиме, при попытке команды обращения к недоступной области памяти и т.д. Как правило, обработка ОС прерывания-исключения приводит к принудительному завершению процесса, в котором произошло исключение (и запомненный вектор состояния уже не восстанавливается). Однако в некоторых случаях (некоторые из таких случаев рассматриваются нами в последующих главах) исключение является штатной ситуацией, замаскированной формой системного вызова, сигнализирующего ОС о необходимости выполнить для процесса некоторое обслуживание.

## **1.5. Ядро и процессы**

Прежде чем приступить к изложению основного материала этого раздела, сделаем несколько общих замечаний о принципах управления ресурсами и о представлении их в нашем пособии.

Ядром (kernel) называется часть ОС, выполняющая некоторый минимально необходимый набор функций по управлению ресурсами. Дополнительные функции управления ресурсами выполняются вспомогательными модулями – утилитами. Точное определение ядра дать трудно, так как оно по-разному понимается в разных ОС. В "старых", не работавших с виртуальной памятью системах под ядром обычно понималась часть системы, резидентная в оперативной памяти. В современных ОС ядро резидентно в виртуальной памяти, и это также может служить его классификационным признаком. Более узкое определение, трактующую ядро как часть ОС, которая работает в привилегированном режиме, представляется нам более подходящей для определения микроядра (см. раздел 1.6).

На ядро, как правило, возлагаются такие основные функции:

- обработка прерываний;
- создание и уничтожение процессов;
- переключение процессов из одного состояния в другое;
- управление памятью;
- синхронизация и взаимодействие процессов;
- поддержка операций ввода-вывода (не всегда);
- учет работы системы и использования ресурсов пользователями;
- и т.д.

Для ОС процесс представляется блоком контекста процесса (вспомните примечание 3 к определению процесса). Блок контекста содержит информацию о процессе, необходимую для ОС. Обязательной составляющей блока контекста является вектор состояния процессора. Остальная часть блока контекста содержит описание выделенных процессу ресурсов, например:

- идентификатор пользователя-владельца процесса;

- информацию для планирования процесса на выполнение;
- информацию об оперативной и вторичной памяти;
- информацию о других выделенных процессу ресурсах;
- информацию об открытых устройствах и файлах;
- учетную информацию.

Составляющие блока контекста могут храниться в разных местах памяти и даже вытесняться на внешнюю память. Действия ОС по управлению процессами сводятся к манипуляциям с блоками контекста процессов и с отдельными полями этих блоков.

Вспомним теперь примечание 1 к определению процесса: процесс в системе может находиться в различных состояниях. Количество состояний процесса разное в разных ОС (так, в ОС Unix различают 9 возможных состояний процесса), но все они сводятся к трем основным, показанным на рисунке 1.5.



Рисунок 1.5 Состояния процесса

**Активное** состояние – процесс имеет все необходимые для выполнения ресурсы, в том числе и ресурс центрального процессора; активный процесс выполняется.

**Готовое** состояние – процесс имеет все необходимые для выполнения ресурсы, кроме ресурса центрального процессора.

**Заблокированное** (ожидающее) состояние – процессу не хватает еще какого-либо ресурса (ресурсов).

Рассмотрим (пока в общих чертах) переходы между состояниями.

Прежде всего – вход в систему (1 на рисунке 1.5). Мы указали в числе функций ядра порождение процессов, то есть создание для процесса блока контекста. Интерактивный процесс создается, когда пользователь за терминалом вводит команду `login`. Пакетный процесс выбирается ОС из очереди введенных заданий. В последнем случае ОС сама выбирает, какое задание и в какой момент времени выбрать. Кроме того, новый процесс может быть порожден из уже выполняющегося при помощи соответствующего системного вызова. Операции по принятию решений ОС о создании нового процесса называются планированием заданий, или долгосрочным планированием.

Активный процесс может перейти в заблокированное состояние (2 на рисунке 1.5) по двум причинам: по собственной инициативе – процесс выдает системный вызов – запрос на ресурсы, которые не могут быть ему предоставлены немедленно (например, выполнение операции ввода-вывода); по инициативе ОС. Во втором случае ОС "принудительно" отбирает у процесса ресурсы, чтобы отдать их другому (более приоритетному) процессу. По этой же причине ОС может забрать ресурсы и у процесса, находящегося в готовом состоянии (4 на рисунке 1.5). Когда ресурс, которого не хватает процессу, освобождается, ОС назначает его процессу и, если у процесса теперь есть все ресурсы, переводит процесс в готовое состояние (5 на рисунке 1.5). Теперь процесс будет состязаться с другими готовыми процессами за обладание ресурсом центрального процессора. В некоторых случаях (в зависимости от принятой в ОС дисциплины планирования) высокоприоритетный процесс может сразу после получения ресурса переводиться в активное состояние (3 на рисунке 1.5), вытесняя текущий активный процесс. (Как правило, переход 3

непосредственно не реализуется, а выполняется через переходы 5 и 7.) Перемещение процессов между активным/готовым и заблокированным состояниями называется планированием ресурсов, или среднесрочным планированием.

Процесс может перейти из активного состояния в готовое (6 на рисунке 1.5) либо по собственной инициативе, добровольно отказавшись от использования центрального процессора, либо по инициативе ОС. Перевод процессов из готового состояния в активное (7 на рисунке 1.5) выполняет ОС в соответствии с принятой дисциплиной планирования процессов, или краткосрочного планирования.

Для каждого из состояний ОС создает список или списки процессов, находящихся в этом состоянии. В системе с одним центральным процессором список активных процессов содержит только один элемент. Список готовых процессов может содержать несколько элементов. Что же касается списка заблокированных процессов, то в ОС, как правило, имеется несколько таких списков – свой список для каждого класса ожидаемых ресурсов. Смена состояния процесса вызывает перемещение процессов между списками. Рассмотрим с этой точки зрения выполнение системного вызова. Если процесс выдает системный вызов, то обязательно происходит переключение контекста – из контекста процесса в контекст ядра. Если системный вызов может быть выполнен немедленно (например, запрос текущего времени), то он выполняется, и сразу же происходит обратное переключение контекста. Если же выполнение вызова требует времени, то текущий активный процесс переносится в соответствующий список заблокированных, из списка готовых выбирается и назначается активным другой процесс, и контекст переключается на новый активный процесс, то есть в этом случае происходит переключение процессов. Прерывание вызывает переключение контекста на ядро ОС. Обработав прерывание, ОС либо выполняет обратное переключение на контекст

прерванного процесса, либо переводит прерванный процесс в готовое состояние, а активным назначает другой процесс и возвращается из прерывания в его контекст. Как мы показали ранее, переключение контекстов – операция быстрая, а вот переключение процессов – значительно более медленная. Поэтому для минимизации "накладных расходов" по управлению ОС стремится по возможности уменьшить число переключений процессов. Само переключение процессов (но не его планирование) часто называют диспетчеризацией (dispatching).

Механизм переключения контекста с одного процесса на другой обычно реализуется следующим образом. Решение о переключении принимает ОС, следовательно, текущий процесс прерывается (например, по внешнему прерыванию от таймера), его вектор состояния запоминается, как было описано в разделе 1.4, а в регистры процессора загружается вектор состояния модуля-планировщика ОС. В таблице процессов сохраняются векторы состояния всех выполняющихся в системе процессов, соответствующие тем моментам, в которые их выполнение было прервано в последний раз. Если планировщик принимает решение о том, что активным должен стать другой процесс, то он переписывает вектор состояния прерванного процесса в его блок контекста, а в стек (или в ту область памяти, из которой восстанавливается вектор состояния) записывает вектор состояния того процесса, который должен стать активным. При возврате из обработки прерывания, таким образом, восстанавливается вектор состояния нового процесса.

Каждый уровень планирования осуществляется отдельным планировщиком (или планировщиками) в составе ОС – процедурой или процессом. В последующих главах мы рассмотрим работу большей части этих планировщиков. В литературе часто собственно планировщиком (scheduler) называют планировщик процессорного времени, планировщики различных ресурсов называют менеджерами (manager) или мониторами



(monitor) соответствующих ресурсов, а планировщик заданий носит название диспетчера заданий (jobs dispatcher).

В заключение раздела необходимо сделать некоторые обобщения, важные для нашей дальнейшей работы. Мы показали, что процесс, с точки зрения ОС, представляется блоком контекста. Обобщение этого принципа можно сформулировать так: активные объекты с точки зрения ниже лежащего уровня представляются структурами данных. Вот пример такого подхода: машинная команда, с точки зрения программиста, является активной единицей, так как она выполняет некоторые действия, но, с точки зрения процессора, команда – структура данных, содержащая поля кода операции и операндов и подлежащая обработке по алгоритмам процессора. Пример из совершенно другой области: сотрудник любого учреждения, безусловно, считает себя активной личностью, но, с точки зрения отдела кадров, он – всего лишь стандартная карточка учета, и все его перемещения по службе осуществляются простым изменением в графах этой карточки.

Поскольку наше рассмотрение будет сосредоточено в основном на нижних уровнях, становится очевидным, что первостепенную важность для нас имеют структуры данных, описывающие объекты, которые обрабатываются в ОС и алгоритмы их обработки. Блок контекста процесса – первая из таких структур данных. При представлении системных структур данных мы в большинстве случаев решили отказаться от попыток представить их сколько-нибудь формализованно, например, средствами какого-либо языка программирования. Такой отказ объясняется тем, что мы стремились избежать даже намека на то, что та или иная структура является универсальной, обязательной, фиксированной для всех ОС – это ни в коем случае не так. Состав компонентов таких структур, их именование, взаимное расположение, типы данных и т.д. чрезвычайно разнятся для разных ОС в соответствии с их назначением и даже личными

вкусами разработчиков. Поэтому мы ограничиваемся самым свободным описанием структур – простым перечислением той информации, которая в большинстве случаев должна в них содержаться. Описания структур средствами языка программирования или в виде таблиц мы будем применять только там, где речь будет идти о конкретных ОС.

При решении некоторых задач управления ресурсами ОС должна создавать интегрированные структуры данных, содержащие набор объектов одинакового типа. Списки процессов – пример таких интегрированных структур. Как известно, имеется два общих метода представления таких структур в памяти: смежное и связанное.

Конструктор ОС также свободен в выборе того или иного метода для представления той или иной интегрированной структуры. Мы в дальнейшем будем употреблять термин "таблица" – для обозначения интегрированной структуры, которая скорее всего (но не обязательно) реализуется смежным представлением, и "список" – для структуры, которая реализуется скорее всего связным представлением.

## **1.6. Архитектурные концепции операционных систем**

ОС является сложным программным изделием, поэтому при ее проектировании невозможно пренебрегать вопросами структурирования, что подчас допустимо при разработке небольших программ. Все архитектуры ОС в том или ином варианте используют модульно-интерфейсные методы структурирования [10, 14]. ОС представляется состоящей из ряда модулей (планирование процессов, управление памятью, подсистема ввода-вывода и т.д.), для каждого из которых определены спецификации функционирования и интерфейсы. ОС, однако, различаются по способам оформления модулей и связей между ними.

Модульный состав и организация межмодульного взаимодействия и составляют архитектуру ОС.

Первой из четко сформулированных архитектурных концепций ОС была иерархия абстрактных машин, предложенная Дейкстрой в 1968 году для ОС TNE (описание можно найти в [29]). Иерархия абстрактных машин в этой системе показана на рисунке 1.6. Самый нижний (нулевой) уровень иерархии составляет реальная машина с ее интерфейсом оборудования. Нижний слой программного обеспечения составляет первый уровень. Совместно с аппаратными средствами он представляет некоторую абстрактную машину со своим, более высокоуровневым интерфейсом оборудования. На основе этого интерфейса строится абстрактная машина второго уровня и т.д. Последовательным наращиванием слоев программного обеспечения интерфейс абстрактной машины доводится до уровня интерфейса процессов.

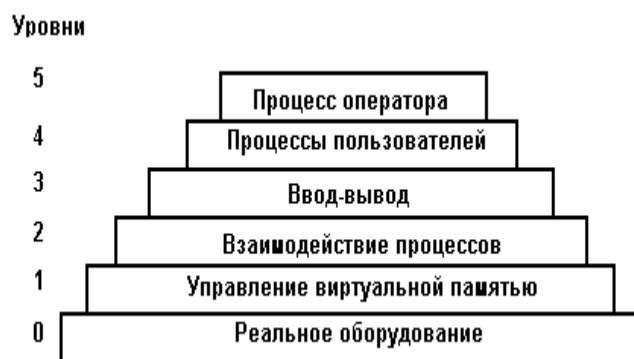


Рисунок 1.6 Иерархия абстрактных машин в системе TNE

Реализация архитектуры абстрактных машин сопряжена со значительными трудностями, связанными с правильным выбором уровней и их иерархическим упорядочением. Система TNE представляет только один из возможных вариантов, применимый далеко не во всех случаях. Успешность решения этой проблемы во многом зависит от выбранного метода проектирования. В первоисточнике реализация иерархии

абстрактных машин производилась методом "снизу вверх". Другие авторы (например, [15]) настаивают на реализации методом "сверху вниз". По-видимому, наиболее продуктивным является комбинированный метод, пример применения которого приведен в [14]: спецификации уровней разрабатываются "сверху вниз", а реализация ведется "снизу вверх". При любом методе проектирования обеспечиваются некоторые общие свойства уровней абстракции, важнейшие из которых следующие:

- каждый уровень обеспечивает некоторую абстракцию данных в системе и, располагая определенными ресурсами, либо скрывает их от других уровней, либо предоставляет другим уровням виртуальные ресурсы;
- на каждом уровне ничего не известно о свойствах более высоких уровней;
- на каждом уровне ничего не известно о внутреннем строении других уровней;
- связь между уровнями осуществляется только через жесткие, заранее определенные сопряжения.

Иногда иерархию абстрактных машин иллюстрируют набором концентрических окружностей (например, [41]), чтобы подчеркнуть, что каждый следующий уровень иерархии полностью скрывает все лежащие ниже него уровни и каждый уровень может обращаться только к непосредственно нижележащему уровню. Обращения, адресованные к более низким уровням, последовательно проходят все промежуточные уровни.

Популярными современными вариациями на тему иерархической архитектуры являются концепции виртуальной машины и микроядра. В обоих случаях некоторый уровень иерархии получает особый статус и служит границей между двумя основными уровнями системного программного обеспечения. Спецификации интерфейса между двумя

основными уровнями четко определены, что делает их независимыми друг от друга.

В концепции виртуальной машины интерфейс процесса выглядит как интерфейс оборудования. В предельном случае, который можно наблюдать, например, в VM/ESA [28, 46] внешние формы этих двух интерфейсов полностью совпадают. В этом случае процессу доступны все машинные команды, в том числе и привилегированные. Но эта доступность кажущаяся. На самом деле, выдача процессом привилегированной команды вызывает исключение. В большинстве ОС обработка такого исключения включает в себя аварийное завершение процесса, но в z/VM управление по исключению получает нижний уровень системы – СР (управляющая программа). СР определяет причину исключения и выполняет для процесса требуемую команду или моделирует выполнение этой команды на виртуальном оборудовании. У процесса создается иллюзия, что в его полном распоряжении находится реальная вычислительная система (с той, однако, поправкой, что временные соотношения выполнения некоторых команд не выдерживаются). А если так, то процесс, в свою очередь может быть ОС, так называемой гостевой (guest) ОС, которая в полном объеме управляет выделенным ей подмножеством ресурсов.

Другой вариант концепции виртуальной машины представляет интерфейс нижнего уровня системного программного обеспечения как полнофункциональный набор команд некоторой воображаемой машины. Все вышележащие уровни программного обеспечения пишутся в этом наборе команд (или компилируются в него). Программный модуль, готовый для выполнения, представляет собой двоичный код программы в командах виртуальной машины. Перевод команд виртуальной машины в команды конкретной аппаратной платформы выполняется нижележащим уровнем программного обеспечения в режиме перекомпиляции (например,

AS/400 [11, 39]) или интерпретации (например, технология Java [5, 37]), компилятор или интерпретатор входит в состав нижнего уровня. Использование промежуточного кода в командах виртуальной машины обеспечивает переносимость программного обеспечения на другие платформы, так как все программное обеспечение (в том числе и системное), лежащее выше уровня интерфейса виртуальной машины, является платформенно-независимым и при переносе не требует даже перекомпиляции.

Другая вариация на тему иерархической архитектуры – концепция микроядра. Суть ее заключается в том, что части системного программного обеспечения, которые выполняются в режиме ядра, сосредоточены на нижнем уровне иерархии, они и составляют микроядро. Объем микроядра минимизируется, что повышает надежность системы. Прочие модули ОС выполняются в режиме процесса и, с точки зрения микроядра, ничем не отличаются от процессов пользователя. В микроядро включаются также наиболее важные платформенно-зависимые функции с тем, чтобы обеспечить оптимизацию их выполнения и относительную независимость от платформы модулей ОС, не входящих в микроядро. Минимальный набор функций микроядра включает:

- управление реальной памятью (это всегда платформенно-зависимая функция);
- переключение контекстов (но не процессов!), решение о том, какой процесс в какое состояние должен перейти, принимает планировщик, который не должен работать в режиме ядра, а в мультипроцессорных системах – и управление загрузкой процессоров;

- предварительная обработка аппаратных прерываний (для полной обработки прерывания перенаправляются тем процессам, которым они адресованы);
- обеспечение коммуникаций между всеми процессорами вне микроядра – системными и пользовательскими, в системах, изначально ориентированных на распределенную обработку – также и сетевых коммуникаций.

Архитектурная концепция микроядра также обеспечивает переносимость системного программного обеспечения верхнего уровня (хотя и с необходимостью его перекомпиляции).

Набор преимуществ, обеспечиваемых микроядром, очень велик и в разных системах это понятие трактуется по-разному – в зависимости от того, какие требования к системе являются доминирующими. Так, описанный выше подход минимизации кода, выполняемого в режиме ядра, и повышения эффективности в полной мере реализован, например, в ОС QNX [42]. В Windows NT/2000 [25, 43] микроядром называют часть, обеспечивающую независимость от внешнего оборудования и ряд функций режима ядра, но одним микроядром эти функции не исчерпываются. В AS/400 [27] часть кода, лежащую ниже интерфейса виртуальной машины, тоже иногда называют микроядром, хотя для программного обеспечения, состоящего из более, чем 1 млн. строк кода на языке C++, префикс "микро" вряд ли уместен.

Еще одной тенденцией в развитии ОС является объектно-ориентированный подход к их проектированию. Как известно, основными свойствами объектно-ориентированного программирования являются инкапсуляция, полиморфизм и наследование. Из указанных свойств в объектно-ориентированных ОС в полной мере реализуется, прежде всего, первое. Ресурсы в таких системах представляются в виде экземпляров тех или иных классов, внутренняя структура класса недоступна вне класса, но

для класса определены методы работы с ним. Наряду с повышением степени интеграции тех базовых элементов, из которых строится ОС, инкапсуляция обеспечивает также защиту ресурсов и возможность менять в новых версиях ОС или при переносе на новую платформу структуру системных объектов без изменения тех программ, которые оперируют объектами. Для каждого типа объектов определен набор операций, допустимых над ним. Свойство полиморфизма состоит в том, что для различных системных классов могут быть определены одноименные операции, выполнение которых для разных классов будет включать в себя как общие, так и специфические для каждого класса действия. Важнейшей из таких операций является получение доступа к объекту, отдельно рассматриваемое нами в главе 10. Свойство наследования реализуется в объектно-ориентированных ОС лишь отчасти, в связи с чем некоторые авторы (например, [27]) считают, что правильнее называть эти ОС объектно-базированными. В системах с иерархической структурой (Windows NT, AS/400) объекты более высокого уровня могут включать в себя объекты нижних уровней, однако производные классы не наследуют методы базовых и, следовательно, их экземпляры не могут обрабатываться как экземпляры базового класса. Нельзя, однако, говорить об этом ограничении как о недостатке, так как оно диктуется концепцией иерархической архитектуры: каждый уровень должен оперировать только объектами своего уровня.

Архитектурные концепции построения ОС не являются взаимоисключающими. Как вы, по-видимому, заметили из приводимых примеров, существуют системы, в архитектурах которых комбинируются несколько подходов.

Важным архитектурным вопросом является оформление модулей ОС. Модули могут представлять собой процедуры или процессы. В первом случае все ядро ОС представляется как один многомодульный процесс и



передача управления между модулями ОС выполняется просто командами типа CALL. Во втором случае каждый модуль представляется в виде отдельного процесса (процесса ядра), и передача управления сопровождается переключением процессов. Хотя во втором случае передача управления занимает больше времени, такой подход обеспечивает, во-первых, лучшую защиту ресурсов, используемых и управляемых ОС, а во-вторых, делает модульную структуру ОС более гибкой. Архитектура процессов ядра может совмещаться с архитектурой иерархии абстрактных машин: каждый уровень иерархии обеспечивается своим набором процессов ядра. Промежуточным случаем является подход, характерный, например, для ОС Unix: обращение процесса к ОС вызывает переключение контекста на ядро, но не переключение процессов, то есть модули ядра выполняются в контексте вызвавшего их процесса. В тех ОС, в которых отношения между процессами строятся по схеме "предок–потомок", иерархия может непосредственно отображаться в "родственных отношениях" процессов. Что касается процедурной архитектуры, то такие отношения в ней естественным образом отображаются на вложенности вызовов процедур.

Оформление модулей ОС непосредственно связано с проблемой управления ресурсами, особенно ресурсами разделяемыми. В случае процедурного оформления управление каждым видом (классом) ресурса выполняется отдельной процедурой-монитором. Монитор является процедурой, используемой всеми функционирующими в системе процессами, и процедура эта всегда выполняется в контексте вызвавшего ее процесса. Структура монитора предотвращает конфликты при одновременном обращении к нему двух или более процессов. При оформлении модулей ОС в виде процессов ядра каждый ресурс обслуживается своим процессом-менеджером. Доступ к ресурсу из любого

другого процесса выполняется через обращение к менеджеру и переключение в контекст менеджера.

Еще один важный вопрос организация взаимодействия между модулями, и здесь можно выделить две модели [19]: интерфейс процедур и интерфейс сообщений. Интерфейс процедур подразумевает непосредственное обращение вызывающего модуля к вызываемому, подобное обращению к подпрограмме в языках программирования. Обращение может быть либо, действительно, обращением к процедуре (команда CALL), либо сводиться к прерыванию и переключению процессов. Модель интерфейса процедур синхронная, то есть как и при вызове подпрограммы, выполнение вызывающего модуля приостанавливается до получения результата вызова. Эта модель может быть построена как на базе процедурных модулей ОС, так и на базе модулей-процессов. Другая модель обеспечивает взаимодействие процессов через единый системный механизм очередей. Процесс-клиент (в этой модели модули ОС должны быть именно процессами) оформляет свой запрос в виде сообщения и отправляет его процессу-серверу. Процесс-сервер получает сообщение из своей входной очереди, выполняет содержащийся в сообщении запрос и отправляет результат в виде сообщения процессу-клиенту. Процесс-клиент после отправки своего сообщения может либо продолжать выполняться, либо ожидать прихода ответного сообщения. Взаимодействие процессов, таким образом, происходит асинхронно.

Подходы, которые выбирают в современных ОС, в значительной степени определяются их назначением. Однопользовательские ОС стремятся в максимальной степени повысить быстродействие выполнения приложений. Хотя в сверхвысоком быстродействии зачастую нет функциональной необходимости, оно является существенным фактором в конкурентной борьбе. Поэтому в таких ОС проявляется тенденция к

предельной минимизации числа переключений процессов и к выполнению системных вызовов в контексте процесса пользователя. Отсюда – процедурная форма модулей ядра, мониторинг управления ресурсами, процедурная модель взаимодействия. С другой стороны, многопользовательские ОС, как правило, представляют модули ядра в виде процессов-менеджеров ресурсов и для взаимодействия между ними предпочтение отдается модели сообщений. Это обеспечивает значительно лучшую защиту ресурсов, особенно в среде с высоким уровнем мультипрограммирования.

## **КОНТРОЛЬНЫЕ ВОПРОСЫ**

1. Один из авторов заявляет, что не может дать определения ОС, но сразу узнает ОС, если ее увидит. В чем, по-вашему, состоит ошибочность такого утверждения?
2. Прокомментируйте примечания 1-3 к определению ОС, данному в разделе 1.1. Покажите их отображения на реальные ОС.
3. Дайте определение пакетному и интерактивному режимам функционирования ОС. Какой из режимов представляется вам более полезным?
4. В чем сходство работы многопользовательской интерактивной ОС с ОС-сервером? В чем их различия?
5. Каковы достоинства и недостатки изоляции пользователя от реальных ресурсов?
6. Назовите основные состояния процесса в системе и охарактеризуйте переходы между ними. Какие состояния вы считаете необязательными?
7. Почему ОС, называемые объектно-ориентированными, правильнее называть объектно-базируемыми?

8. Назовите общие черты архитектурных концепций микроядра, виртуальной машины и иерархической ОС. В чем различия между ними?
9. В чем достоинства архитектуры микроядра? Почему разработчики стремятся минимизировать объем микроядра?
10. Сравните способы обращения процесса к ОС: через вызов процедур и через прерывания. В чем достоинства и недостатки этих способов?

## **ГЛАВА 2. ПЛАНИРОВАНИЕ ПРОЦЕССОВ**

В предыдущей главе мы определили три уровня планирования, теперь остановимся более подробно на стратегиях или дисциплинах планирования. Многие дисциплины применимы на любых уровнях планирования, но мы сосредоточимся, прежде всего, на планировании краткосрочном или планировании процессорного времени. Процессорное время является ключевым ресурсом любой вычислительной системы и наличие или отсутствие этого ресурса в распоряжении процесса отличает активное состояние процесса от остальных его состояний. Дисциплины распределения этого ресурса в значительной степени определяют эффективность функционирования всей системы в целом.

### **2.1. Дисциплины планирования – требования, показатели, классификация**

В общем случае планирование (на любом уровне) может быть представлено как система массового обслуживания, показанная на рисунке 2.1. Применительно к планированию процессорного времени компоненты этой системы могут быть интерпретированы следующим образом: заявкой

является процесс, обслуживающим прибором – центральный процессор (ЦП), очередь заявок – это очередь готовых процессов. Процессы-заявки поступают в очередь, при освобождении ЦП один процесс выбирается из очереди и обслуживается в ЦП. Обслуживание может быть прервано по следующим причинам:

- выполнение процесса завершилось;
- процесс запросил выполнение операции, требующей ожидания какого-либо другого ресурса;
- выполнение прервано системой.

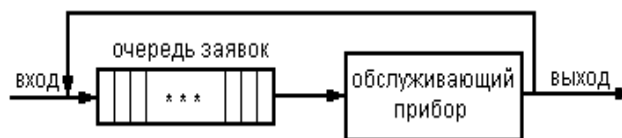


Рисунок2.1 Представление планирования процессов в виде системы массового обслуживания

Первые два случая с точки зрения системы массового обслуживания одинаковы: в любом случае процесс выходит из данной системы. Если процесс не завершился, то после получения запрошенного ресурса процесс вновь поступит во входную очередь. В случае прерывания процесса по инициативе системы прерванный (вытесненный) процесс поступает во входную очередь сразу же. Порядок обслуживания входной очереди, очередность выбора из нее заявок на обслуживания и составляет дисциплину или стратегию планирования. Методы теории массового обслуживания применяются для аналитического моделирования процесса планирования, хотя формальному анализу поддаются только простейшие дисциплины (см., например, [16]).

Для оценки эффективности функционирования данной системы массового обслуживания могут быть применены количественные

показатели. Обозначим через  $t$  – процессорное время, необходимое процессу для выполнения (будем его называть длительностью процесса). Обозначим через  $T$  – общее время пребывания процесса в системе. Эту величину – интервал между моментом ввода процесса в систему и моментом получения результатов – также называют иногда временем реакции процесса. Наряду с временем реакции могут быть полезны также и другие показатели.

Потерянное время

$$M = T - t;$$

определяет время, в течение которого процесс находился в системе, но не выполнялся.

Отношение реактивности

$$R = t / T;$$

показывает долю процессорного времени (времени выполнения) или долю потерянного времени в общем времени реакции.

Штрафное отношение

$$P = T / t;$$

показывает, во сколько раз общее время выполнения процесса превышает необходимое процессорное время.

Средние значения величин  $T$ ,  $M$ ,  $R$  и  $P$  могут служить количественными показателями эффективности. Реальные системы, как правило, ориентированы на конкретные характеристики процессов, в частности, на определенные диапазоны значений  $t$ , поэтому указанные показатели удобно рассматривать как функции длительности процесса:  $T(t)$ ,  $M(t)$ ,  $R(t)$ ,  $P(t)$ .

К дисциплине планирования в общем случае может применяться широкий спектр требований, наиболее существенные из которых следующие:

- дисциплина должна быть справедливой – она не должна давать преимуществ одним процессам за счет других и ни в коем случае не должна допускать бесконечного откладывания процессов;
- дисциплина должна обеспечивать максимальную пропускную способность системы – выполнение максимального количества единиц работы (процессов) в единицу времени;
- дисциплина должна обеспечивать приемлемое время реакции для интерактивных пользователей;
- дисциплина должна обеспечивать гарантированное время реакции для процессов реального времени;
- дисциплина должна быть предсказуемой – дисперсия времен выполнения процессов, обладающих одинаковыми характеристиками, должна быть минимальной;
- дисциплина должна учитывать внешние приоритеты, присваиваемые процессам пользователями и/или администратором системы;
- накладные расходы по реализации дисциплины (затраты процессорного времени и других ресурсов) должны быть минимизированы;
- дисциплина должна учитывать комплексное использование ресурсов вычислительной системы, обеспечивая высокую загрузку системы в целом и рациональное использование ключевых ресурсов.

Очевидно, что выполнение всех перечисленных требований в одинаковой степени невозможно, так как некоторые из них противоречат друг другу. В конкретных системах те или иные требования выдвигаются на передний план – в зависимости от задач системы и характеристик выполняемых в ней процессов, возможно и выдвижение на первый план новых требований, не упомянутых в нашем списке.

В большинстве случаев рассмотрение оценок эффективности планирования процессорного времени производится при условии одного

существенного допущения: не принимаются во внимание другие уровни планирования. Выполнение реального процесса состоит из выполнения программы в ЦП и операций ввода-вывода. Последние, во-первых, занимают значительно больше времени (но не времени процессорного), во-вторых, могут включать в себя ожидание ресурсов ввода-вывода. Реальные процессы могут быть классифицированы как вычислительные или обменные. Первые состоят в основном из вычислений в ЦП, вторые – содержат большое количество обращений к вводу-выводу. Оценки эффективности планирования процессорного времени выполняются в предположении, что все процессы относятся к вычислительному типу. Обменные процессы могут быть приведены к этой модели путем представления каждой последовательности процессорных команд между двумя операциями ввода-вывода как отдельного процесса вычислительного типа. Для систем, требующих комплексного сбалансированного управления ресурсами, стратегия затем расширяется учетом факторов, определяемых другими ресурсами.

С точки зрения реализации дисциплины планирования подразделяются, прежде всего, на дисциплины вытесняющие (preemptive) и невытесняющие (non-preemptive), иначе – кооперативные (cooperative). Для первых возможно прерывание активного процесса и лишение его ресурса ЦП по инициативе планировщика, для вторых – нет. Дисциплины с вытеснением выполняют более частые переключения процессов, следовательно, имеют большие накладные расходы. Но в большинстве случаев только дисциплины с вытеснением могут обеспечить требуемые показатели справедливости обслуживания.

Другие размерности классификации дисциплин связаны со способами определения и реализации приоритетов процессов. Различают приоритеты:



- внешние или внутренние – первые назначаются администратором системы или пользователем, вторые определяются самой системой по характеристикам процесса;
- статические или динамические – первые определяются при поступлении процесса в систему и не изменяются впоследствии, вторые перевычисляются планировщиком периодически или/и при событиях, влияющих на планирование процессов;
- абсолютные или относительные – в абсолютных к выполнению допускается только процессы, имеющие наивысший приоритет, в относительных допускается планирование на выполнение и низкоприоритетных процессов.

Еще одной важной с точки зрения реализации характеристикой дисциплины планирования является объем априорной информации о процессе, необходимой планировщику. Если дисциплина не учитывает использование других ресурсов, кроме ЦП, то такой информацией может быть длительность процесса, так как показатели эффективности являются функциями именно этого аргумента. Если дисциплина использует комплексные приоритеты, то может появиться необходимость и в другой априорной информации. При наличии априорной информации появляется возможность более эффективной реализации, но обязанность подготовки такой информации возлагается на пользователя-владельца процесса, что снижает удобства применения системы. Для процессов, не являющихся чисто счетными, информация, логически эквивалентная априорной, может быть получена методами экстраполяции: на основании предшествовавшего поведения процесса делается предположение о его последующем поведении, например, так, как описано ниже.

Пусть процесс использовал  $S$  единиц времени ЦП до перехода в ожидание ввода-вывода. Тогда прогноз на следующий интервал времени ЦП, который понадобится процессу, может быть сделан так:

$$E' = W1 * E + W2 * S,$$

где  $E$  – прогноз, сделанный на предыдущем интервале для текущего интервала,  $W1$  и  $W2$  – весовые коэффициенты, подбираемые так, что

$$W1 + W2 = 1.$$

При изменении соотношения весовых коэффициентов в сторону увеличения  $W2$  прогноз становится более реактивным (более чувствительным к изменению поведения процесса), в обратную сторону – более инерционным.

## **2.2. Базовые дисциплины планирования**

Ниже приводятся описания некоторых базовых дисциплин планирования. Эти дисциплины достаточно просты в реализации и хорошо исследованы методами как аналитического (например, [16]), так и имитационного (например, [36]) моделирования. Мы называем их базовыми, поскольку в реальных системах они служат основой для построения более сложных и гибких модификаций и комбинаций, для которых аналитические модели построить, как правило, невозможно.

FCFS (first come – first serve; первым пришел – первым обслуживается) – простейшая дисциплина, работа которой понятна из ее названия. Это дисциплина без вытеснения, то есть процесс, выбранный для выполнения на ЦП, не прерывается, пока не завершится (или не перейдет в состояние ожидания по собственной инициативе). Как дисциплина без вытеснения FCFS обеспечивает минимум накладных расходов. Среднее потерянное время при применении этой дисциплины не зависит от длительности процесса, но штрафное отношение при равном потерянном времени будет большим для коротких процессов. Поэтому дисциплина FCFS считается лучшей для длинных процессов. Существенным достоинством этой дисциплины наряду с ее простотой является то обстоятельство, что FCFS гарантирует отсутствие бесконечного

откладывания процессов: любой поступивший в систему процесс будет в конце концов выполнен независимо от степени загрузки системы.

На рисунке 2.2 показан пример планирования по дисциплине FCFS для трех процессов – А, В и С. На временной диаграмме каждый прямоугольник представляет интервал времени, в течение которого процесс находится в системе. Над верхним левым углом такого прямоугольника указан идентификатор процесса, а в скобках – его длительность. Незатемненные участки соответствуют активному состоянию процесса, затемненные – состоянию ожидания. Процесс А поступает в момент времени 0 и требует для выполнения 6 единиц процессорного времени. ЦП в этот момент свободен, и процесс А сразу же активизируется. В момент времени 2 поступает процесс В, требующий 11 единиц. Поскольку ЦП занят процессом А, процесс В ожидает в очереди готовых процессов до момента 6, когда процесс А закончится и освободит ЦП. Только после этого процесс В начинает выполняться. Пока процесс В выполняется, поступают еще два процесса: С – в момент времени 8 и D – в момент 10, которые ждут завершения процесса В. Когда процесс В завершится, ЦП будет отдан процессу С, поступившему раньше, а процесс D остается в ожидании. В линейке, расположенной под временной шкалой, указаны идентификаторы процессов, активных в данный момент времени. Читатель может сам определить показатели эффективности планирования – для каждого процесса и усредненные. Следует, однако, предупредить, что к усредненным показателям надо относиться с осторожностью, так как достоверными могут считаться только результаты, полученные на статистически значимой выборке.

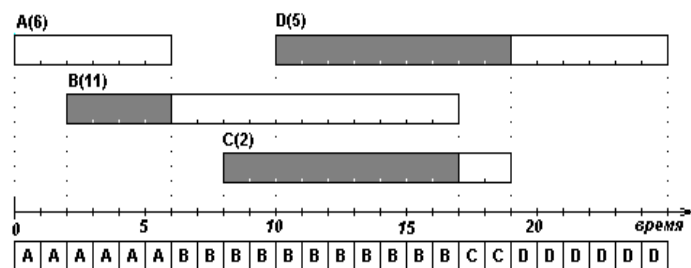


Рисунок 2.2 Планирование процессов по дисциплине FCFS

RR (round robin – карусель) – простейшая дисциплина с вытеснением. Процесс получает в свое распоряжение ЦП на некоторый квант времени  $Q$  (в простейшем случае размер кванта фиксирован). Если за время  $Q$  процесс не завершился, он вытесняется из ЦП и направляется в конец очереди готовых процессов, где ждет выделения ему следующего кванта, и т.д. Показатели эффективности RR существенно зависят от выбора величины кванта  $Q$ . RR обеспечивает наилучшие показатели, если длительность большинства процессов приближается к размеру кванта, но не превосходит его. Тогда большинство процессов укладываются в один квант и не становятся в очередь повторно. При величине кванта, стремящейся к бесконечности, RR вырождается в FCFS. При  $Q$ , стремящемся к 0, накладные расходы на переключение процессов возрастают настолько, что поглощают весь ресурс ЦП. RR обеспечивает наилучшие показатели справедливости: штрафное отношение  $P$  на большом участке длительностей процессов  $t$  остается практически постоянным. Только на участке  $t < Q$  штрафное отношение начинает изменяться и при уменьшении  $t$  от  $Q$  до 0 возрастает экспоненциально. Потерянное же время  $M$  существенно растет с увеличением длительности процесса.

На рисунке 2.3 показаны примеры планирования по дисциплине RR с разными величинами кванта  $Q=1$  (рисунок 2.3.а) и  $Q=4$  (рисунок 2.3.б). Рассмотрим подробнее работу на начальном временном участке рисунка

2.3.а. Процесс А поступает в момент времени 0 и получает квант времени ЦП. К моменту окончания кванта в очереди уже есть процесс В. Процесс А отправляется в очередь, а следующий квант получает процесс В. В момент времени 2 процесс В направляется в очередь, а из очереди выбирается процесс А. В этот же момент поступает новый процесс С. Этот процесс ставится в конец очереди, а первым в очереди стоит процесс А, поэтому следующий квант отдается процессу А и т.д. Предоставляем читателю самостоятельно закончить рассмотрение этого примера, а также примера, показанного на рисунке 2.3.б.

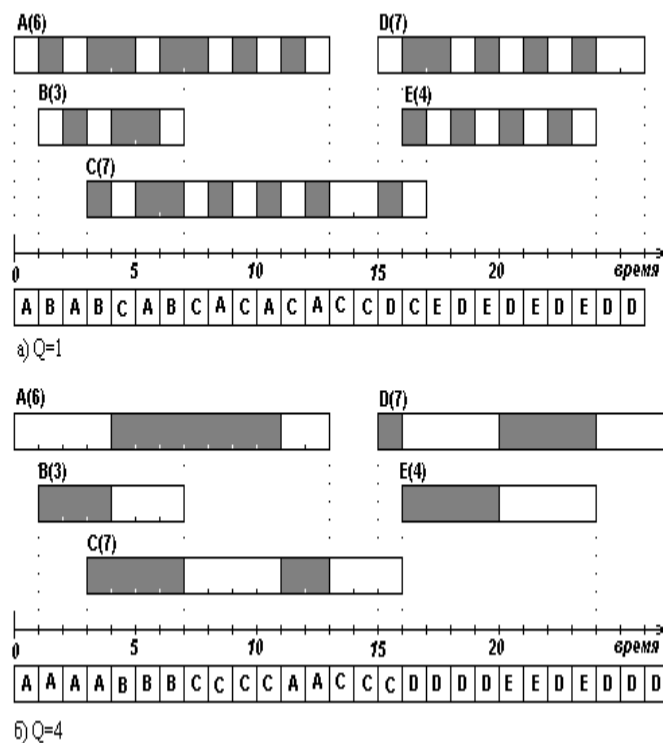


Рисунок 2.3 Планирование процессов по дисциплине RR

SJN (shortest job next – самая короткая работа – следующая) – невытесняющая дисциплина, в которой наивысший приоритет имеет самый короткий процесс. Для того, чтобы применять эту дисциплину, должна быть известна длительность процесса – задаваться пользователем или вычисляться методом экстраполяции. Для коротких процессов SJN обеспечивает лучшие показатели, чем RR, как по потерянному времени,

так и по штрафному отношению. SJN обеспечивает максимальную пропускную способность системы – выполнение максимального числа процессов в единицу времени, но показатели для длинных процессов значительно худшие, а при высокой степени загрузки системы активизация длинных процессов может откладываться до бесконечности. Штрафное отношение слабо изменяется на основном интервале значений  $t$ , но значительно возрастает для самых коротких процессов: такой процесс при поступлении в систему имеет самый высокий приоритет, но вынужден ждать, пока закончится текущий активный процесс.

Пример планирования по этой дисциплине показан на рисунке 2.4. Поступивший в момент времени 0 процесс А захватывает ЦП. Процесс В, поступивший в момент 1, вынужден ждать освобождения ЦП процессом А, хотя процесс В и более короткий. К моменту 6 (освобождения ЦП) из двух имеющихся в очереди процессов (В и С) выбирается более короткий процесс В. Процесс С получает ЦП только в момент времени 9, когда заканчивается процесс В. Когда в момент времени 16 процесс С освобождает ЦП, из двух имеющихся в очереди процессов выбирается более короткий процесс Е, хотя он поступил позже, чем процесс D.

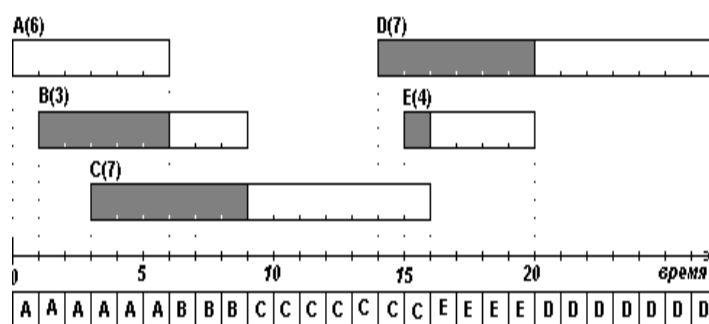


Рисунок 2.4 Планирование процессов по дисциплине SPN

PSJN (preemptive SJN – SJN с вытеснением) – текущий активный процесс прерывается, если его оставшееся время выполнения больше, чем у новоприбывшего процесса. Дисциплина обеспечивает еще большее

предпочтение коротким процессам перед длинными. В частности, в ней устраняется то возрастание штрафного отношения для самых коротких процессов, которое имеет место в SJN.

Рассмотрим пример, представленный на рисунке 2.5. Процесс А поступает в систему первым и успевает использовать единицу времени ЦП прежде, чем в систему приходит процесс В. Процесс В требует 3 единицы процессорного времени, а процессу А осталось использовать еще 5 единиц. Процесс А вытесняется, ЦП отдается процессу В. При освобождении ЦП в очереди уже есть и процесс С, но его длительность больше, чем остаток времени процесса А, поэтому процесс С получает ЦП только в момент времени 9, когда процесс А завершится. Процесс С успевает использовать только одну единицу времени ЦП, когда приходит короткий процесс Е и вытесняет процесс С из ЦП. Выполнение С вновь откладывается до освобождения ЦП, которое происходит в момент 14. В момент 17 приходит процесс D. Его длительность (6) меньше, чем полная длительность процесса С (7), но к этому времени процесс С уже использовал 4 единицы времени ЦП и для завершения ему необходимо еще только 4 единицы, поэтому процесс D не вытесняет процесс С.

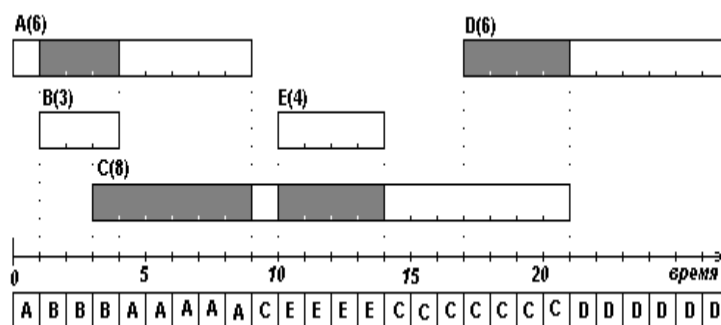


Рисунок 2.5. Планирование процессов по дисциплине PSPN

HPRN (highest penalty ratio next – с наибольшим штрафным отношением – следующий) – дисциплина без вытеснения, обеспечивающая наилучшие показатели справедливости. Это достигается за счет

динамического переопределения приоритетов. Всякий раз при освобождении ЦП для всех готовых процессов вычисляется текущее штрафное отношение

$$p[i] = (w[i] + t[i]) / t[i]$$

где  $i$  – номер процесса;  $w[i]$  – время, затраченное процессом на ожидание;  $t[i]$  – длительность процесса, предзаданная или прогнозируемая. Для только что поступившего процесса  $p[i]=1$ . ЦП отдается процессу, имеющему наибольшее значение  $p[i]$ . Для коротких процессов HPRN обеспечивает примерно те же показатели справедливости, что и SJN, для длинных – более близкие к FCFS. На большом диапазоне средних длительностей процессов показатели, обеспечиваемые HPRN, представляют среднее между SJN и FCFS и слабо зависят от длительности. Еще одно достоинство HPRN в том, что во времени ожидания может учитываться (с некоторыми весовыми коэффициентами) и ожидание в других очередях и, таким образом, выполняется более комплексный учет загрузки системы. Существенным недостатком метода является необходимость перевычисления штрафного отношения для всех процессов при каждом переключении, что плохо согласуется с общей политикой минимизации накладных расходов в дисциплинах без вытеснения.

В примере, показанном на рисунке 2.6, под временной шкалой даны текущие значения штрафного отношения для процессов-претендентов в те моменты времени, когда выполняется переключение. Так, в момент времени 6 два процесса – В и С – претендуют на использование ЦП. Текущее штрафное отношение для процесса В составляет:

$$p[V] = (5 + 3) / 3 = 2.33,$$

а для процесса С:

$$p[C] = (3 + 7) / 7 = 1.43;$$



следовательно, ЦП отдается процессу В. Аналогичные вычисления производятся в моменты времени 9 и 16.

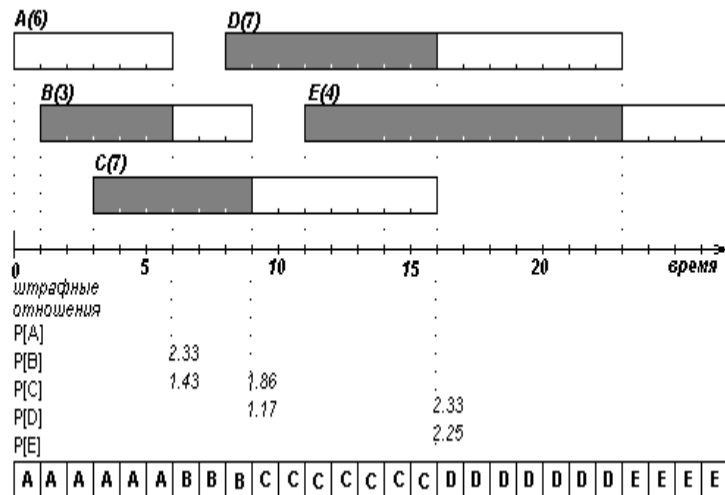


Рисунок 2.6 Планирование процессов по дисциплине HPRN

SRR (selfish RR – эгоистичный RR) – метод с вытеснением, дающий дополнительные преимущества выполняемым процессам, что позволяет повысить пропускную способность. Все процессы разделяются на две категории: новые и выбранные. Новыми считаются те процессы, которые не получили еще ни одного кванта времени ЦП, все остальные процессы – выбранные. При поступлении в систему каждому процессу дается некоторый приоритет  $P_0$ , одинаковый для всех процессов, который в дальнейшем возрастает. В конце каждого кванта времени пересчитываются приоритеты всех процессов, причем приоритеты новых процессов возрастают на величину  $dA$ , а выбранных – на величину  $dB$ . ЦП отдается процессу с наивысшим приоритетом, а при равенстве приоритетов – тому, который раньше поставлен в очередь. Показатели дисциплины существенно зависят от выбранного соотношения между  $dA$  и  $dB$ . При  $dB/dA=0$  дисциплина вырождается в обыкновенную RR, при  $dB/dA \geq 1$  – в FCFS. Собственно дисциплина SRR обеспечивается в диапазоне значений  $0 < dB/dA < 1$ .

Рассмотрим работу дисциплины на примере, показанном на рисунке 2.7. Параметры дисциплины в этом примере:

$$P_0=0; \quad dA=2; \quad dB=1; \quad Q=1.$$

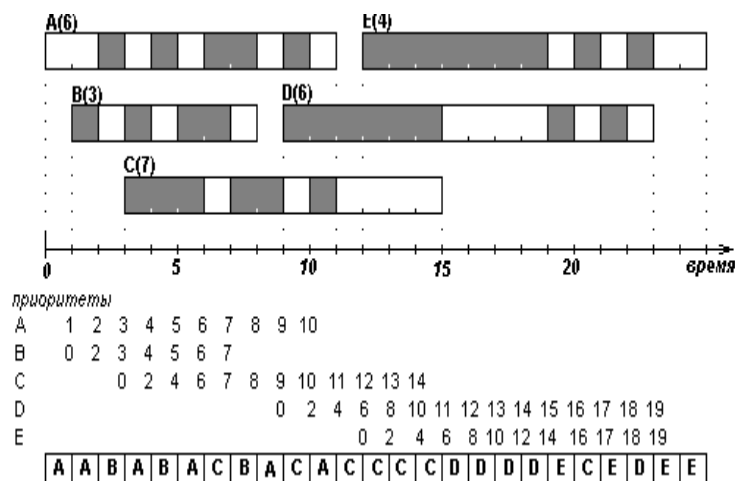


Рисунок 2.7 Планирование процессов по дисциплине SRR

Под временной шкалой здесь показаны текущие значения приоритетов процессов. Процесс А при поступлении получает приоритет 0. Поскольку на этот момент других процессов нет, процесс А начинает выполняться. Получив ЦП, процесс А попадает в категорию выбранных, поэтому при окончании кванта в момент 1 приоритет процесса А возрастает на 1. В момент 1 поступает процесс В, ему присваивается начальный приоритет 0, на текущий момент это ниже, чем приоритет А, поэтому ЦП остается у процесса А. По прошествии еще одного кванта, к моменту времени 2 приоритет процесса А увеличивается еще на 1 и становится равным 2, но приоритет процесса В как нового увеличивается на 2 и становится равным приоритету А. По принципу RR ЦП отдается процессу В, как дольше ожидающему. Процесс В теперь также становится выбранным, и в дальнейшем его приоритет растет медленнее. Поступающий позже новый процесс С имеет нулевой начальный приоритет и вынужден ждать 3 кванта, пока его приоритет не сравняется с

приоритетами выбранных процессов. Аналогичным образом происходит обслуживание и остальных поступающих процессов.

FB (foreground-background – передний-задний планы) – очередь готовых процессов расщепляется на две подочереди – очередь переднего плана и очередь заднего плана. Очереди обслуживаются по дисциплине RR, но очередь переднего плана имеет абсолютный приоритет: пока в ней есть процессы, очередь заднего плана не обслуживается. Новый процесс направляется в очередь переднего плана. Если процесс использовал установленное число  $N$  квантов в очереди переднего плана, но не завершился, он переводится в очередь заднего плана.

Обобщение дисциплины FB на  $n$  очередей с номерами  $0, 1, \dots, n-1$  и с абсолютными приоритетами, убывающими при возрастании номера очереди, носит название MLFB (multiplylevel feed back – многоуровневые очереди с обратной связью). Расщепление очереди готовых процессов на две и более подочереди обеспечивает селекцию процессов по длительности: более длинные процессы попадают в очереди с большими номерами и, соответственно, с меньшими приоритетами. Дисциплина MLFB очень эффективна для систем, работающих в интерактивном режиме.

На рисунке 2.8 показаны примеры работы MLFB для  $N=1$ . Под временной шкалой показаны состояния процессов в каждый момент времени: "а" – для активного процесса и номер очереди – для неактивного. Процесс А поступает в очередь 0 и, поскольку ЦП свободен, сразу же выбирается из нее на выполнение. После использования одного кванта времени ЦП процесс А переводится в очередь 1. В этот момент (момент 1) в очередь 0 поступает процесс В. Поскольку очередь 0 имеет более высокий приоритет, чем очередь 1, на выполнение выбирается процесс В. Процесс В после использования кванта (момент 2) попадает также в очередь 1. Поскольку в момент времени 2 очередь 0 пуста, обслуживается

очередь 1, из нее выбирается процесс А, который был поставлен в эту очередь раньше, чем процесс В. После этого кванта (момент 3) процесс А переходит в очередь 2, а в очереди 0 появляется новый процесс С, которому и будет отдан следующий квант. После этого кванта (момент 4) процесс С будет направлен в очередь 1. На этот момент времени мы имеем 3 процесса: процесс А – в очереди 2, процесс В – в очереди 1 и процесс С – в очереди 1. Обслуживается очередь 1, процесс В попал в эту очередь раньше, он получает следующий квант и т. д.

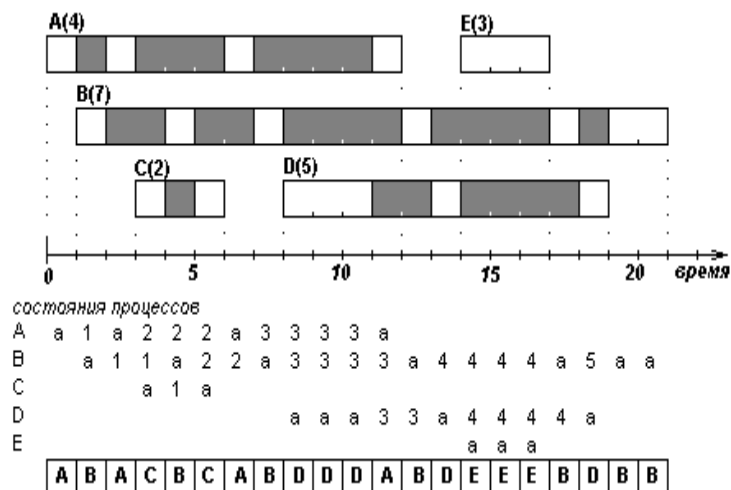


Рисунок 2.8 Планирование процессов по дисциплине MLFB

В простейшем варианте MLFB очередь с большим номером не обслуживается до тех пор, пока есть процессы в очередях с меньшими номерами. Возможны, однако, многочисленные вариации метода MLFB, например, такие:

- наряду с предпочтительным обслуживанием высокоприоритетной очереди предоставлять (но с меньшей частотой) кванты времени и очередям с низкими приоритетами;
- выполнять обратное перемещение процесса в очередь с меньшим номером после того, как процесс прождал установленный интервал времени в низкоприоритетной очереди;

- установить размер кванта зависящим от номера очереди, например:  $Q[n]=q*n$  или  $Q[n]=q*2^n$ ; поскольку в очереди с большими номерами попадают более длинные процессы, их обслуживание с большим квантом позволит сэкономить расходы на переключение;
- обслуживать разные очереди по разным дисциплинам (например: RR – для первой очереди, FCFS – для второй).

### **2.3. Планирование процессов в реальных системах**

Как мы отмечали выше, в реальных ОС при планировании процессорного времени применяются модификации и/или комбинации базовых алгоритмов, обеспечивающие большую эффективность и гибкость. Можно утверждать, что в реальных ОС применяются почти исключительно комбинированные методы, учитывающие как внешние приоритеты, так и поведение процесса, и степень загрузки ЦП, и, возможно, других ресурсов системы. Можно также утверждать, что дисциплины планирования без вытеснения в ОС общего назначения бесперспективны. Доживающая свой век Windows 3.x и Mac OS – последние из современных ОС, применяющая кооперативную многозадачность.

По-видимому, в ближайшее время наиболее интенсивно будут применяться и развиваться интерактивные ОС и ОС, обеспечивающие режим клиент/сервер, поэтому современные ОС применяют дисциплины, отдающие предпочтение обменным процессам. Для таких ОС достаточно типичной можно считать следующую макросхему определения приоритетов процессов в очереди к ЦП. Наивысший абсолютный приоритет имеют системные процессы, которые не могут вытесняться. Далее следуют системные процессы, которые могут быть вытеснены. Наконец, низший приоритет имеют пользовательские процессы.

Пользовательские процессы в свою очередь могут делиться на классы. Типовое деление включает в себя три класса:

- с высоким приоритетом – процессы реального времени;
- со средним приоритетом – интерактивные процессы;
- с низким приоритетом – счетные (пакетные) процессы.

Внутри каждого класса предусматривается еще несколько градаций приоритета, которые могут назначаться пользователем. Наконец, ОС может формировать еще динамическую добавку к приоритету, зависящую от истории выполнения процесса, текущего состояния ресурсов и т.д. Эта добавка может повышать или снижать приоритет процесса внутри класса, но, как правило, не выводит процесс за пределы назначенного ему класса. Динамическая составляющая совершенно необходима для процессов класса с нормальным приоритетом (интерактивных), так как их поведение во время выполнения наиболее трудно предсказать. Процессы других классов ОС может планировать и по статическим приоритетам.

Общие закономерности в динамическом вычислении приоритетов можно свести к следующим:

- приоритет процесса, долгое время находящегося в состоянии ожидания, повышается;
- приоритет процесса, часто выполняющего операции ввода-вывода, повышается;
- приоритет процесса, чаще получающего внешние сообщения и прерывания, повышается;
- если приоритет процесса не повышается, он убывает.

Ниже мы рассматриваем два примера динамического вычисления приоритетов. Еще раз подчеркнем, что рассматриваемые нами алгоритмы относятся только к пользовательским процессам: системные процессы имеют абсолютный и более высокий приоритет.

ОС Unix [33] – система многопользовательская и многозадачная, ориентированная, прежде всего, на интерактивную работу – дает пример изящного алгоритма динамического вычисления приоритетов, называемого иногда "алгоритмом полураспада" – модификацию дисциплины RR. С каждым  $i$ -м процессом связано некоторое приоритетное число  $P[i]$ . Чем оно меньше, тем выше приоритет процесса. Каждый новый процесс получает некоторое исходное значение приоритетного числа  $P_0$ , одинаковое для всех процессов. Кроме того, с каждым процессом связан счетчик процессорного времени  $U[i]$  с исходным значением 0. Процесс с наименьшим значением  $P[i]$  получает квант времени ЦП (при равенстве приоритетных чисел ЦП отдается процессу, ожидающему дольше). За время кванта интервальный таймер выдает несколько сигналов-прерываний. По каждому такому прерыванию счетчик  $U[i]$  активного (только активного!) процесса увеличивается на 1. Использование ЦП процессом заканчивается при истечении кванта или при переходе процесса в ожидание. При этом модифицируются счетчики процессорного времени всех (в том числе и неактивных) процессов:

$$U[i] = U[i] / 2$$

и для всех процессов перевычисляются приоритетные числа:

$$P[i] = P_0 + U[i] / 2.$$

На рисунке 2.9 показан пример работы алгоритма полураспада для случая трех, одновременно поступивших процессов: А, В, С. Для этого примера мы задались начальным значением приоритетного числа  $P_0=16$  и размером кванта, равным 16 "тикам" таймера.

время	активный процесс	процесс А		процесс В		процесс С	
		P[A]	U[A]	P[B]	U[B]	P[C]	U[C]
0	A	16	0	16	0	16	0
1	A	16	1	16	0	16	0
2	A	16	2	16	0	16	0
...	...	...	...	...	...	...	...
15	A	16	15	16	0	16	0
16	A	16	16	16	0	16	0
16	B	20	8	16	0	16	0
17	B	20	8	16	1	16	0
18	B	20	8	16	2	16	0
...	...	...	...	...	...	...	...
31	B	20	8	16	15	16	0
32	B	20	8	16	16	16	0
32	C	18	4	20	8	16	0
33	C	18	4	20	8	16	1
34	C	18	4	20	8	16	2
...	...	...	...	...	...	...	...
47	C	18	4	20	8	16	15
48	C	18	4	20	8	16	16
48	A	17	2	18	4	20	8
49	A	17	3	18	4	20	8
50	A	17	4	18	4	20	8
...	...	...	...	...	...	...	...
63	A	17	16	18	4	20	8
64	A	17	17	18	4	20	8
64	B	21	9	17	2	18	4
...	...	...	...	...	...	...	...

Рисунок 2.9 Пример применения алгоритма полураспада (Q=16; P0=16)

Поскольку Unix не накладывает ограничений на количество процессов, порождаемых одним пользователем, для ОС может оказаться более важным справедливое распределение ЦП не между процессами, а между пользователями. Эта задача решается незначительной модификацией алгоритма. С каждым процессом связывается еще и групповой счетчик процессорного времени  $G[i]$ . Этот счетчик с каждым "тиком" таймера увеличивается на 1 как в активном процессе, так и во всех процессах, принадлежащих тому же пользователю. В конце кванта  $G[i]$  также "полураспадается", а приоритетное число вычисляется как

$$P[i] = P0 + U[i] / 2 + G[i] / 2.$$

ОС VM/370 [28] демонстрирует нам значительно более сложный (но и более гибкий) пример планирования, рассчитанный на одновременное выполнение задач разных типов. Этот алгоритм можно рассматривать как некоторую версию дисциплины MLFB. Единицей планирования ЦП в этой ОС является виртуальная машина (ВМ). Планировщик ВМ определяет



последовательность использования ЦП виртуальными машинами и длительность этого использования. Последовательность определяется положением ВМ в очередях планировщика, длительность – величиной кванта и частотой его получения.

Планирование осуществляется, исходя из таких требований:

- равномерное (на некотором интервале времени) использование ЦП всеми ВМ;
- обеспечение гарантированного времени ответа при заданной загрузке системы;
- соблюдение нормативов потерь на страничный обмен (о страничном обмене – см. главу 3).

Для выполнения этих требований планировщик периодически вычисляет затраты на страничный обмен и среднее время использования ЦП одной ВМ, а также постоянно ведет для каждой ВМ учет использованного ею процессорного времени и времени пребывания в очередях.

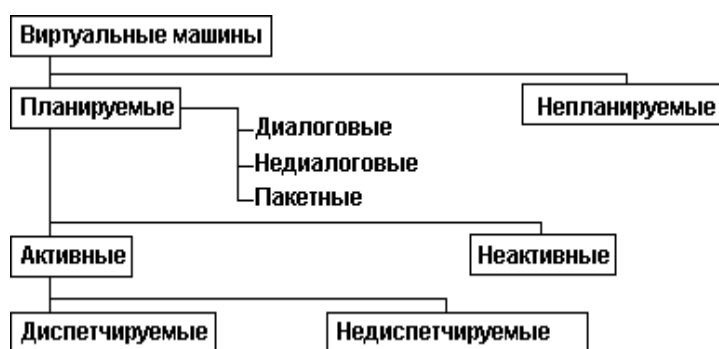


Рисунок 2.10 Состояния виртуальных машин в ОС VM/370

С точки зрения планировщика ВМ может находиться в одном из состояний, показанных на рисунке 2.10.

Непланируемыми называются ВМ, ожидающие завершения операции ввода-вывода на реальном внешнем устройстве или какого-либо другого

внешнего события. Непланируемые ВМ исключаются из очередей планировщика.

Планируемые – все остальные ВМ – могут быть активными или неактивными. Активной является ВМ, попавшая в очередь на обслуживание RUNLIST. Размер этой очереди ограничен соображениями эффективности страничного обмена. Все ВМ, не попавшие в эту очередь из-за ее ограниченности, являются неактивными. По мере разгрузки очереди RUNLIST она пополняется из очередей неактивных ВМ. Активные ВМ, в свою очередь, подразделяются на диспетчируемые и недиспетчируемые. Диспетчируемые ВМ – это те, которые полностью готовы получить ЦП. Недиспетчируемой является ВМ, для которой:

- моделируется выполнение привилегированной команды;
- или моделируется выполнение операции ввода-вывода без связи с реальным устройством (см. главу 6);
- или обрабатывается страничный отказ.

Кроме этой, основной классификации, планируемые ВМ подразделяются на диалоговые, недиалоговые и чисто пакетные. Для некоторых статусов ВМ установлены следующие обозначения:

- Q1 – диалоговые активные;
- Q2 – недиалоговые активные;
- Q3 – пакетные активные;
- E1 – диалоговые неактивные;
- E2 – недиалоговые неактивные.

Все активные ВМ находятся в очереди RUNLIST, но статус ВМ влияет на ее положение в очереди. Для неактивных ВМ существуют две разные очереди: очередь E1 и очередь E2.

При пополнении очереди RUNLIST абсолютный приоритет имеет очередь E1, BM из очереди E2 переводятся в RUNLIST только, если очередь E1 пуста.

Новые BM (не показано на рисунке) вначале поступают в очередь RUNLIST, а при ее заполнении – в очередь E2. При попадании BM в очередь RUNLIST ей назначается размер кванта  $dt$  и квота обслуживания  $dT$  – интервал времени ЦП, который BM может использовать, планируясь из очереди RUNLIST. Начальное значение кванта устанавливается равным фиксированному значению  $dt_0$ , в дальнейшем оно может быть изменено по таким правилам:

- квант сохраняется равным  $dt_0$ , если на предыдущем кванте не было прерывания по вводу-выводу,
- квант назначается равным  $4 * dt_0$  в противном случае.

Квота обслуживания назначается:

- $8 * dt$  для BM статуса Q1;
- $64 * dt$  для BM статуса Q2;
- $512 * dt$  для BM статуса Q3.

Таким образом, диалоговые BM имеют меньшие кванты, чем недиалоговые, но получают их чаще.

Очередность предоставления ЦП диспетчируемым BM определяется связанным с каждой BM приоритетным числом (чем оно меньше, тем выше приоритет BM). Начальное значение приоритетного числа определяется временем поступления BM в систему. Таким образом, та BM, сеанс на которой начался раньше, имеет более высокий приоритет. В дальнейшем планировщик формирует динамическую добавку к приоритетному числу, которая может его существенно изменять. Величина добавки зависит от поведения BM, которое мы рассмотрим, обращаясь к

схеме на рисунке 2.11, где показана схема движения ВМ между ЦП и очередями планировщика.

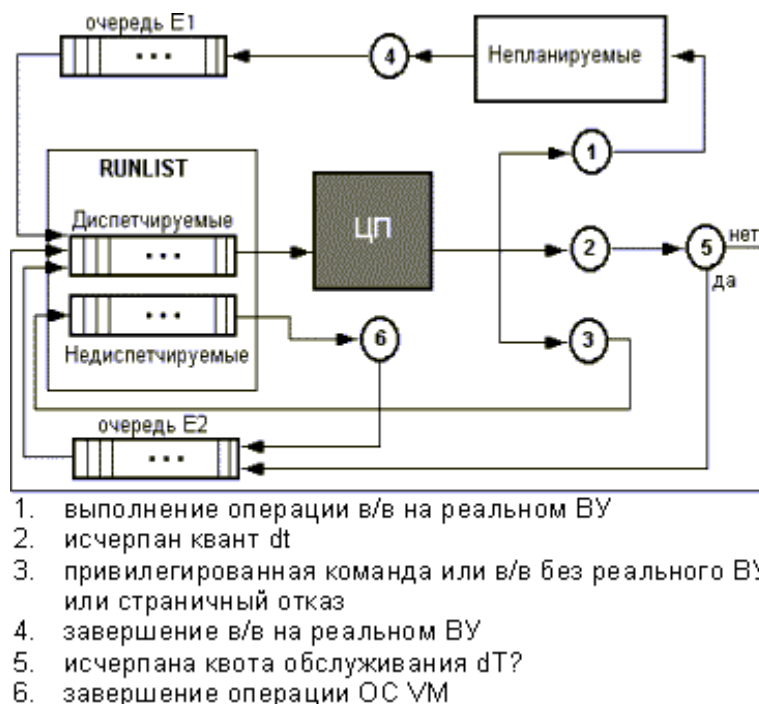


Рисунок 2.11 Планирование виртуальных машин в ОС VM/370

Из диспетчируемых ВМ в очереди RUNLIST выбирается ВМ с высшим приоритетом, и ей выделяется квант времени ЦП –  $dt$ . ВМ может освободить ЦП по одной из следующих причин:

- ВМ запрашивает операцию ввода-вывода, выполняющуюся на реальном внешнем устройстве (1 на рисунке 2.11), такая ВМ становится непланируемой и исключается из очередей планировщика;
- ВМ исчерпала квант времени ЦП (2 на рисунке 2.11) – для этого случая проверяется, исчерпала ли ВМ квоту обслуживания  $dt$  (5 на рисунке 2.11); если квота не исчерпана, ВМ возвращается в очередь RUNLIST, но ее приоритетное число несколько увеличивается; если же квота исчерпана, ВМ получает статус недиалоговой и направляется в очередь E2;

- ВМ запрашивает операцию, которую моделирует для нее ОС VM без использования реального внешнего устройства, или для ВМ обрабатывается страничный отказ (3 на рисунке 2.11), такая ВМ переводится в состояние ожидания (устанавливается соответствующий бит в ее виртуальном PSW), она остается в очереди RUNLIST, но становится недиспетчируемой.

ВМ, ставшие непланируемыми, ожидают в других очередях ОС, которые не имеют отношения к планировщику. Когда завершается операция ввода-вывода для такой ВМ (4 на рисунке 2.11), эта ВМ получает статус диалоговой и направляется в очередь E1. При этом приоритетное число ВМ перевычисляется с учетом:

- старого приоритета;
- времени предыдущего ухода ВМ из очереди RUNLIST;
- времени, потерянного ВМ в очередях планировщика.

Новое значение приоритета определяет порядок выборки ВМ из очереди E1 в очередь RUNLIST и сохраняется за ВМ при переводе ее в очередь RUNLIST.

ВМ, получившие статус недиспетчируемых, ожидают, когда ОС переведет их виртуальное PSW из состояния ожидания в состояние счета (6 на рисунке 2.11). После этого такая ВМ переводится в очередь E2. Таким образом, ВМ может попасть в очередь E2 либо по исчерпанию квоты обслуживания, либо по выполнению операций ОС. При постановке в очередь E2 приоритетное число перевычисляется с учетом:

- старого приоритета;
- эффективности использования ВМ памяти;
- времени пребывания ВМ в очередях;
- штрафа, накладываемого на ВМ, если она превысила среднее время использования процессора.

Те ВМ, которые 6 раз переходили из очереди E2 в очередь RUNLIST, минуя очередь E1, получают статус чисто пакетных и добавка к приоритетному числу для них в 8 раз больше, чем для диалоговых.

Всякий раз, когда какая-либо ВМ покидает очередь RUNLIST, ОС пытается пополнить последнюю из очередей неактивных ВМ. Возможность пополнения очереди RUNLIST определяется эффективностью управления памятью в соответствии с политикой "размера рабочего набора", рассматриваемой в следующей главе.

В современных версиях ВМ сохранились основные черты приведенного алгоритма, но развитие аппаратных средств мейнфреймов и передача им некоторых задач управления производительностью позволили значительно его упростить.

## **2.4. Другие уровни планирования**

Выше мы сосредоточились только на краткосрочном планировании. Методы, рассмотренные нами, могут применяться и на других уровнях планирования. Не всегда, правда, можно провести четкую границу между уровнями планирования. Те или иные методы вычисления приоритета доступа к другим (кроме ЦП) ресурсам могут использоваться для формирования динамической добавки к приоритету процесса в очереди готовых процессов или/и влиять на параметры дисциплины планирования (как мы видели для ОС VM/370, где в планировании ВМ учитываются и соображения управления памятью).

В тех случаях, когда среднесрочное планирование осуществляется отдельными планировщиками соответствующих ресурсов, применяются обычно базовые дисциплины планирования без вытеснения, поскольку планируемые ресурсы часто не являются повторно используемыми. Дисциплина SJR применяется обычно к тем ресурсам, которые являются

для системы узким местом, для повышения пропускной способности; дисциплина FCFS – в тех случаях, когда крайне важно избежать бесконечного откладывания. При среднесрочном планировании ведущую роль играют соображения предупреждения тупиков, рассматриваемые нами в главе 4.

Долгосрочное планирование может также рассматриваться как вариант среднесрочного: новый процесс ожидает получения ресурсов (а таким ресурсом может быть и свободная запись в системной таблице процессов). В явном виде долгосрочное планирование выполняется в системах пакетной обработки и на уровне не процессов, а заданий. Пакетное задание (batch job) – единица работы с точки зрения пользователя. Задание подразумевает выполнение одного или нескольких процессов. В долгосрочном планировании ведущую роль играют внешние приоритеты, назначаемые пользователем и администратором. Дисциплины обслуживания очереди заданий могут меняться в зависимости от характеристик потока задач, решаемых системой, от привилегий работающих в системе пользователей, от времени суток. Так, для вычислительных центров, работавших в пакетном режиме, было характерным обслуживание в дневное время коротких заданий по дисциплине SJN – чтобы обслужить максимальное число пользователей в течение рабочего дня, а в ночное время – счет длинных заданий, выбираемых по дисциплине FCFS – чтобы обеспечить минимальные потери процессорного времени.

## **КОНТРОЛЬНЫЕ ВОПРОСЫ**

1. Какие требования предъявляются к дисциплинам планирования процессов? Назовите те пары требований, которые кажутся вам взаимоисключающими.

2. Если ОС применяет дисциплину планирования с абсолютными приоритетами, то каким образом может получить процессор низкоприоритетный процесс?
3. Каковы показатели эффективности планирования процессов? Поясните их смысл для пользователя. Какие характеристики процессов являются существенными с точки зрения планирования?
4. В чем состоят достоинства, дисциплины FCFS, обеспечивающие ее частое применение?
5. Дисциплина планирования RR обеспечивает практически постоянное значение показателя  $P(t)$  на широком диапазоне значений  $t$ , но на краях этого диапазона значение  $P(t)$  резко возрастает. Объясните причину этого.
6. Предложите различные варианты обслуживания очередей на разных уровнях для дисциплины MLFB и обоснуйте их.
7. С каким из базовых алгоритмов планирования вы можете связать алгоритм полураспада, применяемый в Unix?
8. По какой причине процесс, готовый к выполнению может быть исключен из очереди претендентов на выполнение?
9. Должен ли приоритет процесса в очереди на выполнение использоваться и в других очередях? Приведите соображения "за" и "против".
10. В чем преимущества и недостатки вытесняющих дисциплин планирования по сравнению с кооперативными?



## **Глава 3. Управление памятью**

### **3.1. Виртуальная и реальная память**

Мультипрограммирование будет эффективным только в том случае, когда несколько процессов одновременно находятся в оперативной памяти, тогда переключение процессов не требует значительного перемещения данных между оперативной и внешней памятью. Но тогда на ОС возлагается задача распределения оперативной памяти между процессами и защиты памяти, которая выделена процессу, от вмешательства другого процесса. Таким образом, память является одним из важнейших ресурсов системы, и от эффективности функционирования менеджера этого ресурса в значительной степени зависят показатели эффективности всей системы в целом.

Процессор обрабатывает данные, которые находятся в оперативной памяти, и процессы размещают свои коды и данные в адресном пространстве, которое они рассматривают как пространство оперативной памяти. В очень редких случаях программист задает при разработке программы реальные адреса в оперативной памяти, в большинстве же случаев между программистом и средой выполнения его программы стоит тот или иной аппарат преобразования адресов. В общем случае то адресное пространство, в котором пишется программа, называется виртуальной памятью в отличие от реальной или физической памяти, в которой происходит выполнение программы (процесса). Работу с памятью можно

представить в виде трех функций преобразования, которые показаны на Рисунке 3.1.

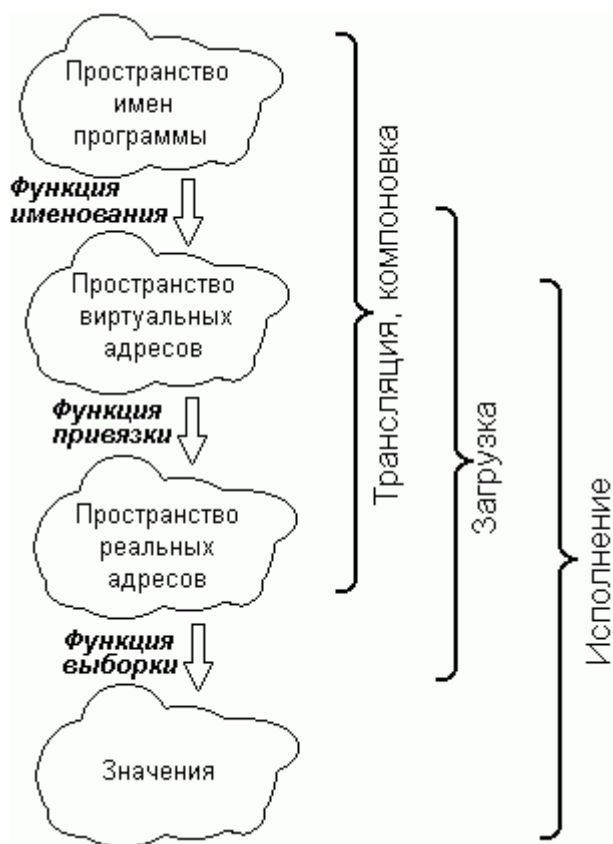


Рисунок 3.1 Функции управления памятью

Функция именования производит отображение точки из пространства имен программы в пространство адресов в виртуальной памяти, иными словами – переводит символьные имена, используемые программистом, в виртуальные адреса.

Функция привязки производит отображение точки из пространства виртуальных адресов в пространство реальных адресов, то есть переводит виртуальные адреса в адреса физических ячеек памяти.

Функция выборки отображает точку из пространства реальных адресов в значение, то есть выбирает содержимое памяти по заданному адресу.

Функция именования реализуется по большей части обслуживающими программами, мы рассматриваем ее в следующей главе. Функция выборки всегда реализуется аппаратно. В данной главе нас будет интересовать, прежде всего, функция привязки адресов. Относительно нее конструктором ОС должен быть решен основной вопрос: на каком этапе подготовки/выполнения программы ее выполнять?

Программист может писать программу, сразу привязывая ее к заведомо известным адресам физической памяти, – это называется программированием в абсолютных адресах. Такое программирование выполняется в специфических случаях, например, для программ, записываемых в ПЗУ. Даже в таких случаях программист часто пользуется символическими именами, возлагая задачу перевода имен в физические адреса на транслятор. Полученная таким образом программа называется абсолютной или перемещаемой. Она может выполняться, только будучи загруженной по определенному адресу оперативной памяти.

Все прикладные программы и подавляющее большинство системных программ являются перемещаемыми. Это значит, что в программе, подготовленной к выполнению (в том образе программы, который хранится на внешней памяти), обращения к памяти настроены на виртуальные адреса, не привязанные пока к адресам реальной памяти.

Выполнение функции привязки адресов может быть перенесено на этап загрузки. Простейшим вариантом такой системы трансляции является тот, в котором виртуальный адрес представляет собой смещение относительно начала программы, а при загрузке программы в память ко всем виртуальным адресам прибавляется начальный адрес области, в которую программа загружена. Другой вариант – виртуальная адресация производится относительно содержимого некоторого базового регистра, в который при загрузке заносится базовый адрес. Программы, подготовленные таким образом, называются перемещаемыми при загрузке

– они могут быть загружены в любую область оперативной памяти, но после загрузки должны оставаться на том же месте в памяти.

Наконец, привязка адресов может делаться уже на этапе выполнения программы. Программа (процесс) обращается к памяти только по виртуальным адресам, а перевод виртуального адреса в реальный производится только при обращении по этому адресу. Между двумя обращениями по одному и тому же виртуальному адресу процесс может быть перемещен операционной системой в другую область реальной памяти. Системные средства, ответственные за привязку адресов, будут "знать" о произведенном перемещении и при втором обращении привяжут тот же виртуальный адрес к новому реальному адресу. Поскольку виртуальный адрес не изменился, то перемещение окажется совершенно прозрачным для процесса; такие процессы называются перемещаемыми динамически. При такой трансляции адресов привязка выполняется при каждом обращении к памяти, то есть очень часто. Поэтому естественным решением является выполнение этой функции на аппаратном уровне. Аппаратура переводит виртуальные адреса в реальные, используя некоторые таблицы трансляции. Подготовка таблиц трансляции адресов – операция, выполняемая для процесса одноразово, их модификация производится также нечасто (при перемещении процесса), поэтому задача формирования и модификации таблиц трансляции возлагается на ОС.

Отметим, что иногда виртуальной памятью называют именно эти свойства аппаратуры вычислительной системы и вытекающие из них возможности для процессов работать виртуальным адресным пространством большего размера, чем размер имеющейся в системе реальной памяти. Мы же следуем более широкой интерпретации [17]: виртуальная память это то адресное пространство, в котором разрабатывается процесс. Такое понимание соответствует определению ОС с точки зрения пользователя, которое мы дали в разделе 1.4. В данном

случае ОС скрывает от процесса организацию низкоуровневого ресурса (реальной памяти) и конструирует ресурс более высокого уровня (виртуальная память), более удобный в обращении. Такая интерпретация не принимает во внимание, на каком этапе – загрузки или выполнения – производится трансляция адресов, и имеется ли в системе аппаратная поддержка этой трансляции. В частном случае размер виртуальной памяти может быть и меньше реальной.

Разработка программ, работающих в виртуальном адресном пространстве, имеет целый ряд преимуществ, которые можно сгруппировать по трем основным направлениям.

1. Удобство для программиста. Программист имеет в своем распоряжении виртуальную память, представляющую собой адресное пространство либо совершенно плоское – с адресами, линейно возрастающими от 0 до максимального значения, либо сегментированное в соответствии с потребностями задачи. При этом он не заботится о том, как будет размещен его процесс в реальной памяти. ОС может отобразить виртуальную память в реальную даже таким образом, что смежные участки виртуального адресного пространства будут отображаться в несмежные участки реальной памяти. Каждый процесс разрабатывается в собственном адресном пространстве, независимом от пространств других процессов.

2. Реорганизация памяти. Системные средства управления памятью могут выбрать такое отображение виртуальной памяти в реальную, которое обеспечит максимально эффективное использование ресурса реальной памяти. В случае, когда обеспечивается динамическая трансляция адресов, реорганизация ресурса реальной памяти может производиться и в ходе выполнения процесса, причем совершенно прозрачно для последнего.

3. Защита. Процесс никак не может обратиться за пределы своего виртуального адресного пространства. Если ОС обеспечивает отображение таким образом, что виртуальные пространства двух любых процессов не могут перекрывать друг друга в реальной памяти, то никакой процесс не будет иметь доступа к адресному пространству другого процесса.

При наличии аппаратной поддержки системы виртуальной памяти позволяют работать с виртуальными адресными пространствами, размер которых превышает доступный размер оперативной памяти. Это достигается за счет хранения части программ и данных на внешней (дисковой) памяти и управления миграцией данных между оперативной и внешней памятью. Современные вычислительные системы следуют тенденции вводить кэширование между двумя этими уровнями памяти. В ОС Unix, например, тотальное кэширование обеспечивается самой ОС для любого обмена с дисками, следовательно, работает оно и для данных, перемещаемых при управлении памятью. В вычислительных системах ESA имеется два аппаратных промежуточных уровня памяти: расширенная память, которая включается в пространство реальных адресов, но используется только как буфер обмена и как собственная буферная память дискового запоминающего устройства. (Расширенная память, может быть выполнена как на тех же элементах, что и основная оперативная память, так и на специальных, – не столь быстродействующих, но дешевых).

Многоуровневая память строится обычно по иерархическому принципу. Это означает, что для каждого следующего уровня время доступа больше, чем для предыдущего:  $t[i] > t[i-1]$ , и объем больше:  $V[i] > V[i-1]$ . Последнее обстоятельство делает возможным дублирование информации на уровнях: если данные имеются на  $i$ -м уровне, то их копии сохраняются и на всех уровнях с большими номерами. Обозначим через  $h[i]$  отношение присутствия – вероятность того, что данные, запрошенные на  $i$ -м уровне памяти, уже имеются на этом уровне.

Если мы имеем  $n$  уровней памяти, то для  $n$ -го уровня отношение присутствия равно 1 и среднее время доступа  $\tau[n]$  совпадает с  $t[n]$ . Для всех уровней с меньшими номерами среднее время доступа может быть определено рекурсивно:

$$\tau[i] = h[i] * t[i] + (1 - h[i]) * \tau[i-1].$$

На программном уровне мы не можем воздействовать ни на  $t[i]$ , ни на  $V[i]$ , которое в значительной степени определяет и  $h[i]$ . Но мы можем влиять на величину  $h[i]$ , выбирая для хранения на уровне с меньшим номером только те данные, обращение к которым производится наиболее часто.

В общем случае проектирование менеджера памяти в составе ОС требует выбора трех основных стратегий:

- Стратегии размещения: какую область реальной памяти выделить процессу; как вести учет свободной/занятой реальной памяти?
- Стратегии подкачки: когда размещать процесс (или часть его) в реальной памяти?
- Стратегии вытеснения: если реальной памяти не хватает для удовлетворения очередного запроса, то у какого процесса отобрать ранее выделенный ресурс реальной памяти (или часть его)?

Ниже мы рассмотрим способы реализации этих стратегий для различных моделей памяти. Порядок рассмотрения будет соответствовать принципу "от простого к сложному" и в основном отображать также и историческое развитие моделей памяти:

- фиксированные разделы – модель, не использующая аппаратную трансляцию адресов;
- односегментная виртуальная память – развитие фиксированных разделов для аппаратной трансляции адресов;

- модели виртуальной памяти, использующие развитые средства аппаратной трансляции адресов;
  - многосегментная;
  - страничная;
  - комбинированная сегментно-страничная;
- модели виртуальной памяти, представляющие собой возврат к простым моделям, но на более высоком уровне:
  - плоская;
  - одноуровневая.

### **3.2. Фиксированные разделы**

Эта модель памяти применяется в вычислительных системах, не имеющих аппаратных средств трансляции адресов. Процесс загружается в непрерывный участок памяти (раздел), привязка адресов выполняется при загрузке. Размер раздела равен размеру виртуального адресного пространства процесса, который, следовательно, не может превышать размера доступной реальной памяти. Процесс в ходе своего выполнения может выдавать запросы на выделение/освобождение памяти. Все эти запросы удовлетворяются только в пределах виртуального адресного пространства процесса, а, следовательно, – в пределах выделенного ему раздела реальной памяти.

Примером ОС, работающей в такой модели памяти, может быть OS/360, ныне уже не применяемая, но существовавшая в двух основных вариантах: MFT (с фиксированным числом задач) и MVT (с переменным числом задач). В первом варианте при загрузке ОС реальная память разбивалась на разделы оператором. Каждая задача (процесс) занимала один раздел и выполнялась в нем. Во втором варианте число разделов и их



положение в памяти не было фиксированным. Раздел создавался в свободном участке памяти перед началом выполнения задачи и имел размер, равный объему памяти, заказанному задачей. Созданный раздел фиксировался в памяти на время выполнения задачи, но уничтожался при окончании ее выполнения.

В более общем случае для процесса может выделяться и несколько разделов памяти, причем их выделение/освобождение может выполняться динамически (пример – MS DOS). Однако общими всегда являются следующие правила:

- раздел занимает непрерывную область реальной памяти;
- выделенный раздел фиксируется в реальной памяти;
- после выделения раздела процесс работает с реальными адресами в разделе.

Задача эффективного распределения памяти (в любой ее модели) сводится, прежде всего, к минимизации суммарного объема дыр. Ниже мы даем определения дыр, общие для всех моделей памяти.

Дырой называется область реальной памяти, которая не может быть использована. Различают дыры внешние и внутренние. Рисунок 3.2 иллюстрирует внешние и внутренние дыры в системе OS/360.

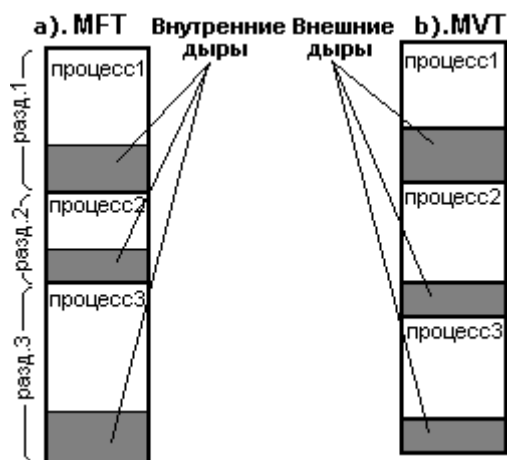


Рисунок 3.2 Разделы в реальной памяти OS/360

Внешней дырой называется область реальной памяти, которая не выделена никакому процессу, но слишком мала, чтобы удовлетворить запрос на память. На рисунке 3.2.а суммарный размер свободных областей превышает запрос, но каждая из этих областей в отдельности меньше запроса, поэтому все эти свободные области являются внешними дырами.

Внутренней дырой называется память, которая выделена процессу, но им не используется. Так, на рисунке 3.2.б процессу 1 выделен раздел P1, но виртуальное адресное пространство процесса меньше размера раздела, оставшееся пространство раздела составляет внутреннюю дыру.

Для управления памятью формируются те или иные управляющие структуры (заголовки), которые также занимают память. В некоторых системах общий объем заголовочной памяти может быть очень большим, и в таких случаях следует учитывать также и заголовочные дыры – области памяти, которые содержат не используемую в данный момент управляющую информацию. В системах с реальной памятью заголовочные дыры практически отсутствуют.

Хотя в любом случае невозможность использовать память является потерей ресурса, внешние дыры, являются "меньшим злом", так как, во-первых, блок свободной памяти становится дырой из-за малого своего размера, а "малый" – понятие относительное, во-вторых, в моделях памяти с динамической трансляцией адресов существуют методы борьбы с внешними дырами.

Модель с фиксированными разделами представляет весьма ограниченную версию управления памятью. Вытеснение здесь вообще не реализуется, процесс, которому не хватает памяти, просто блокируется до освобождения требуемого ресурса (в OS/360 MFT можно было наблюдать даже такое явление: задача, требующая объема памяти, превышающего размер раздела, просто "запирала" этот раздел до перезагрузки системы).

Стратегия подкачки здесь примитивная: весь процесс размещается в реальной памяти при его создании. В варианте MFT практически отсутствует и стратегия размещения: процесс размещается с начала раздела, а решение о размещении раздела и о распределении задач по разделам принимает оператор. А вот в варианте MVT, поскольку границы разделов не зафиксированы, ОС необходимо принимать решение о размещении. В этой ОС уровень мультипрограммирования был невысок (обычно 4 - 5 процессов), поэтому стратегия размещения принималась простейшая, а вот в системах с более высоким уровнем мультипрограммирования и, следовательно, со значительной фрагментацией памяти может оказаться целесообразным выбор более сложной и гибкой стратегии.

Первый вопрос, решаемый в стратегии размещения, – способ представления свободной памяти. Существует два основных метода такого представления: списки свободных блоков и битовые карты. В первом методе ОС из свободных блоков памяти организует линейный список и хранит адрес начала этого списка. При обработке такого списка должна учитываться необходимость слияния двух смежных свободных блоков в один свободный блок суммарного размера. Эта задача может быть существенно упрощена, если список упорядочивается по возрастанию адресов блоков. Во втором методе память условно разбивается на некоторые единицы распределения (параграфы) и создается "карта памяти" (memory map) – битовый массив, в котором каждый бит соответствует одному параграфу памяти и отображает его состояние: 1 – занят, 0 – свободен. Поиск свободного блока требуемого размера сводится к поиску цепочки нулевых бит требуемой длины. Выбор размера параграфа определяется компромиссом между потерями памяти на внутренних дырах (при большом размере параграфа) и потерями на размещение в памяти самой карты (при малом размере параграфа).

При не очень большой фрагментации памяти списки свободных блоков занимают меньше места, чем карта, но их обработка несколько сложнее.

Выбранная структура представления свободной памяти не ограничивает выбор стратегии размещения.

Простейшей стратегией размещения является стратегия "первый подходящий" (first hit): просматривается список свободных блоков (или карта памяти) и выбирается первый же найденный блок, размер которого не меньше требуемого. Если размер найденного блока превышает запрошенный, то оставшаяся его часть оформляется как свободный блок. При всей своей простоте эта стратегия дает неплохие результаты и применяется в большинстве систем с фиксированными разделами и с сегментацией (см. ниже). При значительной фрагментации алгоритм может быть модифицирован кольцевым поиском. Если всякий раз поиск начинается с начала списка/карты, то маленькие свободные участки будут накапливаться в списке/карте и для нахождения свободного блока значительной длины надо будет сначала выбрать и отбросить много маленьких блоков. При кольцевом поиске поиск всякий раз начинается с того места, на котором он закончился в прошлый раз, а при достижении конца списка/карты – продолжается с самого начала. Таким приемом сокращается среднее время поиска.

Другой несложной стратегией является "самый подходящий" (best hit): просматривается весь список свободных блоков (или карта памяти) и выбирается блок, размер которого равен запросу или превышает его на минимальную величину. Хотя, на первый взгляд, этот метод может показаться более эффективным, он дает в среднем худшие результаты, чем "первый подходящий". Во-первых, это объясняется тем, что здесь следует обязательно просмотреть весь список или карту; во-вторых, тем, что здесь более интенсивно накапливаются внешние дыры, так как остатки от

"самых подходящих" блоков оказываются маленького размера. Существенным же преимуществом этой стратегии является то, что она охраняет большие свободные блоки от "разбазаривания по мелочам".

Первый недостаток метода "самый подходящий" может быть преодолен при упорядочении списка свободных блоков по возрастанию размеров: полный просмотр списка тогда не потребуются, но будет усложнена задача слияния смежных блоков. Еще более ускоряет поиск разбиение списка свободных блоков на несколько подсписков, каждый из которых содержит блоки, размер которых ограничен сверху и снизу. Поиск ускоряется за счет просмотра только подсписка, соответствующего запросу, если в подсписке выбирать первый подходящий, то отчасти сглаживается и острота проблемы внешних дыр. При представлении свободной памяти в виде карты аналогичный эффект может быть получен при создании нескольких карт разного масштаба и поиске по той карте, масштаб которой соответствует запрошенному размеру блока.

Поскольку модель с фиксированными разделами ограничивает виртуальное адресное пространство размерами реальной памяти, возникает проблема больших программ, не вмещающихся в доступную память. Преодоление этого ограничения достигается созданием программ с оверлейной (overlay – перекрытие) структурой. Структура межмодульных вызовов оверлейной программы имеет вид дерева. В корне дерева (нулевой уровень) находится модуль-монитор, из которого происходят обращения к модулям, расположенным на ветвях дерева. Каждый модуль первого уровня в свою очередь может быть корнем поддеревя и т.д. Принцип оверлейности или перекрытия заключается в том, что ветви дерева занимают одни и те же области в виртуальном адресном пространстве программы. Поскольку модули, расположенные в разных ветвях дерева, не могут выполняться одновременно, то только монитор является резидентным в оперативной памяти все время выполнения программы,

ветви же сменяют друг друга в одной и той же области памяти. На самом деле построить программу, имеющую идеальную древовидную структуру, довольно трудно, поскольку при этом обязательно догматическое соблюдение дисциплины программирования "сверху вниз". Необходимо тщательно следить, чтобы ветви дерева не пересекались: никакой модуль не имеет права передавать управление в модуль, расположенный на другой ветви, или обращаться к данным, определенным в модуле другой ветви. Реальные программы обычно нарушают эти правила, так как при их разработке используется комбинация дисциплин "сверху вниз" и "снизу вверх", за счет чего появляются низкоуровневые процедуры, обращения к которым производится из любых ветвей дерева. Выход из противоречия состоит либо в помещении этих процедур в корневой модуль, либо создании еще одной или нескольких резидентных областей для размещения модулей, содержащих эти процедуры. В системах, поддерживающих развитые средства создания оверлейных программ (та же OS/360), выполнение функции именования – отображения имен входных точек и общих переменных на виртуальное адресное пространство возлагается на редактор связей (link editor), который и формирует содержимое адресного пространства процесса в соответствии с оверлейной структурой, описываемой программистом с помощью специального языка. На Рисунке 3.3 представлен пример оверлейной структуры программы.

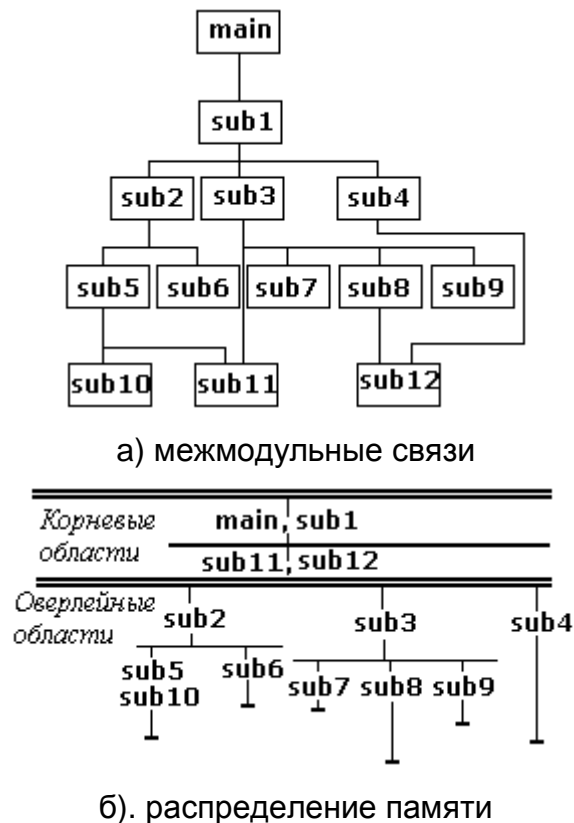


Рисунок 3.3 Пример оверлейной структуры программы

В модели с фиксированными разделами после загрузки процесса все обращения к памяти в нем производятся по реальным адресам. Следовательно, ничто не может помешать процессу обратиться к памяти, принадлежащей другому процессу или даже самой ОС, и последствия такого обращения могут быть фатальными. Защита от такого доступа должна поддерживаться аппаратно. Приведем два примера такой защиты.

1) В вычислительной системе имеются специальные регистры, в которых содержатся значения верхней и нижней границ адресов, к которым доступ разрешается. При любом обращении к памяти аппаратура сравнивает заданный адрес с этими граничными значениями и, если адрес не лежит в границах, выполняет прерывание-ловушку. При размещении процесса в памяти ОС определяет для него эти границы и записывает их значения в контекст процесса (вектор состояния процессора). При переключении контекста на данный процесс эти значения загружаются в

регистры границ и таким образом ограничивается область памяти, к которой имеет доступ процесс. В контексте самой ОС граничные адреса, естественно, соответствуют всему имеющемуся объему памяти.

2) Реальная память разбивается на "блоки защиты" одинакового размера. С каждым таким блоком связан некоторый код – ключ. При размещении процесса ОС дает ему какой-то ключ доступа, сохраняет ключ в векторе состояния и присваивает тот же ключ всем блокам памяти, выделенным этому процессу. При любом обращении к памяти аппаратура сравнивает ключ текущего процесса с ключом блока, к которому производится обращение и выполняет прерывание-ловушку, если эти ключи не совпадают. Сама ОС, конечно же, имеет "универсальный ключ" дающий ей доступ к любому блоку.

### **3.3. Односегментная модель**

Нам неизвестны ОС, поддерживающие односегментную модель "в чистом виде", но ее рассмотрение облегчит понимание более сложных моделей.

Внешне (с точки зрения программиста) эта модель очень похожа на модель с фиксированными разделами: программа-процесс готовится в плоском виртуальном адресном пространстве. Процесс занимает непрерывное пространство виртуальной памяти, и в реальную память он также загружается в один непрерывный раздел (сегмент). Сегмент может начинаться с любого адреса реальной памяти и иметь любой размер, не превышающий, однако, размера реальной памяти. Существенное отличие сегментной модели состоит в том, что она использует аппаратную динамическую трансляцию адресов. Загруженный в реальную память и выполняющийся процесс продолжает обращаться к памяти, используя виртуальные адреса, и лишь при каждом конкретном обращении



виртуальный адрес аппаратно переводится в реальный. В вычислительной системе, поддерживающей односегментную модель, должен существовать регистр дескриптора сегмента, содержимое которого состоит из двух полей: начального (базового) адреса сегмента в реальной памяти и длины сегмента. Когда процесс размещается в памяти, для выделенного ему сегмента формируется дескриптор, который записывается в вектор состояния в контексте процесса. При переключении контекста дескриптор сегмента загружается в аппаратный регистр дескриптора сегмента и служит той "таблицей трансляции", по которой аппаратура переводит виртуальные адреса в реальные. Сама трансляция адресов происходит по простейшему алгоритму. Поскольку виртуальное адресное пространство процесса представляет собой линейную последовательность адресов, начинающуюся с 0, виртуальный адрес является простым смещением относительно начала сегмента. Реальный адрес получается сложением виртуального адреса с базовым адресом, выбранным из дескриптора сегмента, как показано на рисунке 3.4. Единственный путь для выхода процесса за пределы своего виртуального адресного пространства – задание виртуального адреса, большего, чем размер сегмента. Этот путь легко может быть перекрыт, если аппаратура при трансляции адресов будет сравнивать виртуальный адрес с длиной сегмента и выполнять прерывание-ловушку, если виртуальный адрес больше.

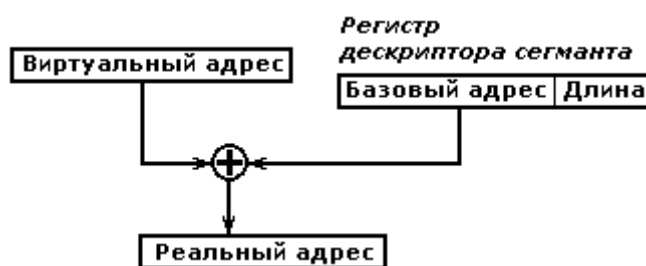


Рисунок 3.4 Односегментная модель

То обстоятельство, что процесс работает в виртуальных адресах, делает возможным перемещение сегментов в реальной памяти. Переместив процесс в другую область реальной памяти, ОС просто изменяет поле базового адреса в дескрипторе его сегмента. Поскольку, как и в модели с фиксированными разделами, реальная память распределяется непрерывными блоками переменной длины, здесь применяются те же стратегии размещения. Но возможное здесь перемещение сегментов является эффективным способом борьбы с внешними дырами. Сегменты переписываются в реальной памяти таким образом, чтобы свободных мест между ними не оставалось, все свободное пространство сливается в один большой свободный блок и, таким образом, оказывается доступным для последующего распределения.

Другой возможностью, которую открывает динамическая трансляция адресов, является вытеснение сегментов. Если даже после перемещения сегментов запрос на память не может быть удовлетворен, то ОС может переписать какой-либо сегмент на внешнюю память и освободить занимаемую им реальную память. Поскольку контекст процесса, который содержится в вытесненном сегменте, сохраняется, то впоследствии ОС может вновь загрузить этот сегмент в реальную память, откорректировать базовый адрес в его дескрипторе и возобновить выполнение процесса. Перемещение сегментов и (см. ниже) страниц между оперативной и внешней памятью и наоборот – называется свопингом (swapping), а составные его части – вытеснением (swap out) и подкачкой (swap in). Поскольку в модели происходит вытеснение сегментов, должна быть реализована какая-то его стратегия. Естественно, что наилучшим кандидатом на вытеснение должен быть сегмент процесса, находящегося в заблокированном состоянии. Но следует при этом иметь в виду, что процесс может быть заблокирован потому, что он ожидает завершения операции ввода-вывода. При вводе-выводе, использующем канал или

прямой доступ к памяти, аппаратура ввода-вывода не выполняет трансляцию адресов (см. главу 6), а производит обмен данными с областью памяти, реальный адрес которой был задан ей при инициировании операции. Сегмент, участвующий в такой операции ввода-вывода, должен быть заблокирован не только от вытеснения, но и от перемещения в реальной памяти. (Эти же соображения должны учитываться и в других моделях памяти). При отсутствии подходящих кандидатов на вытеснение в очереди заблокированных процессов жертва может быть выбрана из очереди готовых процессов. Естественно назначить жертвой процесс, имеющий самый низкий приоритет у планировщика процессов. Но если брать этот приоритет единственным критерием, то имеется потенциальная опасность возникновения избыточных перемещений. Процесс, имеющий низкий приоритет, может быть несколько раз вытеснен и вновь подкачан, но между всеми подкачками и вытеснениями может так и не получить ни одного кванта обслуживания на центральном процессоре. В системах с динамическими приоритетами процесс (например, Unix), подкачанный в оперативную память, защищается от вытеснения некоторой временной выдержкой, в течение которой он имеет шанс повысить свой приоритет и получить квант обслуживания. Вытесненный процесс должен быть также некоторое время выдержан на внешней памяти, прежде чем он может быть подкачан. В системах со статическими приоритетами приоритет процесса у планировщика определяет и его приоритет в очереди к ресурсу реальной памяти.

Процесс, работающий в односегментной модели памяти, имеет возможность динамически изменять размер своего виртуального адресного пространства. При выполнении такого запроса от процесса ОС просто изменяет поле длины в дескрипторе его сегмента, если в реальной памяти вслед за сегментом процесса имеется свободный участок достаточного

размера. Если же такого участка нет, ОС может переместить процесс или заблокировать его в ожидании освобождения ресурса.

### **3.4. Многосегментная модель**

Расширим модель, рассмотренную в предыдущем разделе, на случай  $N$  сегментов.

Виртуальное пространство процесса разбивается на сегменты, которые нумеруются от 0 до  $N-1$ . Виртуальный адрес, таким образом, состоит из двух частей: номера сегмента и смещения в сегменте. Эти части могут либо представляться по отдельности каждая, либо упаковываться в одно адресное слово, в котором определенное число старших разрядов будет интерпретироваться как номер сегмента, а оставшаяся часть – как смещение. В первом случае сегменты могут размещаться произвольным образом в виртуальном адресном пространстве. Во втором случае создается впечатление плоского адресного пространства с адресами от 0 до максимально возможного при данной разрядности виртуального адреса, но в этом пространстве могут быть дыры – виртуальные адреса, для процесса недоступные: из-за отсутствия соответствующих сегментов или из-за сегментов, длина которых меньше максимально возможной.

Количество сегментов и максимальный размер сегмента ограничивается аппаратурой – разрядностью полей адресного слова. При выделении процессу реальной памяти каждый сегмент размещается в непрерывной области реальной памяти, но сегменты, смежные в виртуальной памяти, могут попадать в несмежные области памяти реальной. Теперь для процесса уже недостаточно одного дескриптора сегмента: он должен иметь таблицу таких дескрипторов в составе своего блока контекста. Аппаратный регистр дескриптора сегмента превращается в регистр адреса таблицы дескрипторов, он хранит указатель на таблицу

дескрипторов активного процесса и перезагружается при смене активного процесса. Вычисление реального адреса аппаратурой несколько усложняется, как показано на рисунке 3.5:

- выбирается сегментная часть виртуального адреса, она служит индексом в таблице дескрипторов; по индексу выбирается запись той таблицы, адрес которой находится в регистре адреса таблицы дескрипторов;
- выбранная запись является дескриптором сегмента, часть виртуального адреса, соответствующая смещению, сравнивается с полем длины в дескрипторе;
- если смещение в сегменте не превышает его длины, вычисляется реальный адрес как сумма базового адреса из дескриптора сегмента и смещения из виртуального адреса.

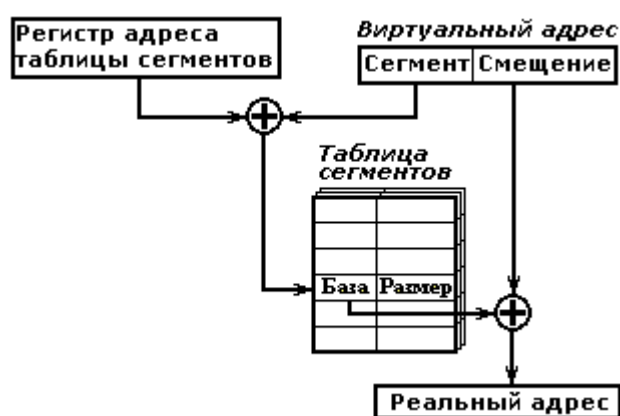


Рисунок 3.5 Трансляция адресов. Многосегментная модель

Примерная структура аппаратно поддерживаемого дескриптора сегмента приведена на рисунке 3.6. Подчеркиваем, что данная структура не является обязательной, для всех компьютерных архитектур, мы привели лишь наиболее распространенный ее вариант и использовали наиболее часто употребляемые имена полей.

имя поля	тип	назначение
base	реальный адрес	нач. адрес сегмента
size	целое число	размер сегмента
access	3 бита	доступ
present	1 бит	присутствие
used	1 бит	использование
dirty	1 бит	изменение

Рисунок 3.6. Примерная структура дескриптора сегмента

Допустимое количество сегментов определяется разрядностью соответствующего поля виртуального адреса и может быть весьма большим. Либо аппаратура должна иметь специальный регистр размера таблицы дескрипторов (такой регистр есть в Intel-Pentium), либо ОС должна подготавливать для процесса таблицу максимально возможного размера, отмечая в ней дескрипторы несуществующих сегментов (например, нулевым значением поля `size`). Отметим, что для систем, упаковывающих номер сегмента и смещение в одно адресное число, разрядность смещения не является ограничением на длину виртуального сегмента. Виртуальный сегмент большего размера представляется в таблице двумя и более обязательно смежными дескрипторами. С точки зрения процесса он обращается к одному сегменту, задавая в нем большое смещение, на самом же деле переполнение поля смещения переносится в поле номера сегмента. Если же простая двоичная арифметика не обеспечивает модификацию номера сегмента, возможность работы с большими сегментами может поддерживаться ОС путем особой обработки прерывания-ловушки "защита памяти".

Каковы преимущества многосегментной модели памяти?

Самое первое преимущество заключается в том, что у процесса появляется возможность разместить данные, обрабатываемые различным образом, в разных сегментах своего виртуального пространства (так, в ОС

Unix, например, каждый процесс имеет при начале выполнения три сегмента: кодов, данных и стека). Каждому сегменту могут быть определены свои права доступа. Поскольку обращения к памяти могут быть трех видов: чтение, запись и передача управления, то для описания прав доступа достаточно 3-битного поля Read-Write-eXecute, каждый разряд которого определяет разрешение одного из видов доступа. Аппаратные средства большинства архитектур обеспечивают контроль права доступа при трансляции адресов: поле прав доступа включается в дескриптор сегмента и если поступивший вид обращения не разрешен, то выполняется прерывание-ловушка "нарушение доступа".

Другое важное преимущество многосегментной модели заключается в том, что процесс имеет возможность использовать виртуальное адресное пространство, размер которого больше, чем размер доступной реальной памяти. Это достигается за счет того, что не обязательно все сегменты процесса должны одновременно находиться в реальной памяти. Дескриптор каждого сегмента содержит бит `present`, который установлен в 1, если сегмент подкачан в оперативную память, или в 0, – если сегмент вытеснен из нее. Аппаратура трансляции адресов проверяет этот бит и при нулевом его значении выполняет прерывание-ловушку "отсутствие сегмента" (`segment failure`). В отличие от большинства других ловушек, которые в основном сигнализируют об ошибках, при которых дальнейшее выполнение процесса невозможно, эта не приводит к фатальным для процесса последствиям. ОС, обрабатывая это прерывание, находит образ вытесненного сегмента на внешней памяти и подкачивает его в реальную память. Естественно, что процесс, обратившийся к вытесненному сегменту, переводится в ожидание; это ожидание может затянуться, если у ОС имеются проблемы с ресурсом реальной памяти. Когда сегмент будет подкачан, процесс перейдет в очередь готовых и будет активизирован вновь с той команды, которая вызвала прерывание-

ловушку. В тех аппаратных системах, которые не обрабатывают бит присутствия в дескрипторе сегмента, можно вместо него использовать поле `size`: ОС должна сбрасывать это поле в 0 при вытеснении сегмента и восстанавливать при его подкачке.

Естественно, что для подкачки сегмента ОС должна знать его адрес на внешней памяти. Подавляющее большинство систем не поддерживают аппаратно поле внешнего адреса в дескрипторе сегмента. Для хранения его ОС может либо использовать поле `base`, либо хранить этот адрес в расширении таблицы дескрипторов, в котором для каждого сегмента хранится информация о нем, обрабатываемая только программно. В том же расширении может храниться истинное значение поля `size` при использовании его вместо поля `present`.

Очевидно, что в многосегментной модели свопинг приобретает более интенсивный характер. Следовательно, эффективность стратегий управления памятью имеет еще больший вес. Поскольку в реальной памяти размещаются сегменты разной длины, проблемы размещения остаются те же. Для многосегментной организации характерны внешние дыры, борьба с которыми ведется перемещением сегментов. Если перемещение сегментов не освобождает такого количества реальной памяти, которое требуется для удовлетворения запроса, ОС может принять решение о вытеснении какого-либо сегмента и освобождении занимаемой им реальной памяти. Некоторые системы (Unix) вытесняют из памяти какой-либо процесс целиком со всеми принадлежащими ему сегментами, но такое решение может быть эффективным только в том случае, если процессы имеют по незначительному числу сегментов каждый. Стратегии вытеснения мы подробно рассмотрим позже – в страничной модели памяти.

Отметим, что вытеснение сегмента – не обязательно долгий процесс. Если сегмент не имеет прав доступа для записи, то он содержит команды



или статические инициализированные данные, копия этого сегмента уже есть на внешней памяти – в файле, содержащем загрузочный образ процесса. Поэтому вытеснение такого сегмента не сопровождается перезаписью его на внешнюю память. Даже если в сегмент и разрешена запись, то, возможно, запись еще не производилась, такой сегмент тоже незачем перезаписывать на внешнюю память. Поля `used` и `dirty` в дескрипторе сегмента используются при принятии стратегических решений о вытеснении.

Вытеснение одного или нескольких сегментов процесса еще не приводит к его блокированию. Процесс может продолжать выполняться до тех пор, пока не обратится к одному из вытесненных сегментов.

В многосегментной модели значительно увеличивается объем управляющей информации, сохраняемой в блоке контекста процесса, даже когда процесс не выполняется. Реальная память, занимаемая блоками контекста неактивных и даже полностью вытесненных процессов, не может быть отдана другим процессам. Такие участки памяти носят название заголовочных дыр. Если потери реальной памяти на заголовочные дыры оказываются недопустимо большими, то имеет смысл разбить блок контекста процесса на две части: меньшую, обязательно сохраняемую в реальной памяти, и большую, которая может быть вытеснена. В литературе [29, 41] описана ОС MULTICS, в которой для номера сегмента отводится 18 двоичных разрядов. Таблица сегментов процесса может быть настолько большой, что и одна она не поместится в оперативной памяти. Для преодоления этого противоречия таблица сегментов разбивается на страницы, которые подвергались свопингу. (ОС MULTICS была признана неудачным проектом и не получила широкого распространения, но значительно повлияла на последующие проекты ОС, прежде всего, – на Unix. Небольшое число инсталляций MULTICS эксплуатируется еще и сейчас.)

Чрезвычайно важным преимуществом многосегментной модели является возможность разделения сегментов процессами. Процессы могут быть разработаны так, чтобы виртуальные пространства двух или более процессов перекрывались в каких-то областях. Процессы могут использовать общее виртуальное пространство для обмена данными. Реализация этой возможности в какой-то мере зависит от аппаратных средств. Во многих вычислительных системах процесс может работать более чем с одной таблицей дескрипторов, поскольку в системе имеется несколько регистров адресов таблиц (так, в процессоре Intel 80286 и последующих предусмотрены две таблицы, называемые локальной и глобальной). Решение, использующее это свойство, заключается в том, что дескриптор разделяемого сегмента помещается в общую для всех процессов (глобальную) таблицу, такой сегмент может быть доступным для всех процессов и имеет общий виртуальный номер для всех процессов. Другое решение, возможное и при наличии только одной таблицы для каждого процесса, заключается в том, что для общего сегмента создается по записи в таблице каждого процесса, с ним работающего, для каждого процесса этот сегмент имеет свой виртуальный номер. Второе решение представляется более удачным с точки зрения защиты, так как, во-первых, доступ к сегменту имеют только те процессы, в таблицах которых созданы соответствующие дескрипторы, во-вторых, есть возможность дать разным процессам разные права доступа к разделяемому сегменту. Но за такое решение приходится платить тем, что при свопинге разделяемого сегмента и при учете его использования необходимо корректировать его дескрипторы в таблицах всех процессов.

Что касается стратегии подкачки, то все ОС применяют в сегментной модели "ленивую" политику: вытесненный сегмент подкачивается в реальную память только при обращении к нему. Некоторые системы (например, OS/2) позволяют управлять начальной подкачкой сегментов

при запуске процесса: сегмент может быть определен программистом как предварительно загружаемый (preload) или загружаемый по вызову (load on call). Разработать неубыточную стратегию упреждающей (до обращения) подкачки сегментов при свопинге пока не представляется возможным из-за отсутствия надежных методов предсказания того, к какому сегменту будет обращение в ближайшем будущем.

В многосегментной модели процесс имеет возможность динамически изменять структуру своего виртуального адресного пространства. Для этих целей ему должен быть доступен минимальный набор системных вызовов, приводимый ниже.

Получить сегмент:

```
seg = getSeg (segsizе, access);
```

ОС выделяет новый сегмент заданного размера и с заданными правами доступа и возвращает процессу виртуальный номер выделенного сегмента.

Освободить сегмент:

```
freeSeg(seg);
```

сегмент с заданным виртуальным номером становится недоступным для процесса.

Изменить размер сегмента:

```
chSegSize(seg, newsizе).
```

Изменить права доступа к сегменту:

```
chSegAccess(seg, newaccess).
```

В конкретных системах этот минимальный набор может быть расширен дополнительным системным сервисом.

Системные вызовы, связанные с разделяемыми сегментами, мы рассмотрим в главе, посвященной взаимодействию между процессами.

### 3.5. Страничная модель

Страничную организацию памяти легко представить как многосегментную модель с фиксированным размером сегмента. Такие сегменты называются страницами. Вся доступная реальная память разбивается на страничные кадры (page frame), причем границы кадров в реальной памяти фиксированы. Иными словами, реальная память представляется как массив страничных кадров.

Виртуальный адрес состоит из номера страницы и смещения в странице, система поддерживает таблицу дескрипторов страниц для каждого процесса. Дескриптор страницы в основном подобен дескриптору сегмента, но в нем может быть сокращена разрядность поля `base`, так как в нем хранится не полный реальный адрес, а только номер страничного кадра, а необходимость в поле `size` вообще отпадает, так как размер страниц фиксирован.

Проблема размещения значительно упрощается, так как любой страничный кадр подходит для размещения любой страницы, необходимо только вести учет свободных кадров. За счет этого страничная организация оказывается удобной даже при отсутствии свопинга, так как позволяет разместить непрерывное виртуальное адресное пространство в несмежных страничных кадрах. (Иногда для обозначения свопинга на уровне страниц применяют специальный термин `paging` – страничный обмен.) Внешние дыры в страничной модели отсутствуют, зато появляются внутренние дыры за счет недоиспользованных страниц. При наличии в системе свопинга нулевое значение бита `present` вызывает прерывание-ловушку "страничный отказ" (`page failure`) и подкачку страницы в реальную память. Для учета занятых/свободных страниц подходит техника битовой карты,

но большинство ОС используют в качестве элементов карты (таблицы страничных кадров) не биты, а куда более сложные структуры, из которых могут составляться и многосвязные списки (в том числе и списки свободных кадров).

Какой размер страницы выгоднее – большой или малый? Соображения, которые могут повлиять на выбор размера, следующие:

- при малых страницах получаются меньшие внутренние дыры;
- при малых страницах меньше вероятность страничного отказа (так как больше страниц помещается в памяти);
- при больших страницах меньшие аппаратные затраты (так как разбиение памяти на большие блоки обойдется дешевле);
- при больших страницах меньшие заголовочные дыры и затраты на поиск и управление страницами (таблицы имеют меньший размер);
- при больших страницах выше эффективность обмена с внешней памятью.

Интересно, что разные авторы [12] и [36], приводя почти идентичные соображения по этому поводу, приходят к диаметрально противоположным выводам – о преимуществе малых [12] или больших [36] страниц. Вообще можно проследить по большинству источников, что рекомендуемые размеры страниц растут с возрастанием года издания. По-видимому, решающим фактором здесь является вопрос стоимости памяти. Со временем стоимость этого ресурса уменьшается, а доступные объемы увеличиваются, поэтому более поздние авторы придают меньше значения потерям памяти.

Поскольку размер страницы обычно выбирается намного меньше, чем размер сегмента в предыдущей модели, страничная организация позволяет значительно увеличить уровень мультипрограммирования за счет того, что в реальную память могут в каждый момент отображаться

только самые необходимые части виртуальных пространств процессов. Но при этом еще более возрастает интенсивность свопинга и соответственно роль стратегии вытеснения/подкачки страниц. При проектировании систем со страничным свопингом разработчики должны придерживаться двух основополагающих правил:

- процесс при большинстве своих обращений к памяти должен находить требуемую страницу уже в реальной памяти;
- если потери на свопинг превышают допустимую норму, то должен понижаться уровень мультипрограммирования.

Несоблюдение этих правил делает весьма вероятным возникновение такой ситуации, когда система будет занята только хаотичным перемещением страниц. Например, выполнение команды процессора S/390:

MVC память, память

может потребовать трех страниц памяти: команды, первого и второго операндов. Если одной из страниц недостает, процесс блокируется в ожидании ее подкачки. Но при неудачной стратегии за время этого ожидания он может потерять другие необходимые страницы, и повторная попытка выполнить команду вновь приведет к прерыванию-ловушке. В англоязычной литературе эту ситуацию называют *trash* – толкотня, в русскоязычной часто используется транскрипция – треш.

При оценке эффективности стратегии свопинга показательной является зависимость частоты страничных отказов от числа доступных страничных кадров. Качественный вид этой зависимости, присущий всем стратегиям, показан на рисунке 3.7. Из него видно, что при уменьшении объема реальной памяти ниже некоторого ограниченного значения число страничных отказов начинает расти экспоненциально. Естественно, что показателем эффективности стратегии может служить степень близости колена этой кривой к оси ординат. Другой возможный критерий

эффективности – площадь области, расположенной под этой кривой; чем она меньше, тем выше эффективность.



Рисунок 3.7 Зависимость частоты отказов от объема реальной памяти

С точки зрения стратегии размещения не имеет значения, какой страничный кадр занимать при подкачке. Но как выбрать один кадр из всего пула кадров? Конечно, наилучшим кандидатом является полностью свободный кадр – такой, который еще не занимался или был когда-то выделен процессу, ныне уже закончившемуся. Но если суммарный объем виртуальных адресных пространств всех процессов существенно превышает объем реальной памяти, то такие страницы являются более чем дефицитным ресурсом. Выход состоит в том, что для каждого распределенного кадра в таблице страничных кадров (и/или в таблице дескрипторов) ведется учет факторов, определяющих выбор его в качестве жертвы для вытеснения, и из всех кадров выбирается тот, который является лучшим (с точки зрения принятой стратегии) кандидатом в жертвы (OS/2, Windows 95). Несколько более сложное решение: ОС ведет список свободных (условно свободных) кадров, и очередная жертва выбирается из этого списка. Страничный кадр попадает в список жертв,

если его "показатель жертвенности" превышает некоторое граничное значение, но может быть еще "спасен" из списка жертв, если во время пребывания в этом списке он будет востребован. Помещение кадра в список жертв может выполняться либо сразу по достижении "показателя жертвенности" граничного значения (VM), либо (ленивая политика – Unix) размер списка поддерживается в определенных границах за счет периодического его пополнения путем просмотра всей таблицы страничных кадров.

В качестве образцов для сравнения в литературе рассматриваются стратегии вытеснения RANDOM и OPT. Стратегия RANDOM заключается в том, что страница для вытеснения выбирается случайным образом. Понятно, что достичь высокой эффективности при такой стратегии невозможно, и любая другая введенная нами стратегия может считаться сколько-нибудь разумной только в том случае, если она, по крайней мере, не хуже стратегии RANDOM. Стратегия OPT требует, чтобы в первую очередь вытеснялась страница, к которой дольше всего не будет обращений в будущем. Интуитивно понятно и строго доказано, что эта стратегия является наилучшей из всех возможных. Но, к сожалению, эта стратегия в идеальном варианте нереализуема из-за невозможности точно прогнозировать требования. Реально применяемые стратегии могут оцениваться по степени приближения их результатов к результатам OPT.

Стратегия FIFO (первый на входе – первый на выходе) является простейшей. Согласно этой стратегии из реальной памяти вытесняется та страница, которая была раньше других в нее подкачана. Для реализации этой стратегии ОС достаточно организовать список-очередь страниц в реальной памяти с занесением подкачиваемой страницы в "голову" списка и выборкой страницы для вытеснения из "хвоста" списка. Хотя стратегия FIFO и лучше, чем RANDOM, она не учитывает частоты обращений: может быть вытеснена страница, к которой происходят постоянные



обращения. Более того, при некоторых комбинациях страничных требований FIFO может давать аномальные результаты: увеличение числа страничных отказов при увеличении числа доступных страничных кадров (см. задания в конце главы).

Многие используемые в современных ОС стратегии вытеснения могут рассматриваться как разновидности стратегии LRU (least recently used) – наименее используемая в настоящее время: вытесняется та страница, к которой дольше всего не было обращений. Это можно рассматривать как попытку приближения стратегии OPT путем экстраполяции потока страничных требований из прошлого на будущее. Разновидности LRU различаются тем, как они учитывают время использования страницы. Очевидно, что запоминание точного времени обращения к каждой странице обошлось бы слишком дорого. Стратегии LRU используют биты `used` и `dirty` дескриптора страницы для оценки этого времени. Бит `used` устанавливается в 1 аппаратно при любом обращении к странице, бит `dirty` устанавливается аппаратно при обращении к странице для записи; оба бита сбрасываются ОС по своему усмотрению. Все множество присутствующих в реальной памяти страниц разбивается на четыре подмножества, в зависимости от значений этих полей:

- 1) неиспользованные чистые (`used=0`, `dirty=0`);
- 2) неиспользованные грязные (`used=0`, `dirty=1`);
- 3) использованные чистые (`used=1`, `dirty=0`);
- 4) использованные грязные (`used=1`, `dirty=1`).

Чем меньше номер подмножества, в которое входит страница, тем желательнее она в роли жертвы. Внутри одного подмножества жертва может выбираться методом циклического поиска или случайным образом. ОС должна выбрать момент, когда она будет сбрасывать биты `used` в 0.

Ленивая политика состоит в том, чтобы делать это только, когда уже не остается неиспользованных страниц. Противоположные варианты – при каждом поиске жертвы сбрасывать биты `used` для всех страниц или только для проверенных в ходе поиска. Наконец, общий сброс битов `used` может производиться по таймеру.

Интересным образом используется бит `dirty` в стратегии SCC (`second cycle chance`) – цикл второго шанса. Алгоритм этого варианта стратегии LRU осуществляет циклический просмотр таблицы страничных кадров. Разумеется, лучшим кандидатом является неиспользованная страница (подмножества 1 и 2). Но если таковых нет, то выбирается страница с нулевым полем `dirty` (подмножество 3). Просмотренные страницы, оказавшиеся грязными, ОС не трогает (пока), но сбрасывает в них поле `dirty`. Таким образом, даже если при полном обороте поиска не будет найдена жертва, она обязательно найдется при втором обороте. "Второй шанс" здесь состоит в том, что страница, принудительно отмеченная как чистая, может еще в промежутке между двумя поисками восстановить поле `dirty`. При такой стратегии поле `dirty` уже не отражает истинного состояния страницы, ОС должна сохранять действительное состояние в расширении дескриптора страницы.

Более сложные варианты стратегии LRU предусматривают более чем одноразрядный учет времени. Метод временного штампа (`timestamp`) предусматривает хранение для каждой страницы многоразрядного кода. Периодически этот код сдвигается на разряд влево, а в освободившийся старший разряд заносится текущее значение поля `used`, после чего поле `used` сбрасывается в 0. Код временного штампа хранит, таким образом, предысторию использования страниц. Наилучшей жертвой оказывается та страница, у которой значение штампа (если интерпретировать его как целое число без знака) минимальное.

Метод возраста страницы (Unix) подобен предыдущему, но здесь с каждой страницей связывается число – ее "возраст". При периодическом просмотре таблицы страничных кадров для страницы, у которой бит `used` равен 0, возраст увеличивается на 1, если же бит `used` установлен в 1, то он сбрасывается в 0, а возраст не меняется. Когда системный процесс "сборщик страниц" пополняет список свободных страниц, он заносит в него те страницы, для которых превышен установленный граничный возраст.

Выше мы использовали биты `used` и `dirty`, предполагая, что они поддерживаются аппаратно. Однако возможна реализация указанных стратегий и при отсутствии такой аппаратной поддержки. В этом случае недостающие поля содержатся в расширении дескриптора страницы, а вместо них ОС сбрасывает в 0 бит `present`. Бит `present` в дескрипторе уже не отражает истинного состояния и дублируется в расширении. Обращение к такой странице вызывает ловушку "страничный отказ", однако ОС убеждается по расширению дескриптора, что страница на самом деле уже есть в реальной памяти, и вместо ее подкачки корректирует бит `present` в основном дескрипторе, одновременно устанавливая `used` в расширении.

Указанные стратегии применимы как к локальному, так и к глобальному управлению памятью. В первом случае каждому процессу выделяется статический пул страничных кадров, и свопинг ведется в пределах этого пула (например, AS/400). Это дает возможность применять для разнотипных процессов разные стратегии, но требует принятия решения о размере каждого пула. В случае же глобального управления выбранная стратегия применяется по всему доступному множеству страничных кадров и производится постоянное перераспределение ресурсов между процессами (например, VM, MVS). Динамическое

перераспределение – качество полезное, но если оно не учитывает уровень мультипрограммирования, то может привести к толкотне (трешу).

Ряд глобальных стратегий, управляющих уровнем мультипрограммирования, основывается на идее рабочего набора WS (working set). Идея эта базируется на явлении локализации обращений к памяти. Любая программа не обращается ко всему своему адресному пространству одновременно. На каждом временном отрезке программа работает только с некоторым подмножеством адресов и соответственно с некоторым подмножеством страниц. Временной отрезок, на котором это подмножество квазипостоянно, называется фазой программы. Почти полное обновление этого подмножества называется сменой фазы. Рабочим набором процесса  $S(w)$  называется перечень страниц, к которым процесс обращался в течение последнего интервала виртуального времени  $w$ . Методы, основанные на идее рабочего набора, стремятся к тому, чтобы выполняющийся процесс постоянно имел свой рабочий набор в реальной памяти и страницы, входящие в рабочие наборы выполняющихся процессов, не вытеснялись. Если у ОС нет возможности разместить в реальной памяти весь рабочий набор процесса, она снижает уровень мультипрограммирования, переводя процесс в состояние ожидания. При разблокировании процесса ОС имеет возможность перед его активизацией выполнить упреждающую подкачку (preload) всего его рабочего набора и тем самым значительно снизить частоту страничных отказов.

На практике, однако, идеальная реализация стратегии WS не представляется возможной по тем же соображениям, что и для идеальной LRU: нет возможности запоминать время каждого обращения. Адаптированный метод WS состоит в его определении через фиксированные интервалы времени. В начале каждого интервала все страницы, для которых распределены страничные кадры, считаются входящими в рабочий набор. В течение интервала все запросы процесса на

новые страницы удовлетворяются. По окончании интервала те страницы, которые не были использованы (бит *used*), удаляются из рабочего набора. Зафиксированный в конце интервала рабочий набор служит исходным для следующего интервала.

Некоторые методы стратегии WS не сохраняют полный перечень страниц, входящих в рабочий набор, а только определяют его обобщенные показатели. Алгоритм метода рабочего размера измеряет только размер рабочего набора и выделяет процессу соответствующее число страничных кадров (VM). Метод частоты страничных отказов основан на измерении интервала виртуального времени между двумя страничными отказами, если этот интервал меньше нижней границы, процессу выделяется дополнительный страничный кадр, если интервал больше верхней границы, у процесса отбирается один страничный кадр. Естественно, что методы, не сохраняющие всего списка рабочего набора, не имеют возможности выполнять упреждающую его подкачку.

При страничном свопинге, как и при сегментном, применяется, как правило, стратегия подкачки по запросу (*demand paging*), так как реализовать полностью безубыточную стратегию упреждающей подкачки невозможно. Тем не менее, в стратегии WS появляется возможность упреждающей подкачки с минимальными убытками. В системах, имеющих большой объем памяти и не особенно заботящихся о минимизации ее потерь, иногда применяется также кластеризация подкачки. Этот метод также базируется на локализации и исходит из того, что если произошло обращение к некоторой странице, то с большой вероятностью можно ожидать в ближайшем будущем обращений к соседним с ней страницам, которые и подкачиваются, не дожидаясь этих обращений.

Системные вызовы страничной модели "не видят" страничной структуры и обращаются к памяти как к линейному пространству виртуальных адресов.

Выделение памяти происходит при помощи системного вызова:

```
vaddr = getMem(size),
```

который возвращает виртуальный адрес выделенной области заданного размера. На самом деле размер выделенной области кратен размеру страницы, а ее адрес выровнен по границе страницы.

Освобождение памяти:

```
freeMem(vaddr)
```

Поскольку память выделяется страницами, при выделении памяти для маленьких (существенно меньше размера страницы) объектов образуются значительные внутренние дыры. Для того, чтобы избежать этих потерь, некоторые системы обеспечивают работу с кучей (heap) – заранее выделенной областью памяти размером в одну или несколько страниц с последующим выделением памяти для маленьких объектов в куче.

Соответствующие системные вызовы:

```
heapId = createHeap(size);
```

```
vaddr = getHeapMem(heapID,size);
```

```
freeHeapMem(vaddr)
```

### **3.6. Сегментно-страничная модель**

Из предыдущего изложения должно быть ясно, что сегментная модель памяти ориентирована в большей степени на программиста, а страничная – на ОС. Программисту удобно компоновать команды и данные своего процесса в блоки переменной длины (сегменты). ОС же удобнее управлять памятью, разбитой на блоки постоянной длины (страницы). Естественным путем развития моделей памяти явилось объединение достоинств этих двух моделей в одной – сегментно-страничной.

Виртуальный адрес теперь состоит из трех частей: номера сегмента, номера страницы в сегменте и смещения в странице. Аппарат трансляции адресов, представленный на рисунке 3.8, по крайней мере трехшаговый:

- регистр адреса дескриптора указывает на таблицу сегментов, из нее выбирается дескриптор сегмента, а из последнего – адрес таблицы страниц;
- из таблицы страниц выбирается дескриптор страницы, а из него номер страничного кадра;
- реальный адрес получается сложением базового адреса страничного кадра со смещением в странице.

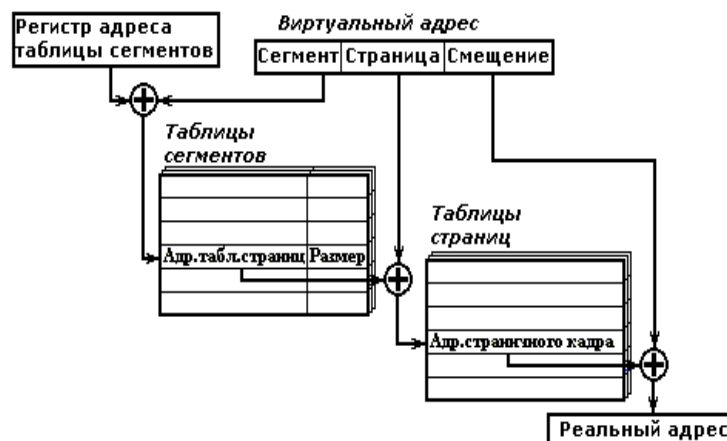


Рисунок 3.8 Трансляция адресов. Сегментно-страничная модель

Такой аппарат трансляции адресов поддерживается во многих современных процессорных архитектурах. Иногда алгоритм вычисления адреса состоит и из большего числа шагов. Серьезным недостатком этой модели является многоступенчатость трансляции адресов. Эта проблема решается на аппаратном уровне путем применения сверхбыстродействующей (обычно ассоциативной) памяти для хранения части таблиц.

Поскольку в модели, приведенной на рисунке 3.8, каждый сегмент имеет собственную таблицу страниц, сами таблицы страниц могут занимать значительный объем в памяти. Простейшее решение этой

проблемы представляет Windows 3.x: в системе существует единственная таблица страниц. Сегментная часть трансляции адреса имеет, таким образом, на выходе адрес в общем для всех процессов виртуальном страничном пространстве, объем которого превышает объем реальной памяти не более, чем в 4 раза. Подобное же, хотя и более гибкое и защищенное решение, представляет VSE: система обеспечивает общий объем виртуальной памяти (до 2 Гбайт), который разбивается на разделы (до 12 статических и до 200 динамических), суммарный объем адресных пространств всех разделов не превышает общего объема виртуальной памяти. Простота решения, однако, может существенно сказываться на его эффективности: во-первых, из-за ограничений на размер виртуальной памяти, во-вторых, из-за необходимости выделять смежные дескрипторы в таблице страниц для страниц, смежных в виртуальной памяти. Поэтому действительно многозадачные системы применяют множественные таблицы страниц и включают память, занимаемую таблицами страниц, в страничный обмен. Вытеснение и загрузка частных таблиц страниц производится либо исключительно самой ОС (Unix), либо ОС использует для этого имеющиеся аппаратные средства (так, ОС, ориентированные на Intel-Pentium, используют каталоги страниц).

С точки зрения программиста, он имеет дело с сегментами, API управления памятью – такой же, как и в сегментной модели. Страничная часть адресации для процессов совершенно прозрачна, но зато эта часть используется ОС для эффективной организации свопинга.

### **3.7. Плоская модель**

Начиная с модели Intel 80386, в микропроцессорах Intel-Pentium адрес состоит из 16-разрядного номера сегмента и 32-разрядного смещения. 32-разрядное поле смещения позволяет адресовать до 4 Гбайт в



пределах одного сегмента, что более чем достаточно для большинства мыслимых приложений и позволяет реализовать действительно "плоскую" (flat) модель виртуальной памяти процесса, представляющую собой линейное непрерывное пространство адресов..

Однако при размере страницы 4 Кбайт таблица страниц должна содержать более  $10^6$  элементов и занимать 4 Мбайт памяти. Для экономии памяти аппаратура трансляции адреса микропроцессора поддерживает таблицы страниц двух уровней. Страничная таблица верхнего уровня называется каталогом страниц. Старшие 10 байт 32-разрядного смещения являются номером элемента в страничном каталоге. Элемент страничного каталога адресует таблицу страниц второго уровня. Следующие 10 байт смещения являются номером элемента в таблице страниц второго уровня. Элемент таблицы второго уровня адресует страничный кадр в реальной памяти, а младшие 12 байт смещения являются смещением в странице. Сегментная часть аппарата трансляции адреса оказывается излишней, но в Intel-Pentium она не может быть отключена, но тот же эффект достигается, если каждому процессу назначается только один сегмент и для процесса создается таблица сегментов, содержащая только один элемент. Поле base этого элемента адресует страничный каталог процесса, а каждый элемент страничного каталога – одну таблицу страниц. Структуры элементов каталога страниц и таблицы страниц второго уровня идентичны, каждая таблица страниц (каталог или таблица второго уровня) содержит 1024 элемента и сама занимает страничный кадр в памяти. Таблицы страниц участвуют в страничном обмене так же, как и страницы, содержащие любые другие данные и коды.

В 4-Гбайтном адресном пространстве появляется возможность разместить не только коды и данные процесса, но и объекты, используемые им совместно с другими процессами, в том числе и модули самой ОС. В этом случае обращение процесса к ОС происходит как

обращение к процедуре, размещенной в адресном пространстве самого процесса. В современных ОС структура адресного пространства процесса обычно бывает следующей:

- самая младшая часть адресного пространства обычно для процесса недоступна, она используется ОС для поддержки реального режима; размер этой части адресного пространства обычно не менее 4 Мбайт, что соответствует одному элементу страничного каталога;
- далее размещается частное адресное пространство процесса, содержащее его коды, локальные данные, стек;
- выше размещаются "прикладные" общие области памяти, используемые несколькими процессами совместно;
- еще выше – системные модули, работающие в непривилегированном режиме, эти модули совместно используются всеми процессами;
- наконец, в самой верхней части размещаются системные модули, работающие в режиме ядра (уровень привилегий – 0), эти модули также используются совместно.

Совместное использование памяти обеспечивается либо тем, что элементы каталогов разных процессов адресуют одну и ту же таблицу страниц второго уровня, либо тем, что таблицы страниц второго уровня разных процессов адресуют один и тот же страничный кадр. В первом случае виртуальные адреса совместно используемых объектов являются одинаковыми для всех процессов, во втором – разными. Все системы используют первый способ для системных модулей, но разные способы – для "прикладных" общих областей памяти.

Большинство разработчиков приложений горячо приветствовали введение плоской модели памяти в современных ОС (OS/2 Warp, Windows 95, Windows NT), так как представление виртуального адреса в виде

одного 32-разрядного слова избавляет программиста от необходимости различать ближние и дальние указатели и упрощает программирование. Но справедливы и предупреждения о том, что отказ от сегментного структурирования виртуального адресного пространства кое в чем ограничивает возможности программиста [23]. Большая же эффективность плоской модели памяти является объективным фактором, так как, во-первых, оперирование с 32-разрядными адресными словами уменьшает число команд в программе, а во-вторых, в 4-Гбайтном виртуальном адресном пространстве процесса могут быть размещены и процедуры, реализующие системные вызовы, таким образом, обращения процесса к ОС происходят, как к собственным локальным процедурам и не требуют переключений контекста.

Несколько усложняется защита памяти при фактическом отказе от сегментирования. В Intel-Pentium в аппаратном дескрипторе сегмента предусмотрено пять двоичных разрядов, которые могут быть использованы для целей защиты, а в дескрипторе страницы – только два таких разряда. Однако объединение средств защиты на уровне каталога страниц и таблиц второго уровня образует достаточно богатые возможности. Надежность защиты памяти в современных ОС определяется только тем, насколько активно и аккуратно эти возможности используются.

### **3.8. Одноуровневая модель**

Дальнейшее расширение разрядной сетки процессоров может привести к появлению совершенно новых моделей памяти. Сейчас трудно с уверенностью прогнозировать, какая модель будет доминировать, на сегодняшний день большинство 64-разрядных ОС представляют собой клоны Unix, и расширенный виртуальный адрес используется в них для

отображения в память файлов. Более активно использует 64-разрядный адрес AS/400 [27]. На примере последней мы и рассмотрим одноуровневую (single-level) модель памяти. Эта модель была реализована уже в System/38 и в ранних моделях AS/400 на базе 48-разрядного адреса, но мы сосредоточимся на ее 64-разрядной реализации в Advanced Series на процессоре Power PC

Отметим, прежде всего, что эта "новая" модель памяти на самом деле является "хорошо забытой старой": в вычислительной системе Atlas (Англия, 1996 г.) – первой системе с виртуальной памятью – была реализована именно эта модель. Впоследствии эта модель не нашла применения из-за ограниченных возможностей аппаратных средств, современное же состояние аппаратных средств позволяет вновь к ней вернуться на качественно ином уровне.

64-разрядное адресное слово позволяет процессу иметь плоскую виртуальную память размером до 16 Эбайт (эксабайт). В AS/400 эта возможность позволяет реализовать два принципиально важных свойства модели памяти:

- в виртуальное адресное пространство процесса включается не только оперативная память (memory), но вся память (storage) – и оперативная, и внешняя – имеющаяся в системе;
- все процессы работают в одном и том же виртуальном адресном пространстве, разделяя его.

В традиционных системах любые объекты – обрабатываемые или выполняемые – должны быть прежде размещены в памяти. Это не обязательно означает их размещения в реальной оперативной памяти, такое размещение может быть отложено и выполняться по требованию механизмами подкачки сегментов или страниц, но виртуальная память процесса должна быть сформирована – в виде таблиц сегментов и/или страниц. В системе с одноуровневой памятью, строго говоря, концепция

памяти отсутствует, она заменена концепцией пространства (space). Новосозданный процесс сразу получает свое распоряжение пространство (виртуальное адресное пространство), в котором уже размещены все имеющиеся в системе объекты, в том числе и программные коды процесса. Имеется также достаточно пространства для размещения любых новых объектов. Для работы с объектом процесс должен не размещать его в памяти, а должен только получить его адрес в пространстве. Для процесса прозрачно местонахождение объекта – в оперативной или на внешней памяти. Физически все объекты размещаются именно на внешней памяти, а оперативная память (ее размер исчисляется сотнями Мбайт) используется почти исключительно как пул страничных кадров.

Одноуровневая модель делает излишним API, связанный с управлением памятью, поскольку процессам нет необходимости запрашивать и освобождать память. Системные вызовы этой группы заменяются вызовами доступа к объектам. Защита памяти, таким образом, реализуется в рамках более общего механизма контроля доступа, который мы подробнее рассмотрим в главе 10. Задача же совместного использования памяти решается совершенно элементарно, так как все процессы работают в одном виртуальном адресном пространстве.

Плоская структура адресного пространства в одноуровневой модели снимает необходимость в сегментной фазе динамической трансляции адреса. В процессоре Power PC имеется возможность программного отключения трансляции сегмента из аппаратного процесса трансляции адреса. AS/400 может работать как в режиме с сегментацией, так и без нее. Режим с сегментацией включается только при повышенном уровне защиты.

Размер страницы в современных моделях AS/400 – 4Кбайт. Даже в процессоре Intel 80386 таблица страниц для 32-разрядного адресного пространства не размещается в оперативной памяти и применяется

двухэтапная трансляция номера страницы, как же решается эта проблема для 64-разрядного адресного пространства? Здесь применяется так называемая инверсная таблица страниц. В оперативной памяти хранятся дескрипторы не всех виртуальных страниц, а только тех, которые уже размещены в оперативной памяти. Поскольку виртуальное адресное пространство общее для всех процессов, таблица страниц также одна. При трансляции виртуального адреса требуемая страница станчала ищется в таблице страниц реальной памяти, а при неуспешном результате такого поиска – на внешней памяти. Для поиска в таблице страниц применяется метод хеширования. 64-разрядный виртуальный адрес путем несложных преобразований, состоящих из ряда побитовых логических операций, преобразуется в номер элемента таблицы страниц. Число элементов в таблице страниц зависит от размера оперативной памяти в системе и выбирается таким образом, что таблица страниц занимает фиксированный процент реальной оперативной памяти. Поскольку разрядность номера элемента значительно меньше разрядности виртуального адреса, в процессе преобразования виртуальных адресов неизбежны коллизии – случаи преобразования разных виртуальных адресов в один и тот же номер элемента. Поэтому в элементе таблицы страниц зарезервировано место для нескольких (восьми) дескрипторов страниц и после выбора элемента продолжается линейный поиск страницы в элементе таблицы. В таблице страниц также предусмотрена область переполнения – для случая, если число коллизий на один элемент таблицы превысит размер элемента, но на практике до ее использования дело не доходит.

Механизм поиска в таблице страниц может показаться достаточно сложным и времяемким, но, во-первых, архитектура микропроцессора Power PC включает в себя конвейер, а во-вторых, значительный объем ассоциативного буфера страниц (512 и более элементов) позволяет более чем в 90% случаев даже не производить поиск в таблице страниц.

В случае, если страница не найдена в таблице, генерируется прерывание-ловушка – страничный отказ. Модуль управления памятью в микроядре при обработке этой ловушки рассматривает виртуальный адрес как адрес на внешней памяти и передает его подсистеме ввода-вывода для подкачки страницы в оперативную память. Подкачка может потребовать освобождения страничного кадра – применяется дисциплина замещения LRU в пределах страничного пула, выделенного данному процессу. В системе предусмотрена также возможность использования реальных адресов памяти – оперативной и внешней, но команды, работающие с ними, используются только ниже уровня интерфейса МІ и недоступны даже для OS/400.

## **КОНТРОЛЬНЫЕ ВОПРОСЫ**

1. Часто единственным достоинством виртуальной памяти называют возможность обеспечить для процесса объем виртуального адресного пространства, превышающий объем реальной памяти. Назовите другие достоинства виртуальной памяти.

2. В чем достоинства и недостатки преобразования виртуальных адресов в реальные во время выполнения программы? Какая часть работы по этому преобразованию выполняется аппаратным обеспечением, а какая – ОС?

3. Иногда считают, что виртуальная память может быть обеспечена только в системах с аппаратной поддержкой динамической трансляции адреса. Докажите, что это не так.

4. Почему при поиске свободной памяти стратегия "самый подходящий" оказывается хуже, чем "первый подходящий".

5. Сравните сегментную и страничную модели виртуальной памяти. Какая из них представляется вам лучшей и почему?

6. Дополните приведенные в разделе 3.5. соображения по поводу выбора размера страницы.

7. Смоделируйте ситуацию применения дисциплины вытеснения FCFS, в которой увеличение числа реальных страниц приведет к увеличению числа страничных отказов.

8. Что такое кластерная подкачка страниц? Почему в современных ОС она становится все более популярной?

9. Каким образом ОС может определять, к каким страницам будут обращения в ближайшее время?

10. Большой размер виртуальной памяти процесса может приводить к тому, что даже таблица страниц не будет помещаться в реальной памяти. Какими путями решается эта проблема в современных ОС?

11. Каким образом снижение стоимости памяти влияет на дисциплины управления памятью?

12. Какие принципиальные изменения в концепции памяти может повлечь за собой увеличение разрядности адреса?



## Глава 4. Порождение программ и процессов

В первой главе мы дали строгое определение понятия процесса. Прикладной программист, однако, разрабатывает не "процесс", а "программу", не задумываясь обычно над тем, как и какие механизмы ОС обеспечат ее представление в виде процесса. Ряд авторов (например, [12, 41]) нестрого определяют процесс как "программу в стадии выполнения". Такое определение, "адаптированное" для уровня прикладного программиста, в ряде случаев может считаться справедливым и весьма удобным, так как соответствует интуитивному пониманию этого термина.

Программа превращается в процесс в тот момент, когда ОС создает для нее блок контекста; последний отвечает за состояние процесса и представляет процесс в состязаниях за обладание ресурсами. Блок контекста, однако, не определяет содержание процесса. Содержательная часть процесса представляется для ОС другой структурой – адресным пространством процесса. Ядро ОС не обрабатывает содержимое адресного пространства, а только отвечает за размещение его в памяти.

В первых двух разделах данной главы мы рассматриваем начальное формирование содержимого адресного пространства: этапы компиляции, компоновки и загрузки. Эти этапы в терминах функций управления памятью (рисунок 3.1) выполняют функцию именования. Как правило, эти этапы выполняются не ядром ОС, а утилитами, задолго до создания блока контекста процесса. Таким образом, на этих этапах мы имеем дело не с процессом, а с программой. Два следующих раздела рассматривают управление адресным пространством программы при ее превращении в процесс.

## 4.1. Компиляция

Этот этап реализуется не ОС, а системами программирования, которые представляют собой "системы, образуемые языком программирования, компиляторами или интерпретаторами программ, представленных на этом языке, соответствующей документацией, а также вспомогательными средствами для подготовки программ в форме, пригодной для выполнения" [8]. Системы программирования – предмет изучения отдельного курса, им посвящена обширная литература (например, [3, 7, 24]), здесь мы остановимся только на тех их аспектах, которые имеют отношение к ОС и аппаратным средствам вычислительной системы.

Основным функциональным назначением системы программирования является генерация объектного кода программы (машинных команд). Компиляторы предварительно формируют структуру виртуального адресного пространства: определяют состав сегментов и формируют содержимое (образы) кодовых сегментов и сегментов инициализированных статических данных.

Система программирования создает также дополнительный интерфейс между программистом и ОС. Состав и спецификации этого интерфейса могут быть либо стандартными для языка, либо определяться конкретной системой программирования. Везде в этом пособии мы описываем системные вызовы ОС как некоторые процедуры или функции высокоуровневого языка программирования. В конкретных системах программирования набор таких процедур составляет библиотеки системных вызовов, эти процедуры обеспечивают передачу вызовов ОС в той форме, в какой они специфицированы для данной ОС (например, в виде программного прерывания с передачей параметров через регистры общего назначения). Многие процедуры систем программирования

включают в себя интегрированный системный сервис – выполнение в составе одной процедуры нескольких системных вызовов с некоторой обработкой их результатов. Можно говорить о том, что системы программирования продолжают тенденцию виртуализации ресурсов: они формируют на базе примитивов, обеспечиваемых системными вызовами ОС, ресурсы более высокого уровня, доступные через средства системы программирования. Так, работая на языке высокого уровня, мы имеем в своем распоряжении виртуальную ЭВМ, в которой "система команд" представлена операциями, операторами и стандартными процедурами языка, а адресация выполняется в пространстве символьных имен. Некоторые языки или их конкретные системы программирования могут включать в себя и более сложные средства управления ресурсами, такие как: буферизацию ввода/вывода (см. главу 6), работу с файлами сложной логической структуры (см. главу 7), средства синхронизации и взаимодействия процессов (см. главу 8) и т.д.

С целью получения наиболее эффективного объектного кода компиляторы могут выполнять оптимизацию обрабатываемой программы. Можно выделить три стадии такой оптимизации:

- системно-независимая;
- системно-зависимая, но аппаратно-независимая;
- аппаратно-зависимая.

Ни одна из указанных стадий не является строго обязательной. На первой стадии выполняется оптимизация логической структуры программы и ее виртуальной памяти. К оптимизационным процедурам этого этапа можно отнести: оптимизацию циклов, устранение избыточных вычислений, преобразование индексных выражений, сокращение числа переменных и т.п. Эти методы требуют не пооператорного просмотра, а анализа всей программы или ее участка. Такая оптимизация может осуществляться либо на уровне исходного языка, либо на уровне

некоторого промежуточного языка. Подавляющее большинство современных языков высокого уровня воплощает принципы структурного программирования [10], а это означает, что, с одной стороны, эффективность алгоритма зачастую приносится в жертву его ясности, но, с другой, – что логическая структура программы получается строгой и, следовательно, легко анализируемой и оптимизируемой. Общность принципов структурного программирования для разных языков позволяют также формировать единое промежуточное представление разноязыковых программ и применять оптимизационные процедуры, не зависящие от языка.

Системно-зависимая стадия оптимизации связана более всего с дисциплинами управления памятью в ОС. Если компилятор "знает", какие алгоритмы управления памятью (например, страничного обмена) применяет ОС, он может "помочь" ОС. Поскольку компилятор, в отличие от ОС, обладает возможностью глобального анализа программы, он может предсказывать будущую потребность в тех или иных страницах памяти, влиять на размер и состав рабочего набора процесса. Компилятор может также перестроить программу таким образом, чтобы повысить степень локализации обращений к памяти и тем самым снизить интенсивность страничного обмена.

Роль аппаратно-зависимой оптимизации все более возрастает с развитием процессорных архитектур. Мы уже отмечали, что на компилятор, таким образом, возлагается задача расположения команд программы в такой последовательности, чтобы обеспечить максимальную загрузку конвейерных линий. Рассмотрим пример возможного дальнейшего развития этой идеи.

Недостатком современных суперскалярных процессоров является необходимость для процессора в каждом цикле анализировать поток команд, чтобы определить, какие конвейерные линии могут быть

использованы. Это является препятствием как для увеличения числа линий, так и для сокращения времени цикла. Перспективной для RISC-процессоров, по-видимому, является идея упаковки нескольких простых команд в одну большую команду фиксированной длины. Такая команда называется VLIW (very long instruction word – очень длинное командное слово). Составляющие VLIW-команды должны выполняться строго последовательно, сами VLIW-команды могут выполняться параллельно. Процессор, таким образом, просто загружает очередную VLIW-команду в очередную конвейерную линию, не занимаясь анализом командного потока. Задача формирования VLIW-команд с оптимизацией их под данную платформу ложится на компилятор. На сегодняшний день подобные подходы применяются, например, в процессорах фирмы Hewlett-Packard и процессоре Itanium фирмы Intel.

## **4.2. Компоновка и загрузка**

Очевидно, что ни одно сколько-нибудь сложное программное изделие не может быть реализовано в виде единственного модуля. Модульность программ и данных является необходимым условием структурного программирования при применении его как нисходящего, так и восходящего вариантов. Преимущества модульного конструирования могут, однако, в полной мере проявиться только, если обеспечивается раздельная компиляция модулей. Но если результаты раздельной компиляции представляют собой также отдельные объектные модули, то должен быть этап в подготовке программы, который бы компоновал из объектных модулей единую программу, готовую к выполнению, – загрузочный модуль. Кроме того, если компилятор обрабатывает каждый модуль отдельно, он не имеет возможности полностью выполнить функцию именования, – в объектном модуле остаются

непреобразованными обращения к процедурам и данным, определенным в других модулях. Обеспечение обращений к внешним по отношению к модулю именам называется связыванием. Операции компоновки и связывания выполняются специальными программами-компоновщиками (синоним – редактор связей, linker) и связывающими загрузчиками.

В соответствии с декларированной нами в предыдущей главе "свободой выбора" операцию связывания можно выполнять на разных этапах подготовки программы:

- статическое связывание этапа подготовки – выполняется компоновщиком, вызываемые модули включаются в загрузочный модуль программы;
- статическое связывание этапа загрузки – выполняется связывающим загрузчиком, вызываемые модули подключаются к программе уже в оперативной памяти;
- динамическое связывание этапа загрузки – отличается от предыдущего варианта тем, что обеспечивает совместное использование одних и тех же копий модулей в памяти разными программами;
- динамическое связывание этапа выполнения – загрузка модулей и установка связей происходит при выполнении программы и управляет ими сама программа.

Статическое связывание обеспечивается соглашениями о формате объектных модулей. В любых вариантах эти форматы предусматривают наличие в объектном модуле двух таблиц – таблицы входов и таблицы внешних ссылок. В первой перечисляются имена и относительные адреса в модуле тех процедур и элементов данных, к которым разрешены обращения из других модулей. Во второй – имена внешних точек, к которым имеются обращения из данного модуля, с каждым таким именем связан также список адресов внутри модуля, которые содержат обращения

к этому внешнему имени. При статическом связывании создается таблица входных точек – глобальная для всей программы. По мере компоновки программы из входящих в нее модулей их входные точки заносятся в эту таблицу, а их локальные адреса в модуле заменяются адресами в общем адресном пространстве программы. Затем в каждом модуле находятся (по таблице внешних ссылок) обращения к внешним модулям и имена в этих обращениях заменяются на адреса, получаемые из глобальной таблицы входных точек.

При статическом связывании может происходить также и формирование оверлейной структуры адресного пространства программы, рассмотренное в предыдущей главе.

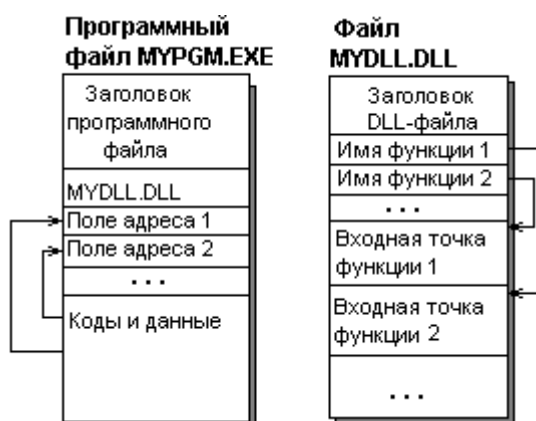
Вспомним, что на этапе загрузки может происходить также трансляция виртуальных адресов программы в физические адреса. В этом случае загрузочный модуль должен содержать таблицу перемещений – список тех адресов в программе, которые должны быть модифицированы при загрузке базовым адресом программы. Эти операции выполняются перемещающим загрузчиком, пример такого загрузчика – загрузчик EXE-файлов в MS DOS.

Статическое связывание и загрузка выполняются системными утилитами, не входящими в ядро ОС, они не составляют основной предмет нашего рассмотрения. Алгоритмы функционирования редакторов связей и перемещающих загрузчиков подробно рассматриваются, например, в [3, 13], здесь же мы остановимся на проблемах динамического связывания, выполняемого ОС.

Современные ОС позволяют сочетать статическую компоновку с динамической. Хотя в литературе, посвященной этим ОС, возможность динамического связывания описывается как принципиально новая, на самом деле она была впервые реализована еще в 1965 году в ОС MULTICS [30], и с тех пор ее механизмы не претерпели значительных изменений. В

современных ОС модули, подключаемые к программам динамически, носят название библиотек динамической компоновки (dynamic link library), соответственно, файлы, содержащие образы таких модулей имеют расширения DLL.

Выполнение динамической компоновки иллюстрируется рисунком 4.1. Структуры данных динамической компоновки в принципе сходны со структурами для компоновки статической. Для модуля основной программы компоновщик создает таблицу, функционально идентичную таблице внешних ссылок. В этой таблице указывается имя файла для каждого динамически подключаемого модуля и имена тех процедур в модуле, к которым имеются обращения в программе. Вызовы процедур в теле программы содержат обращения к соответствующим элементам этой таблицы. С другой стороны, при компоновке модуля, предназначенного для динамического подключения, для него создается таблица входов. При компоновке загрузчик (на этапе загрузки) или ядро ОС (на этапе выполнения) выполняет установку связей: находит в памяти или загружает в память динамически подключаемый модуль и по его таблице входов находит требуемую процедуру. Строка таблицы внешних ссылок модифицируется – теперь она содержит переход по адресу найденной процедуры. Последующие обращения к этой же входной точке уже не вызывают каких-либо дополнительных действий.





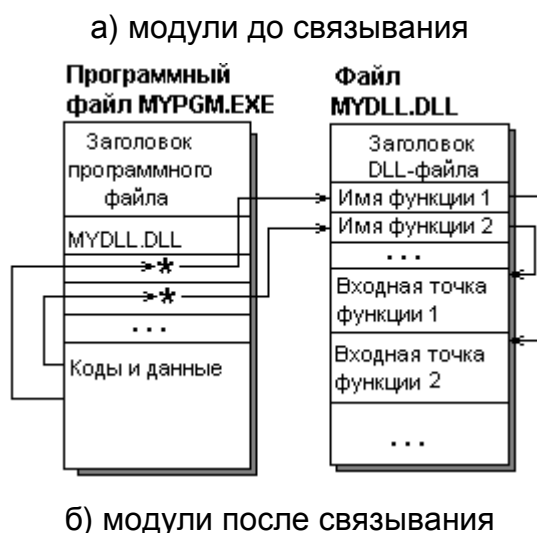


Рисунок 4.1 Установка межмодульных связей при динамической компоновке

При совместном использовании процедур, входящих в состав модулей DLL, несколькими процессами все процессы используют один и тот же экземпляр кода DLL, но для каждого использования создается отдельный стек, таким образом, для каждого использования имеется свой набор экземпляров локальных переменных процедуры.

Динамическая компоновка при загрузке совершенно прозрачна для программы. При подготовке программных и DLL-файлов требуются специальные директивы для компоновщика, но в самом тексте программы обращения к динамически подключаемым процедурам ничем не отличаются от обращений к процедурам, подключенным статически. В современных ОС динамически подключаемые библиотеки широко используются для системного программного обеспечения. API ОС, средства работы с терминалом, графические средства и т.п. представляют собой библиотеки динамической компоновки, совместно используемые всеми программами. При загрузке любой программы эти модули, как правило, уже находятся в памяти.

Динамическая компоновка несколько ухудшает быстродействие программы: во-первых, расходуется время на установление связей при

выполнении, во-вторых, при уже установленных связях обращение производится через посредство таблицы внешних ссылок. Преимущества же динамической компоновки бесспорны: во-первых, достигается экономия оперативной памяти за счет совместного использования модулей, во-вторых, достигается экономия внешней памяти за счет уменьшения объема загрузочных модулей, в-третьих, создается возможность модификации и полной замены модулей динамической компоновки без изменения двоичного кода главной программы.

Динамическая компоновка на этапе выполнения дает возможность программе самой управлять порядком загрузки модулей и установки связей. Программа может, например, менять загруженный в память набор модулей в разных фазах своего выполнения или использовать разные модули в зависимости от конфигурации вычислительной системы или условий выполнения и т.п. Для обеспечения таких возможностей ОС должна предоставлять в распоряжение программиста соответствующий набор системных вызовов. Этот набор может быть следующим.

Системный вызов:

```
mod_handle = loadModule (mod_name);
```

загружает модуль из файла с указанным именем. Если указанного модуля нет в памяти, ОС производит загрузку его из файла, если же модуль в памяти уже есть, ОС просто увеличивает на 1 счетчик использований этого модуля. Вызов должен возвращать манипулятор модуля, используемый для его идентификации во всех последующих операциях с ним.

Возможна модификация этого вызова:

```
mod_handle = getModuleHandle (mod_name);
```

получить манипулятор модуля: если модуль уже есть в памяти, вызов возвращает его манипулятор (и увеличивает счетчик использований), если нет – возвращает признак ошибки. В последнем случае программа может

либо загрузить модуль вызовом `loadModule`, либо перейти на такую свою ветвь, которая не предусматривает обращений к данному модулю.

Системный вызов:

```
freeModule (mod_handle);
```

выгружает модуль. ОС уменьшает на 1 счетчик использования модуля, если этот счетчик обратился в 0, освобождает память.

Системный вызов:

```
vaddr = getProcAddress (mod_handle, proc_name,  
proc_addr);
```

возвращает виртуальный адрес процедуры с заданным именем в уже загруженном модуле. Все дальнейшие обращения к данной процедуре программа производит по этому адресу.

### **4.3. Цикл жизни процесса**

Программа, готовая к выполнению, превратится в процесс только тогда, когда ОС создаст для нее блок контекста и запись в системной таблице процессов. ОС существенно различаются по тому признаку, насколько часто они создают новые процессы и сколько процессов могут одновременно существовать в системе.

В однозадачных системах существует один процесс (или несколько процессов, только один из которых – пользовательский), который последовательно выполняет одну программу за другой.

В многозадачных ОС, осуществляющих пакетную обработку, процессов, обслуживающих пользователей, несколько, но число их либо фиксировано и устанавливается при загрузке, либо меняется оператором, и такие изменения происходят весьма редко.

Различные подходы могут применяться в интерактивных системах.

Во-первых, в интерактивной системе может копироваться стратегия многозадачной системы с пакетной обработкой: с каждым терминалом связывается единственный процесс-сеанс (session). Таким образом, предельное число процессов в системе ограничивается числом терминалов. Пользователь каждого терминала работает как бы в однозадачной среде (VM).

Во-вторых, для преодоления стесненности пользователя в сеансе система может позволять в ходе сеанса порождать дополнительные, фоновые (background) процессы. Такие процессы выполняют программы, не требующие в ходе выполнения взаимодействия с оператором. Фоновые процессы работают параллельно с процессом, поддерживающим интерактивную работу в сеансе.

Наконец, в-третьих, система может позволять порождать любые процессы и в любом количестве. Для каждой новой выполняемой программы создается новый процесс, который уничтожается с завершением программы. Процессы могут выполняться как последовательно, так и параллельно, ограничением на количество параллельно выполняемых процессов является объем ресурсов вычислительной системы и ОС, в частности, возможные ограничения на предельный размер таблицы процессов. Подход без ограничений на число процессов иногда называют философией "дешевых" процессов. В таких системах "накладные расходы" на создание процессов минимальны и наблюдается тенденция к предельному упрощению отдельных процессов: каждый отдельный процесс реализует некоторую весьма элементарную функцию, а сложные действия реализуются как комбинации тех или иных элементарных действий.

Процессы в системе могут либо существовать как отдельные, не связанные друг с другом единицы, либо образовывать какие-либо структуры. Обычно модель независимого существования процессов

используется в тех ОС, которые накладывают ограничения на возможности создания процессов (такая возможность доступна только самой ОС). В тех ОС, которые разрешают процессам в свою очередь порождать новые процессы, применяется иерархическая структура связей между процессами, в которой между процессами существуют отношения "родитель – потомок". Такая структура позволяет установить четкие правила в отношении использования ресурсов и организации управления. В такой структуре все действующие процессы могут в конечном счете являться потомками (более или менее близкими) одного системного процесса.

Передача ресурсов от предка к потомку позволяет родственным процессам легко устанавливать связи друг с другом. Наиболее важными в этом отношении ресурсами являются файлы, каналы и другие средства взаимодействия (см. главу 9). В соответствии с принятыми в конкретной ОС правилами потомок может получать либо все ресурсы родителя, либо только те, для которых установлен признак "наследуемые" (inheritance), этот признак может быть установлен либо при выделении ресурса, либо для уже выделенного ресурса – специальным системным вызовом.

Если пользовательскому процессу дана возможность порождать потомков, то в его распоряжении должны быть соответствующие системные вызовы. Конкретные возможные вызовы мы рассмотрим ниже, но сначала остановимся на общей семантике порождающих вызовов.

Процесс-потомок может быть запущен синхронно или асинхронно с родителем. При синхронном запуске процесс-родитель блокируется до завершения процесса-потомка. Естественно, что в этом случае родитель не может влиять на выполнение потомка. При асинхронном запуске родитель и потомок продолжают выполняться параллельно и могут взаимодействовать. Если родителю теперь необходимо дождаться завершения потомка, он выдает системный вызов ожидания. Синхронный

запуск не является обязательной возможностью, так как тот же эффект может быть обеспечен парой вызовов: "асинхронный запуск" – "ожидание". Для того, чтобы родитель мог воздействовать на потомка, вызов "запуск" возвращает ему идентификатор (манипулятор) порожденного процесса. Этот идентификатор используется родителем при последующих воздействиях на потомка. Поскольку потомок может в свою очередь создавать новые процессы, то есть стать во главе целого подсемейства (поддерева) процессов, интерпретация идентификатора процесса в системных вызовах, связанных с воздействиями на него, должна быть четко определена системными соглашениями или дополнительным параметром таких вызовов. Возможные интерпретации идентификатора следующие:

- только тот процесс, идентификатор которого задан;
- процесс, идентификатор которого задан и все его потомки;
- процесс, идентификатор которого задан, или любой из его потомков.

В конкретных системах могут вводиться некоторые разумные ограничения на число порождаемых процессов, связанные с предельными размерами таблицы процессов и очереди к планировщику. Может ограничиваться, например, число потомков у процесса или число процессов, принадлежащих одному пользователю.

Процесс-потомок при запуске не знает идентификатора своего родителя, но, как правило, может его получить при помощи системного вызова.

Ниже мы приводим набор системных вызовов, обеспечивающих порождение процессов и "родственные отношения" между ними.

Порождение нового процесса и выполнение в нем программы:

```
pid = load(filename);
```

для нового процесса создается новая запись в системной таблице процессов и блок контекста. В блоке контекста формируется описание адресного пространства процесса – например, таблица сегментов. Выполняется формирование адресного пространства – образы некоторых частей адресного пространства (сегментов) процесса (коды и инициализированные статические данные) загружаются из файла, имя которого является параметром вызова, выделяется память для неинициализированных данных. Формирование всех сегментов не обязательно происходит сразу же при создании процесса: во-первых, если ОС придерживается "ленивой" тактики, то формирование памяти может быть отложено до обращения к соответствующему сегменту; во-вторых, в загрузочный модуль могут быть включены характеристики сегментов: предзагружаемый (preload) или загружаемый по вызову (load-on-call). Новому процессу должна быть выделена также и вторичная память – для сохранения образов сегментов/страниц при свопинге. Часть вторичной памяти для процесса уже существует: это образы неизменяемых сегментов процесса в загрузочном модуле. Для более эффективного поиска таких сегментов ОС может включать в блок контекста специальную таблицу, содержащую адреса сегментов в загрузочном модуле. При выделении вторичной памяти для изменяемых сегментов все ОС обычно следуют "ленивой" тактике. Ресурсы процесса-родителя копируются в блок контекста потомка. В вектор состояния нового процесса заносятся значения, выбор которых в регистры процессора приведет к передаче управления на стартовую точку программы. Новый процесс ставится в очередь готовых к выполнению. Вызов `load` возвращает идентификатор порожденного процесса.

Смена программы процесса:

```
exec (filename);
```

Завершается программа, выдавшая этот системный вызов, вместо нее запускается другая программа. Вызов `exec` может быть реализован как комбинация вызовов `exit` (завершить текущий процесс) и `load` (создать новый процесс), но может и не порождать смену процессов, а только обновлять адресное пространство (включая и блок контекста) текущего процесса. В последнем случае сохраняются также и ресурсы процесса. Идентификатор процесса не изменяется.

Расщепление процесса:

```
pid = fork();
```

– порождается новый процесс – копия процесса-родителя. При копировании таблицы сегментов родителя в блок контекста потомка принимаются во внимание характеристики сегментов:

- уникальный – сегмент может принадлежать только одному процессу, в таблицу потомка этот сегмент не копируется;
- разделяемый – элемент таблицы сегментов потомка совершенно идентичен элементу таблицы родителя, включая базовый адрес в реальной памяти;
- копируемый – для потомка выделяется новый сегмент в реальной памяти, в него копируется содержимое соответствующего сегмента родителя и элемент таблицы сегментов потомка содержит базовый адрес нового сегмента.

Для копируемых сегментов часто применяется "ленивая" тактика копирования при записи (copy-on-write). В этом случае выделение сегмента и копирование данных откладывается, родитель и потомок используют один и тот же сегмент в реальной памяти, но в их таблицах этот сегмент помечается как недоступный для записи. Совместное использование сегмента для чтения продолжается, пока один из процессов не попытается писать в него. Попытка записи вызовет прерывание-ловушку по нарушению доступа, обработчик этого прерывания обеспечит создание



копии сегмента, изменит его базовый адрес в таблице потомка и снимет с сегмента защиту записи.

При выполнении вызова `fork` копируется также и счетчик команд процесса-родителя. Выполнение потомка, таким образом, начнется с возврата из системного вызова `fork` и возникает проблема самоидентификации процесса. Процесс, вернувшийся из системного вызова `fork`, должен определиться, кто он – родитель или потомок? Семантика вызова `fork` обуславливает его применение в таком контексте:

```
if ( ( childpid = fork() ) == 0 )  
    < процесс-потомок >;  
else < процесс-родитель >;
```

т.е., вызов возвращает 0 процессу-потомку и идентификатор потомка – процессу-родителю.

В ОС Unix вызов `fork` является единственным средством создания нового процесса. Поэтому, как правило, ветвь, соответствующая процессу-потомку, содержит вызов `exec`, меняющий программу, выполняемую потомком. Эффект в этом случае будет тот же, что и при выполнении вызова `load`. Применение именно такой схемы, видимо, объясняется легкостью передачи ресурсов. Ниже мы покажем структуру отношений системных и пользовательских процессов в Unix.

Ожидание завершения потомка:

```
exitcode = waitchild(pid);
```

процесс-родитель блокируется до завершения процесса-потомка, идентификатор которого является параметром вызова (или всего семейства, возглавляемого этим потомком, или любого из членов этого семейства). Вызов может возвращать код завершения процесса-потомка (задается вызовом `exit`) и идентификатор завершившегося потомка.

Разумеется, применение этого вызова имеет смысл только при асинхронном запуске потомка.

Выход из процесса:

```
exit(exitcode);
```

приводит к освобождению занятых процессом ресурсов, в том числе и ресурса памяти. Ресурсы, запрошенные процессом динамически, требуют явного освобождения процессом (например, процесс должен закрыть все открытые им файлы), но если процесс "забыл" это сделать, это сделает за него ОС при выполнении данного вызова. При выполнении `exit` также могут выполняться процедуры, заданные вызовами `exitlist` (см. ниже). Вызов `exit` не обязательно должен приводить к немедленному полному уничтожению процесса. Может сохраняться соответствующая ему запись в таблице процессов и часть блока контекста, но процесс помечается как завершённый. Неполное удаление процесса объясняется тем, что после процесса остается еще некоторая информация, которая может быть востребована, статистические данные о его выполнении, код завершения (параметр вызова `exit`), который будет прочитан вызовом `waitchilde` в родителе и т.п. Полное удаление процесса произойдет после того, как вся остаточная информация будет обработана.

Формирование списка выхода:

```
exitlist(procaddr);
```

при помощи этого вызова процесс может установить процедуру (адрес такой процедуры – параметр вызова), которая должна быть выполнена при его завершении. Процедуры выхода обычно используются для сохранения параметров программы и аккуратного закрытия каких-либо важных ресурсов при аварийном завершении программы. Процесс может выдать несколько вызовов `exitlist`, назначив несколько процедур выхода, которые будут выполняться в неопределенном порядке. Процедура выхода может также иметь параметр, через который ей будет передаваться

причина завершения: нормальное / по сигналу / по программной ошибке / по аппаратной ошибке.

Принудительное завершение:

```
kill(pid);
```

завершает процесс-потомок или все семейство процессов, им возглавляемое. Выполнение этого вызова заключается в выдаче сигнала `kill` (механизм сигналов описывается в главе 9), по умолчанию обработка этого сигнала вызывает выполнение `exit` с установкой специального кода завершения.

Изменить приоритет:

```
setPriority ( pid, priority );
```

изменяет приоритет потомка или всего его семейства. Приоритет может задаваться как абсолютный, так и в виде приращения (положительного или отрицательного) к текущему приоритету. Как правило, пользовательские процессы не могут изменять свой приоритет в сторону увеличения.

Получение идентификаторов:

```
pid = getpid(mode);
```

вызов возвращает процессу его собственный идентификатор и/или идентификатор процесса-родителя.

На рисунке 4.2 приведена в качестве примера схема наследования процессов в ОС Unix. Корнем дерева процессов является процесс `init`, создаваемый при загрузке ОС. Процесс `init` порождает для каждой линии связи (терминала) при помощи пары системных вызовов `fork-exec` свой процесс `getty` и переходит в ожидание. Каждый процесс `getty` ожидает ввода со своего терминала. При вводе процесс `getty` сменяется (системный вызов `exec`) процессом `logon`, выполняющим проверку пароля. При правильном вводе пароля процесс `logon` сменяется (`exec`)

процессом `shell`. Командный интерпретатор `shell` (подробнее мы рассмотрим его в главе 11) является корнем поддерева для всех процессов пользователя, выполняющихся в данном сеансе. При завершении сеанса `shell` завершается (`exit`), при этом "пробуждается" `init` и порождает для этого терминала новый процесс `getty`.

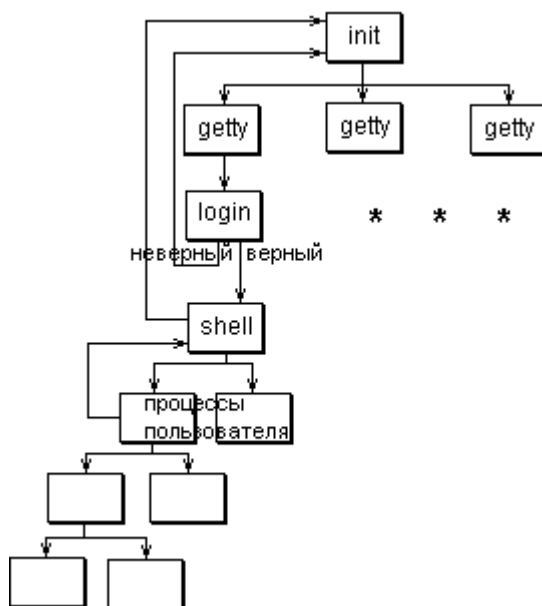


Рисунок 4.2 Процессы в ОС Unix

Хотя на рисунке 4.2. между процессами `init` и `shell` находятся еще два процесса, `shell` может считаться прямым потомком `init`, так как находящиеся между ними процессы завершаются при запуске потомка.

В других ОС может выстраиваться и более сложная иерархия системных процессов. Так, например, в OS/390 между ядром ОС и задачей (`task` – синоним процесса в терминах IBM) пользователя находятся еще задачи:

- управления разделом;
- системного дампа;
- управления командой `START`;

- инициатор (для пакетных задач) или LOGON (для интерактивных задач).

Эти задачи существуют параллельно и выполняют управление ресурсами на разных уровнях.

При завершении родительского процесса не обязательно завершаются все его потомки. Некоторые процессы могут продолжать выполнение даже после завершения их непосредственного предка. Предком таких осиротевших процессов обычно становится процесс, находящийся в корне всего дерева наследования. Такие процессы обычно применяются для выполнения каких-либо фоновых задач, например, для ожидания и обработки каких-то внешних событий, и за ними закрепилось название демоны (daemon).

## 4.4. Нити

Философия дешевых процессов подразумевает, что процесс может быть создан легко и быстро. С одной стороны, это позволяет в максимальной степени обеспечивать распараллеливание работ по решению нескольких задач или внутри одной задачи. Но, с другой стороны, если ОС выполняет большой объем работ по управлению ресурсами, то создание нового процесса и выделение ему ресурсов не может обойтись без значительных "накладных расходов". Преодоление этого противоречия было найдено в концепции "нитей" (thread, часто переводится также как "поток"), реализованной в большинстве современных ОС и зафиксированной в стандартах POSIX и DCE. Нитью называется отдельная ветвь выполнения процесса. Процесс может состоять из одной или нескольких нитей, которые совместно используют ресурсы процесса, но являются самостоятельными объектами при планировании процессорного

времени. Таким образом, нити одного процесса могут выполняться параллельно. Концепция не оговаривает, какие именно ресурсы являются общими, а какие – локальными для нити. В большинстве современных ОС помимо ресурса процессорного времени и вектора состояния процесса, нить имеет собственный:

- вектор состояния,
- приоритет,
- стек (в стеке же размещаются и локальные переменные).

Прочие же ресурсы: общие переменные, файлы, средства взаимодействия и т. д. – являются общими для нити и породившего ее процесса.

Любой процесс состоит из одной или нескольких нитей. Первая нить – основная – порождается системой при запуске процесса. Основная нить может порождать дочерние нити вызовом типа:

```
tid = createThread(thr_addr, stack).
```

Параметрами вызова являются: `thr_addr` – адрес нити; `stack` – адрес стека, выделяемого для нити (стек может также назначаться системой по умолчанию). Вызов возвращает идентификатор нити.

В программе (на языке C, например) нить выглядит как обычная функция. Приведенный выше вызов передает управление на эту функцию, и далее выполнение функции происходит параллельно с выполнением основной программы. Поскольку при порождении каждой новой нити выделяется отдельный стек, одна и та же функция может выполняться в составе двух и более параллельных нитей. При порождении новая нить наследует приоритет породившей ее, но далее этот приоритет может быть изменен. Отношения между основной и дочерней нитями похожи на отношения между процессами – родителем и потомком, рассмотренные в предыдущем разделе.

Для управления выполнением нитей в рамках одного процесса обычно в составе API ОС имеются системные вызовы, позволяющие приостановить выполнение нити и вновь разрешить ее выполнение:

```
suspendThread(tid);  
resumeThread(tid).
```

Отметим две особенности, связанные с возможностью порождения нитей, которые создают некоторые дополнительные трудности для пользователей и для ОС. Во-первых, трудность для пользователей – поскольку нити разделяют общие ресурсы процесса, на программиста ложится задача обеспечить корректность совместного использования ресурсов (см. главы 8 и 9). Во-вторых, трудность для ОС – усложняется задача планировщика процессорного времени: теперь единицей планирования становится не процесс, а нить и планировщик должен обеспечивать справедливость распределения ЦП как между процессами, так и между нитями в каждом процессе.

## **КОНТРОЛЬНЫЕ ВОПРОСЫ**

1. Каким образом при различных внутренних структурах и даже механизмах обращения к ОС может быть обеспечен одинаковый API для разных ОС?

2. Какие стадии оптимизации может проходить программа? Какие стадии оптимизации могут быть одинаковыми для программ, написанных на языках C, Pascal, Cobol, Fortran и на языке Ассемблера? Почему для современных процессорных архитектур оптимизация является обязательной?

3. В чем преимущества динамического связывания по сравнению со статическим?

4. Почему во многих современных ОС значительная часть системы выполняется в виде библиотек динамической компоновки?
5. Являются ли "родственные отношения" между процессами обязательными? Являются ли они полезными?
6. Сравните стратегии систем, в которых порождение процессов выполняется вызовом `fork` и вызовом `load`.
7. Для чего могут быть полезны списки выхода? Приведите примеры задач, для которых было бы целесообразно иметь список выхода, состоящий из более чем одной процедуры.
8. Дайте определение нити. Какие ресурсы являются собственными для нити?
9. Приведите примеры задач, решение которых требует применения нитей?
10. В некоторых клонах ОС Unix нет специального механизма нитей, но нити реализованы как процессы, наследующие адресное пространство родителя. Чем вы объясните такое решение?



## **Глава 5. Монопольно используемые ресурсы**

### **5.1. Свойства ресурсов и их представление**

Процессорное время и оперативная память являются ключевыми ресурсами любой ОС, без них не может выполняться ни один процесс. Ресурсы, которые мы рассматриваем в этой главе, являются монопольно используемыми: неперераспределяемыми и неразделяемыми. Свойство неперераспределяемости означает, что ресурс не может быть отобран у процесса во время его использования. Представьте себе, что процесс выводит платежную ведомость на принтер. Если мы в середине печати отберем у процесса ресурс-принтер и отдадим его другому процессу, то когда первый процесс вновь обретет этот ресурс, ему придется начать печать сначала. Как мы увидим дальше, неотбираемых ресурсов в системе быть не должно, поэтому уточним понятие неперераспределяемости: ресурс не может быть отобран без фатальных для процесса последствий. На том же примере печати поясним понятие неразделяемости: два процесса не могут выводить данные на один и тот же принтер одновременно. Ресурсы процессорного времени и памяти, как мы увидели выше, свойствами неперераспределяемости и неразделяемости не обладают.

К группе монопольных ресурсов относятся многие устройства ввода-вывода: принтеры, магнитные ленты, каналы связи и т.п., файлы (они могут быть разделяемыми, но со значительными оговорками). Не забудем

также вторичные ресурсы, порождаемые самой ОС, – системные структуры данных и коды. Так, например, при создании нового процесса необходимо занести новую запись в таблицу процессов. Эта запись также является ресурсом, причем неперераспределяемым и неразделяемым.

Ресурсы, которые мы рассматриваем, являются также повторно используемыми. Это означает, что ресурсы после их использования процессами не пропадают и не убывают, а могут быть использованы другим процессом. Альтернативу им составляют потребляемые ресурсы, которыми чаще всего могут быть входные данные и сообщения, поступающие в процесс извне.

Наши ресурсы обладают также свойствами дискретности и ограниченности. Первое означает, что ресурсы распределяются некоторыми неделимыми единицами (не может быть полтора принтера). Второе – то, что число единиц ресурса всегда бесконечно. (Процессорное время бесконечно: его достаточно для выполнения любого процесса, и оно может дробиться планировщиком. Реальная память всегда конечна, виртуальная тоже конечна, ограничена разрядностью виртуального адреса, а непрерывность или дискретность ее зависит от принятой модели).

Мы будем называть классом ресурса пул идентичных неименованных единиц ресурса. Неименованными мы считаем их в том смысле, что процесс при запросе ресурса не указывает, какую именно единицу из пула он хочет получить, все единицы ресурса одинаковы. Все ресурсы одного класса управляются одним менеджером.

Из определений ОС (как с точки зрения разработчика, так и с точки зрения пользователя), которые мы дали в первой главе, однозначно следует, что процесс ни в коем случае не может самостоятельно завладеть ресурсом, а только через посредство ОС. Для предоставления процессам

такой возможности в составе API ОС должны быть системные вызовы типа:

```
resourceHandle = getResource  
                (class, number [,action] );  
releaseResource(resourceHandle).
```

Первый вызов выделяет процессу `number` ресурсов из класса `class` и возвращает манипулятор (`handle`) выделенного ресурса, который при всех дальнейших операциях процесса с ресурсом служит для идентификации ресурса. Манипулятор каким-то образом адресует дескриптор ресурса. В защищенных системах такой дескриптор располагается в недоступном для процесса адресном пространстве. Манипулятор обычно является номером в системной таблице или списке дескрипторов, и по нему ядро (но не процесс) выбирает требуемый дескриптор ресурса.

Второй вызов открепляет от процесса ранее выделенный ему ресурс. Возможно, форма выделения/освобождения ресурса напомнила вам знакомые операции открытия/закрытия файла – и не даром. Поскольку файлы также являются ресурсами, операции `open/close` – частные случаи операций `getResource/releaseResource`. Как правило, в реальных API ОС нет общих операций выделения/освобождения ресурсов, но для каждого ресурса имеется своя пара операций, отличающаяся от других названием и, возможно, составом параметров.

Третий, необязательный параметр операции `getResource` задает действия ОС в ситуации, когда выделить ресурс невозможно (не все ОС предоставляют процессам возможности такого выбора). Во-первых (и это действие обычно выполняется по умолчанию), ОС может заблокировать процесс, выдавший запрос, до освобождения требуемого ресурса. Во-вторых, ОС может не блокировать процесс, а вернуть ему отказ – сразу или после некоторой временной выдержки (`timeout`), в этом случае "умный"

процесс может пока заняться другой работой, которую он может выполнить и без этого ресурса, а позже повторить запрос. Как ОС должна реагировать на запрос, который вообще не может быть выполнен (требуемого объема ресурсов просто нет в системе)? По нашему убеждению, такой запрос должен приводить к аварийному завершению выдавшего его процесса. Но если ОС допускает динамическую реконфигурацию ресурсов, то запрос может быть поставлен в очередь в ожидании момента, когда ресурс будет введен в систему. Такие нереализуемые запросы могут составлять серьезную неприятность в работе ОС, так как ресурс может никогда и не быть введен. Для избежания таких запросов целесообразно иметь в составе API вызов, возвращающий общее число ресурсов данного класса. Наконец, при невозможности удовлетворить запрос ОС может потребовать от процесса освободить уже имеющиеся в его распоряжении ресурсы. Если такая возможность имеется, то она может быть очень полезна для развязки тупиков (см. ниже).

Для каждого класса ресурсов ОС должна поддерживать дескриптор класса, в который должны входить:

- идентификатор класса;
- общее число единиц в классе;
- число свободных единиц;
- таблица единиц ресурса;
- список процессов, ожидающих ресурс этого класса;
- точка входа в менеджер класса;
- и т.д.

Для каждой единицы ресурса имеется запись в таблице единиц, содержащая, как минимум, индикатор занятости ресурса и идентификатор процесса, которому ресурс распределен (если он не свободен).

Манипулятор ресурса каким-то образом адресует дескриптор ресурса. В защищенных системах сам дескриптор ресурса располагается в адресном пространстве, недоступном для процесса. Манипулятор представляет собой номер дескриптора в системной таблице или в системном списке дескрипторов ресурсов данного типа. При выполнении системного вызова, параметром которого является манипулятор ресурса, происходит переключение в контекст ядра, для модуля ОС, выполняющего системный вызов, таблица дескрипторов становится непосредственно доступной, и этот модуль выполняет действия над дескриптором, номер которого он получил в качестве параметра.

Информация о ресурсах, выделенных процессу, хранится также в блоке контекста процесса.

## **5.2. Обедающие философы**

Уже классической стала неформальная постановка задачи распределения ресурсов, носящая название "проблемы обедающих философов" и показанная на рисунке 5.1.

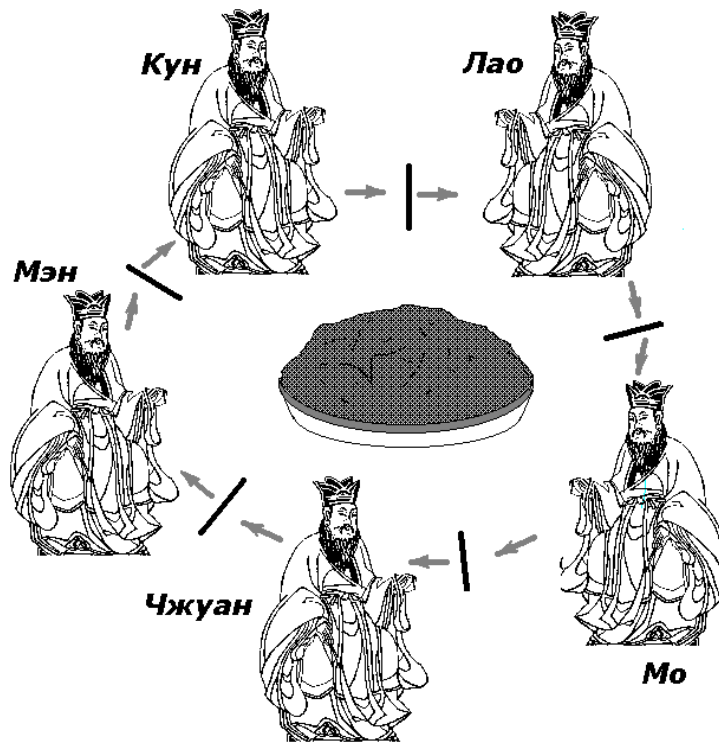


Рисунок 5.1 Обедающие философы. Тупик

Пять философов сидят за круглым столом, в центре которого стоит блюдо с рисом. Между каждой парой философов лежит палочка для еды, палочек, следовательно, тоже пять. Для того, чтобы начать есть, философ должен взять две палочки – слева и справа от себя. Таким образом, если один из философов ест, его соседи справа и слева лишены такой возможности, так как им недостает палочек. Каждый философ "работает" по замкнутому алгоритму: сначала он некоторое время думает, затем берет палочки и ест, затем опять думает и т.д. Временные интервалы мышления и еды случайны, действия философов, следовательно, не синхронизированы. Ничего не говорится в условии о том, каким образом философ берет палочки – наша задача как раз и состоит в том, чтобы обеспечить такую стратегию выделения палочек, которая бы исключала тупики и бесконечное откладывание.

Если мы установим, что каждый философ должен взять одну палочку и не выпускать ее из рук до тех пор, пока не возьмет вторую палочку, то

мы можем получить ситуацию, показанную на рисунке 5.1. (Стрелка от философа к палочке означает, что философ хочет взять эту палочку, стрелка в обратном направлении – что эту палочку этот философ уже взял.) Каждый из философов взял палочку справа от себя, но не может взять палочку слева. Ни один из философов не может ни есть, ни думать. Эта ситуация и называется тупиком (deadlock).

Если же мы установим, что философ должен взять обе палочки сразу, то может возникнуть ситуация, показанная на рисунке 5.2. Философ Чжуан хочет взять палочки, но обнаруживает, что его правая палочка занята философом Мо. Чжуан ожидает. Тем временем философ Мэн берет свои палочки и начинает есть. Мо есть заканчивает, но Чжуан не может начать есть, так как теперь занята его левая палочка. Если Мо и Мэн едят попеременно, то Чжуан попадает в положение, которое называется голоданием (starvation) или бесконечным откладыванием.

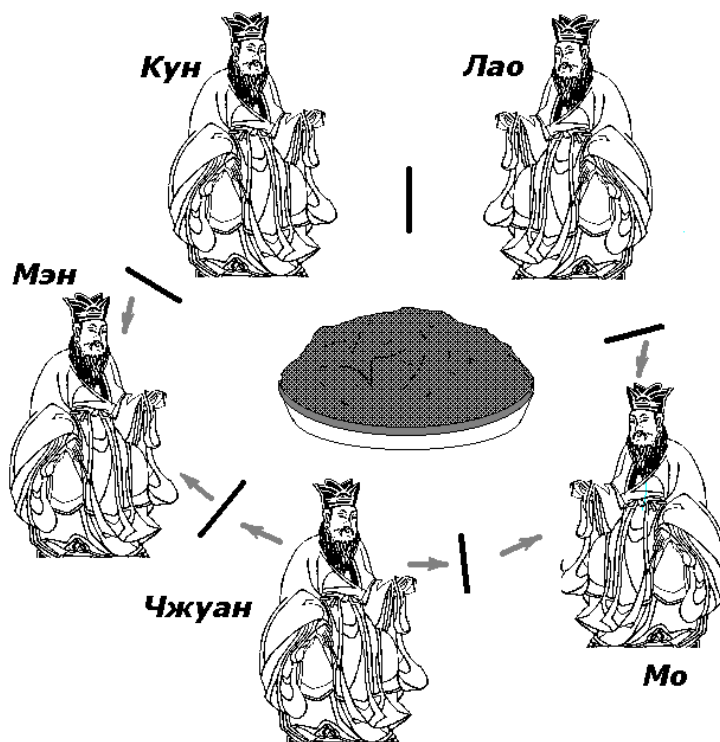


Рисунок 5.2 Обедающие философы. Бесконечное откладывание

Переходя от философов к вычислительным системам, мы можем проиллюстрировать тупик таким примером. Процесс А использует магнитную ленту, но для завершения ему нужен еще и принтер. В это время процесс В удерживает за собой принтер, но ему нужна еще магнитная лента. Процессы А и В блокируют друг друга, то есть находятся в тупике. В системах с множественными ресурсами и с высоким уровнем мультипрограммирования тупиковые ситуации могут быть и не столь очевидными. Тупики могут быть локальными и глобальными. Так, если в приведенном выше примере уровень мультипрограммирования выше 2, то процессы А и В находятся в локальном тупике, другие процессы, которым не требуются ресурсы, занятые процессами А и В, могут выполняться. Философы же на рисунке 5.1 находятся в глобальном тупике.

Бесконечное откладывание – ситуация даже более общая, свойственная управлению любыми ресурсами, а не только монопольными. Так, при планировании процессорного времени по статическим приоритетам низкоприоритетный процесс может откладываться до бесконечности, если в систему постоянно поступают процессы с более высокими приоритетами.

Тупик представляет собой ситуацию более опасную, чем бесконечное откладывание: процессы, попавшие в тупик, удерживают при этом системные ресурсы. Даже если тупик не глобальный, система продолжает работать с уменьшенным объемом ресурсов, следовательно, с пониженной производительностью. Бесконечное же откладывание одного или нескольких процессов может и не повлиять на среднюю пропускную способность системы, но, конечно же, влияет на показатели справедливости обслуживания.

### **5.3. Тупики: предупреждение, обнаружение, развязка**



Борьба с тупиками включает в себя три задачи:

- предупреждение тупиков – какую стратегию распределения ресурсов выбрать, чтобы тупики не возникали вообще?
- обнаружение тупиков – если не удалось применить стратегию, предупреждающую тупики, то как обнаружить возникший тупик?
- развязка тупиков – если тупик обнаружен, то как от него избавиться?

Возможные стратегии распределения ресурсов располагаются между двумя полюсами – от самых либеральных до самых консервативных. Чем либеральнее стратегия, тем "охотнее" ОС удовлетворяет запросы на ресурсы. Но за слишком либеральную стратегию приходится расплачиваться возможностью возникновения тупика. Консервативные стратегии делают тупики невозможными в принципе, задачи обнаружения и развязки при применении таких стратегий не ставятся, но плата за это – частые отказы в выделении ресурсов, следовательно, снижение уровня мультипрограммирования, а следовательно, – и снижение пропускной способности. Ниже мы будем рассматривать стратегии предотвращения, двигаясь от консервативного полюса к либеральному в таком порядке:

- последовательное выделение;
- залповое выделение;
- иерархическое выделение;
- выделение по предварительным заявкам.

### Последовательное выделение

Любыми ресурсами может одновременно пользоваться только один процесс. Если процесс А из предыдущего примера получил ресурс-принтер, то процессу В будет отказано даже в выделении ресурса-ленты. Очевидно, что такая стратегия делает тупики совершенно невозможными.

Очевидно также, что некоторые процессы будут при этом простаивать совершенно неоправданно. Так, например, будет приостановлен некий процесс С, которому принтер и не нужен, а нужна только лента. Поскольку в число распределяемых ресурсов входят устройства ввода-вывода, а работают они медленно (например, печать на принтере), простой может затянуться. Эта стратегия неоправданно снижает уровень мультипрограммирования и неэффективно использует ресурсы (они тоже простаивают) и может применяться только в таких ОС, в которых и расчетный уровень мультипрограммирования невысок.

### Залповое выделение

Процесс должен запрашивать / освобождать все используемые им ресурсы сразу. Эта стратегия позволяет параллельно выполняться процессам, использующим непересекающиеся подмножества ресурсов. (Процесс С работает с лентой, процесс D – с принтером.) Тупики по-прежнему невозможны, однако неоправданное удерживание ресурсов продолжается. Так, если процессу в ходе выполнения нужны ресурсы R1 и R2, причем ресурс R1 он использует все время своего выполнения  $t_1$ , а ресурс R2 требуется ему только на время  $t_2 \ll t_1$ , то процесс вынужден удерживать за собой и ресурс R2 в течение всего времени  $t_1$ .

В рамках залповой стратегии возможны два варианта: выделять все ресурсы при создании процесса и освобождать при его завершении или же позволить процессу запрашивать/освобождать ресурсы несколько раз в ходе своего выполнения (но обязательно все "залпом"). Очевидно, что второй вариант более либеральный, так как он позволяет уменьшить интервал времени удерживания ресурсов и разнести использование разных ресурсов по разным "залпам". Интересны различия в API для этих двух вариантов. В первом случае требования на ресурсы могут быть вообще вынесены за пределы программного кода процесса и задаваться во

внешних описателях процесса (например, в языке управления заданиями). Во втором случае системный вызов `getResource` является обязательным, причем обязательно должна быть обеспечена возможность запроса в одном вызове ресурсов разных классов и выделение всех запрошенных ресурсов одной непрерываемой операцией.

### Иерархическое выделение

Все классы ресурсов разбиваются по уровням с номерами от 1 до N, каждый уровень содержит только один класс. Процесс имеет право запрашивать ресурсы только из классов с более высокими номерами, чем у тех, которыми он уже владеет. Эта стратегия также предотвращает возникновение тупиков. В каждый момент времени в системе один или несколько процессов имеют класс закрепленных за ними ресурсов выше, чем у других. Эти процессы, обладающие ресурсами высокого уровня, могут беспрепятственно выполняться и завершиться без блокировки. Следовательно, в каждый момент времени имеется хотя бы один способный к выполнению процесс. Если не будут поступать новые процессы, то все процессы, уже имеющиеся в системе, в конце концов завершатся – тупик отсутствует. Хотя в иерархической стратегии процесс ограничен в последовательности запросов и возможна ситуация, в которой он должен удерживать за собой ресурс более длительное время, чем это действительно необходимо, эта стратегия позволяет достичь неплохой эффективности, особенно при правильном распределении ресурсов по уровням. Целесообразно более высокие уровни назначать более дефицитным и более дорогостоящим ресурсам; как правило, и использование таких ресурсов является более скоротечным. Иерархическую стратегию применяет, например, OS/390 применительно к некоторым системным структурам данных.

Иерархическая стратегия является самой либеральной из стратегий, предотвращающих возникновение тупиков без дополнительной информации. Более либеральные стратегии предотвращения требуют предварительного знания о характеристиках процессов.

### Предварительные заявки и алгоритм банкира

Эта стратегия названа так потому, что действия ОС напоминают действия банкира, выдающего ссуды клиентам, именно на таком примере эта стратегия была рассмотрена в первоисточнике [11]. Применение этой стратегии требует, чтобы процесс передал ОС "предварительную заявку" (advanced claim) и в ней указал максимальный объем ресурсов, который ему понадобится. В ходе выполнения процесс может в произвольном порядке запрашивать / освобождать ресурсы, не выходя, однако, за пределы заявленного объема. Ситуация в системе называется реализуемой при следующих условиях:

- ни одна заявка не превышает общего числа ресурсов в системе;
- ни один процесс не требует большего числа ресурсов, чем им заявлено;
- суммарное число уже распределенных всем процессам ресурсов не превосходит общего числа ресурсов в системе.

Ситуация называется безопасной, если процессы можно выстроить в такую последовательность, при которой:

- первый процесс может нормально завершиться, даже если он полностью выберет ресурсы по своей заявке;
- второй процесс может нормально завершиться, даже если он полностью выберет ресурсы по своей заявке, при условии, что завершится первый процесс и освободит удерживаемые им ресурсы;

- $i$ -й процесс может нормально завершиться, даже если он полностью выберет ресурсы по своей заявке, при условии, что завершится  $i-1$ -й процесс и освободит удерживаемые им ресурсы.

В противном случае ситуация называется опасной.

Если ситуация безопасна, то при отсутствии новых процессов все уже имеющиеся процессы могут нормально завершиться, выбрав ресурсы в соответствии со своими заявками – тупик невозможен.

Пример представлен на рисунке 5.3. Пусть у ОС имеется всего 12 ресурсов одного класса и на текущий момент выполняются три процесса – P1, P2 и P3, их заявки составляют 12, 4 и 8 ресурсов соответственно. На текущий момент времени процессу P1 выделено 4 ресурса, процессу P2 – 2, процессу P3 – 4 (см. рисунок 5.3.а). В резерве ОС остаются, таким образом, 2 ресурса. Ситуация рисунка 5.2.а является безопасной. ОС может выделить оставшиеся два ресурса процессу P2 и этот процесс, полностью получив по своей заявке, может завершиться, тогда системный резерв увеличится до 4 ресурсов. Этот резерв ОС отдаст процессу P3, он, завершившись, передаст в резерв ОС 8 ресурсов, которых будет достаточно для удовлетворения заявки процесса P1. Ситуация станет опасной, если ОС выделит ресурс какому-либо другому процессу, кроме P2 (см. рисунок 5.3.б). В этой ситуации ОС за счет своего резерва не может полностью удовлетворить ни одну заявку.

Процесс	Заявлено	Выделено
P1	12	4
P2	4	2
P3	8	4
Системный резерв		2

а) безопасная

Процесс	Заявлено	Выделено
P1	12	5
P2	4	2
P3	8	4
Системный резерв		1

б) опасная

Рисунок 5.3 Анализ ситуации

Стратегия алгоритма банкира состоит в том, что запрос на ресурс удовлетворяется только в том случае, если выделение его не сделает ситуацию опасной. Алгоритмическая реализация проверки ситуации на безопасность состоит в "проигрывании" на списке процессов приведенного выше определения безопасной ситуации:

1. Создать список, элементами которого являются процессы с их ресурсами и заявками.
2. Если список пуст, установить флаг безопасной ситуации и перейти к п. 7.
3. Найти в списке процесс, заявка которого может быть удовлетворена из системного резерва.
4. Если такой процесс не найден – установить флаг опасной ситуации и перейти к п.7.
5. Удалить найденный процесс из списка и передать те ресурсы, которыми он обладал в системный резерв.
6. Перейти к п.2.
7. Закончить проверку.

### Алгоритм Габермана

Другая алгоритмическая реализация стратегии с предварительными заявками известна как алгоритм Габермана. Этот алгоритм иллюстрируется следующим программным кодом на языке С:

```

/* Алгоритм Габермана */
static int S[r];
/* Начальные установки */
void HabervanInit(void) {
    int i;
    for (i=0; i<r; S[i++]=0);
}
/* Выделение ресурса */
int HabermanGet(int claim, int hold) {
    int i;
    for (i=0; i<claim-hold; i++)
        if (!S[i]) return -1; /* отказ */
    for (i=0; i<claim-hold; S[i++]--);
return 0; /* подтверждение */
}
/* Освобождение ресурса */
void HabermanRelease(int claim, int hold) {
    int i;
    for (i=0; i<=claim-hold; S[i++]++);
}

```

Если в системе имеется  $r$  ресурсов одного класса, то ОС создает массив  $S$  из  $r$  целых чисел. В ходе начальных установок в массив записывается убывающая последовательность чисел от  $r$  до 1. Если процесс, имеющий заявку на  $claim$  ресурсов и уже получивший  $hold$  ресурсов, запрашивает еще один ресурс, то (функция `HabervanGet`) все элементы массива  $S$  с номерами от 0 до  $claim-hold-1$  включительно уменьшаются на 1. Индикатором опасного состояния является отрицательное значение любого элемента массива  $S$ .

На рисунке 5.4 показана работа алгоритма Габермана для ситуации, рассмотренной нами в примере выше. В графе "Событие" таблицы на рисунке 5.4 указывается идентификатор процесса, которому выделяется единица ресурса; остальная часть каждой строки представляет состояние массива  $S$  после этого выделения. Строка с пустой графой "Событие" показывает исходное состояние массива  $S$ , строка "итог" – состояние на момент, когда ресурсы распределены так, как показано на рисунке 5.3.а. (Убедитесь сами, что на конечное состояние массива  $S$  не влияет последовательность, в которой ресурсы выделялись.) Три нижние строки, в которых идентификатору процесса предшествует символ ' \* ', показывают состояние массива  $S$  для альтернативных вариантов – выделения еще одного ресурса процессу P1 или P2, или P3. Видно, что только процесс P2 может получать ресурсы, не переводя систему в опасное состояние.

Событие	0	1	2	3	4	5	6	7	8	9	10	11
	12	11	10	9	8	7	6	5	4	3	2	1
P1	11	10	9	8	7	6	5	4	3	2	1	0
P1	10	9	8	7	6	5	4	3	2	1	0	0
P1	9	8	7	6	5	4	3	2	1	0	0	0
P1	8	7	6	5	4	3	2	1	0	0	0	0
P2	7	6	5	4	4	3	2	1	0	0	0	0
P2	6	5	4	4	4	3	2	1	0	0	0	0
P3	5	4	3	3	3	2	1	0	0	0	0	0
P3	4	3	2	2	2	1	0	0	0	0	0	0
P3	3	2	1	1	1	0	0	0	0	0	0	0
P3	2	1	0	0	0	0	0	0	0	0	0	0
итог	2	1	0	0	0	0	0	0	0	0	0	0
*P1	1	0	-1	-1	-1	-1	-1	-1	-1	0	0	0
*P2	1	0	0	0	0	0	0	0	0	0	0	0
*P3	1	0	-1	-1	-1	0	0	0	0	0	0	0

Рисунок 5.4 Алгоритм Габермана

К сожалению, простая реализация алгоритма Габермана возможна только для одного класса ресурсов. При наличии в системе нескольких классов безопасность во всех классах по отдельности не гарантирует безопасности всей системы в целом. Так, в примере про процессы А и В, использующие принтер и ленту, ситуация является безопасной как по



классу принтеров, так и по классу лент, но в комплексе она является тупиковой. Алгоритм "проигрывания ситуации" требует значительного объема вычислений, но обеспечивает комплексный анализ безопасности.

Отметим, что алгоритм банкира, хотя и является значительно более либеральным, чем все рассмотренные выше, тоже не обеспечивает оптимального уровня мультипрограммирования. Опасная ситуация еще не является тупиковой. Так, ситуация, представленная на рисунке 5.3.б, еще может разрядиться, если процесс P1 освободит хотя бы один из удерживаемых им ресурсов. ОС же в оценке ситуации исходит из прогноза о наихудшем развитии ситуации – предположения о том, что процессы будут только запрашивать ресурсы, а не освобождать их.

Возможны и стратегии, обеспечивающие еще более либеральную политику, но они требуют большей предварительной информации о процессе и, как правило, – большего объема вычислений при реализации стратегии. Например, нетрудно представить себе, что если ОС заранее знает, в какой последовательности процесс будет запрашивать/освобождать ресурсы, то она может обеспечить более эффективное управление ими. Но часто ли такая последовательность может быть известна заранее? Среди авторов [12, 26, 30 и др.] нет даже единодушного мнения о том, реально ли требовать от процесса предварительной заявки. На наш взгляд, однако, такое требование в большинстве случаев реально выполнимо: для пакетных процессов потребность в ресурсах бывает известна заранее, интерактивные же процессы в многопользовательских системах запускаются только зарегистрированными пользователями, указанный при регистрации пользователя доступный ему объем ресурсов может считаться такой заявкой.

Двигаясь дальше в сторону либерализации, мы пересекаем ту границу, до которой можно было предотвратить тупики. Теперь тупики

возможны. И теперь ведущее значение приобретают методы обнаружения тупиков.

### Методы обнаружения тупиков

Определение нетупиковой ситуации очень похоже на определение безопасной ситуации, но отличается тем, что здесь рассматриваются не потенциальные запросы – заявки, а текущие – уже сделанные, но еще не удовлетворенные запросы. Ситуация называется нетупиковой, если процессы можно выстроить в такую последовательность, при которой:

- первый процесс может получить ресурсы по всем своим текущим запросам;
- второй процесс может получить ресурсы по всем своим текущим запросам при условии, что первый процесс закончился и освободил все выделенные ему ресурсы;
- $i$ -й процесс может получить ресурсы по всем своим текущим запросам при условии, что  $i-1$ -й процесс закончился и освободил все выделенные ему ресурсы.

В противном случае ситуация является тупиковой.

Если анализ опасной ситуации исходит из предположения о худшем развитии событий, то анализ тупиковой – из предположения о лучшем их развитии – что процессы больше не будут запрашивать ресурсы, а будут только их освобождать.

Пусть в системе имеется 11 ресурсов одного класса, как представлено на Рисунке 5.5. Процесс P1 уже имеет 2 ресурса и запрашивает еще 2, процесс P2 уже имеет 3 и запрашивает еще 5, процесс P3 имеет 4 и запрашивает еще 4. Системный резерв – 2 ресурса. В зависимости от того, какие у процессов максимальные заявки (нам это неизвестно), ситуация может быть расценена как опасная или безопасная,

но она определенно не является тупиковой. Процесс P1 может получить требуемые ресурсы, если он после этого закончится, то может быть удовлетворен запрос процесса P3, а после его завершения – запрос процесса P2. Но если мы выделим еще хотя бы один ресурс процессу P3, ситуация станет тупиковой.

Процесс	Выделено	Запрошено
P1	2	2
P2	3	5
P3	4	4
Системный резерв		2

а) нетупиковая

Процесс	Заявлено	Выделено
P1	12	5
P2	4	2
P3	8	4
Системный резерв		1

б) тупиковая

Рисунок 5.5 Анализ ситуации

При исследовании проблемы тупиков удобно представлять систему в виде графа. Пример такого графа показан на рисунке 5.6.

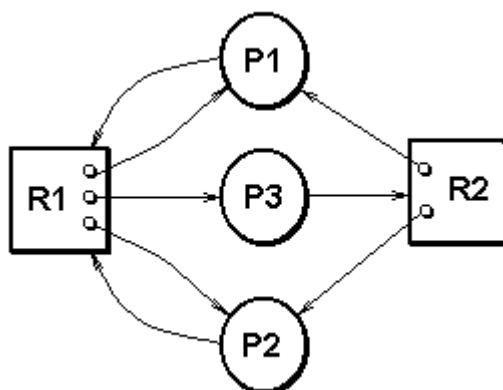


Рисунок 5.6 Граф ресурсов и процессов

Графы такого рода содержат вершины двух типов: процессы (показаны окружностями) и классы ресурсов (показаны прямоугольниками), в последних указывается число ресурсов в классе. Дуги графа могут соединять только разнотипные вершины. Направленность дуг означает: от ресурса к процессу – ресурс выделен данному процессу, от процесса к ресурсу – процесс запрашивает ресурс. Признаком тупика является наличие в графе петли – такого пути, который начинается и заканчивается в одной вершине, и из которого нет выхода. Так, на рисунке 5.5 представлена тупиковая ситуация: следование по графу любыми возможными путями заставит нас проходить одни и те же вершины. Если же мы уберем дугу, ведущую от процесса P3 к ресурсу R2, то появится выход из петли в вершину P3. Обнаружение тупика сводится к попытке редукции (сокращения) графа. Из графа удаляются те вершины-процессы, которые не имеют запросов или запросы которых могут быть удовлетворены. При этом сокращаются также и все дуги, связанные с этими вершинами. За счет освободившихся ресурсов появляется возможность сократить новые вершины-процессы и т.д. Если в графе нет петель, то после многократных сокращений нам удастся сократить все вершины-процессы.

Как часто следует выполнять проверку тупика? Обратим внимание на то, что если конструктор ОС отказывается от предупреждения тупиков, то он делает это в значительной степени из нежелания загружать систему значительным объемом вычислений по анализу безопасности при каждом запросе. Обнаружение тупика выполняется сходным образом с анализом безопасной ситуации и требует такого же объема вычислений, следовательно, желательно выполнять его как можно реже. Единственным событием, которое может перевести систему из нетупикового состояния в тупиковое, является поступление запроса, который не может быть удовлетворен. Следовательно, попытку обнаружения тупика надо

производить только по этому событию, никак не чаще. В некоторых реализациях ОС применяет еще более ленивую политику. Неудовлетворенный запрос может привести к тупику, но может и не привести. "Ленивая" ОС не спешит выполнять проверку даже при поступлении такого запроса, а выжидает некоторое время: может быть "все само собой уладится" – и в большинстве случаев именно так и случается. И только если запрос остается неудовлетворенным в течение определенного времени, ОС принимается за поиск тупика. Размер временной выдержки может определяться скоростными характеристиками запрашиваемого ресурса.

Если тупик обнаружен, то как его ликвидировать? К сожалению, развязка тупика практически всегда связана с потерями. Единственным реальным способом развязки является принудительное прекращение одного или нескольких процессов и освобождение удерживаемых ими ресурсов. Как выбрать жертву для прекращения? Во-первых, ОС, конечно, должна быть уверена в том, что при прекращении выбранных процессов освободится объем ресурсов, достаточный для развязки тупика. Во-вторых, оценивается объем потерь, связанных с прекращением того или иного процесса. Прекращенный процесс, скорее всего, будет запущен повторно, таким образом, ресурсы, использованные им при его незаконченном выполнении, составляют прямые потери. Поэтому естественным решением представляется прекращение того процесса, который к этому моменту использовал меньше всего ресурсов (не только монопольных, но любых ресурсов вообще). Поскольку самым дорогостоящим ресурсом обычно является процессорное время, выбор жертвы по критерию минимального использованного времени производится наиболее часто.

Желательно, чтобы "пострадавший" процесс был снова запущен, причем, возможно даже, с повышенным приоритетом. Но всякий ли

процесс можно прервать на середине, а потом запустить сначала? Представьте себе процесс-программу, которая должна начислить взносы на 10 банковских счетов. Эта программа принудительно завершается в тот момент, когда она успела обработать только 5 счетов. Если при перезапуске программа начнет выполняться с начала, она повторно начислит взносы на первые 5 счетов. ОС не может знать, приведет ли перезапуск процесса к нежелательным последствиям, поэтому решение о повторном запуске обычно перекладывается на пользователя.

## **5.4. Бесконечное откладывание**

Процессы, ожидающие ресурсов, встают в очереди к этим ресурсам. Такая очередь может обслуживаться любой невытесняющей стратегией планирования. Моментом, когда менеджер ресурса принимает решение об обслуживании, является освобождение ресурса.

Единственной дисциплиной, которая гарантирует, что все процессы в очереди будут в конце концов обслужены, является FCFS. Другие методы базируются на приоритетах – внешних или вычисляемых на основании размера запроса или/и безопасности ситуации. Очередь может быть упорядочена по возрастанию размера запроса и при освобождении ресурсов последние могут отдаваться по запросу "самого подходящего" размера. Очередь может быть упорядочена в порядке возрастания опасности выделения ресурсов, и освободившийся ресурс может отдаваться тому процессу, запрос которого самый безопасный. При упорядочении очереди процесс, выдавший самый большой или самый опасный запрос, может надолго в ней застрять, то есть попасть в ситуацию бесконечного откладывания. С другой стороны, применение FCFS в чистом виде может привести к снижению уровня мультипрограммирования, к опасному состоянию и даже к тупику – если

процесс со слишком большим запросом окажется первым в очереди. Отчасти это противоречие может быть сглажено, если мы допустим частичное выделение ресурсов: запрос, стоящий в очереди первым получает столько ресурсов, сколько ему можно выделить, сохраняя ситуацию безопасной, остальные – отдаются следующему в очереди процессу.

Поскольку тупиковая ситуация более опасна, чем бесконечное откладывание, ОС все же отдает предпочтение критериям безопасности, следовательно, заведомо предотвратить бесконечное откладывание невозможно. Бесконечное откладывание процесса устанавливается по времени его пребывания в очереди: если процесс пребывает в очереди дольше некоторого установленного времени, то считается, что он ожидает бесконечно. В зависимости от политики ОС в отношении справедливости обслуживания и от характеристик процесса (если они известны) допустимое время ожидания может устанавливаться большим или меньшим. Если же бесконечное откладывание установлено, то для его ликвидации ОС приостанавливает выдачу ресурсов новым процессам, пока не будет обслужен отложенный процесс.

Проблема тупиков до некоторой степени теряет актуальность в современных ОС, так как они имеют тенденцию к уменьшению количества неразделяемых ресурсов. Одним из способов сделать неразделяемое устройство разделяемым является буферизация, которую мы рассмотрим в следующей главе. Системные структуры данных разделяются с использованием средств взаимного исключения доступа, которым будет посвящена глава 8. Эта проблема, однако, становится все более актуальной для современных СУБД, которые обеспечивают одновременный доступ к ресурсам-данным для тысяч и десятков тысяч пользователей.

## **КОНТРОЛЬНЫЕ ВОПРОСЫ**

1. Приведите примеры ресурсов: разделяемых и монопольно используемых, непрерывных и дискретных, потребляемых и повторно используемых.
2. Какую информацию о ресурсах должна хранить ОС?
3. Почему иерархическое выделение ресурсов является весьма популярным во многих ОС?
4. В чем состоят ограничения на возможность применения алгоритма банкира?
5. В чем сходство безопасной и беступиковой ситуаций? В чем их различия?
6. В какие моменты ОС должна проводить действия по обнаружению тупика?
7. Какие соображения могут влиять на выбор процесса-жертвы при ликвидации тупика?
8. В чем достоинство дисциплины обслуживания FCFS для очередей к монопольно используемым ресурсам?

## **Глава 6. Управление вводом-выводом**

### **6.1. Виртуализация устройств и структура драйвера**

Управление вводом-выводом в полной мере воплощает в себе определение "ОС снаружи": ОС конструирует ресурсы высокого уровня – виртуальные устройства – и предоставляет пользователю интерфейс для работы с ними. Программисты, начинавшие работу в среде MS DOS, привыкли к доступности средств прямого управления вводом-выводом для любой программы, но в многозадачных ОС о такой доступности для



прикладной программы может идти речь только в исключительных случаях, а в многопользовательских ОС она исключается вообще.

Можно в общем случае определить четыре метода, которые могут использоваться ОС для конструирования виртуальных устройств (виртуализации):

- метод закрепления или выделения (allocation);
- метод разделения (sharing);
- метод накопления или спулинга (spooling);
- метод моделирования (simulation).

Одна и та же ОС может использовать разные методы виртуализации для разных устройств.

Метод закрепления однозначно отображает виртуальное устройство на реальное устройство. Метод закрепления наименее эффективен, так как закрепляемое устройство является монопольно используемым ресурсом, и применение этого метода порождает все проблемы, связанные с использованием таких ресурсов.

Метод разделения применим к устройством, ресурс которых является делимым. В этом случае ресурс устройства разбивается на части, каждая из которых закрепляется за одним процессом. Примером применения метода могут служить минидиски в ОС VM/370 [28]: все пространство диска разбивается на участки, каждый из которых выглядит для процесса как отдельный том. Можно, например, разделять между процессами и экран видеотерминала. Зафиксированная часть устройства является также монопольным ресурсом и разделение лишь частично снимает остроту проблем управления таким ресурсом. Возможность дробления устройства предполагает внутреннюю адресацию в устройстве (адрес на диске, адрес в видеопамяти). По аналогии с адресацией в памяти процесс и здесь работает с виртуальными адресами в виртуальном устройстве, а ОС транслирует их в реальные адреса в реальном устройстве.

Метод спулинга заставляет процесс обмениваться данными не с реальным устройством, а с некоторой буферной областью в памяти (оперативной или внешней). Обмен же данными между буфером и реальным устройством организует сама ОС, причем, как правило, с упреждением (при вводе) или с запаздыванием (при выводе). Буферизация прозрачна для процессов и может создавать у них иллюзию одновременного использования устройства, если каждому процессу выделен свой буфер. Примером могут служить спулинг печати, применяемый во всех современных ОС.

Метод моделирования не связан с реальными устройствами вообще. Устройство моделируется ОС чисто программными методами. Естественным применением этого метода является отработка приемов работы с устройствами, отсутствующими в конфигурации данной вычислительной системы. Часто ОС удобно представлять некоторые свои ресурсы как метафоры (подобия) устройств – это также моделируемые устройства. Так, в VM/ESA обмен данными между виртуальными машинами ведется через виртуальный адаптер, метафорой устройства можно также считать межпрограммный канал (pipe), реализованный во многих современных ОС.

При любом методе виртуализации ОС является "прослойкой" между процессами и реальными устройствами. Эту функцию выполняют входящие в состав ОС драйверы устройств. К драйверам обращаются и другие модули ОС, и процессы пользователя, причем последние, как правило, не непосредственно, а через библиотеки вызовов, предоставляющие более удобный API. В некоторых случаях ОС может предоставить пользователю интерфейс, обладающий высокой степенью подобия с интерфейсом реального устройства, но и в этом случае ОС, даже применяя метод выделения, производит обработку управляющих воздействий, сформированных процессом: проверку правильности команд,

трансляцию адресов памяти, адресов устройств и адресов в устройствах и т.п.

Модули ОС, которые осуществляют трансляцию однотипных для всех устройств обращений к ним из процессов и из других модулей ОС в специфические для устройства управляющие воздействия и управляют выполнением этих воздействий, называются драйверами. (Мы не согласны с теми авторами, которые называют драйверами любые программы управления вводом-выводом, драйверы в нашем понимании обязательно включаются в состав ОС и обязательно соответствуют спецификациям данной ОС.) Каждому типу устройства соответствует свой драйвер. Драйвер устройства имеет два основных уровня, как показано на рисунке 6.1. Первый (верхний) уровень принимает системные вызовы от процессов и формирует на основании каждого вызова запрос. Этот же уровень выстраивает запросы в очередь и поддерживает упорядоченность этой очереди в соответствии с принятой дисциплиной обслуживания. Вторым (нижним) уровнем драйвера выбирает из очереди запрос и обслуживает его: формирует управляющие воздействия и передает их на устройство, обрабатывает прерывания от устройства и сообщает ядру ОС о наступлении событий, связанных с вводом-выводом.

Очевидно, что нижний уровень драйвера должен иметь возможность выполнять привилегированные команды – команды ввода-вывода и т.п. Можно назвать три варианта обеспечения такой возможности. Вариант первый – включение драйверов в ядро ОС. При этом достигается высокая эффективность функционирования драйверов, но изменение состава драйверов требует перекомпоновки ядра. Вторым вариантом – выполнение драйверов в виде отдельных модулей, функционирующих в режиме ядра. Этот вариант позволяет изменять состав драйверов без изменения ядра ОС, но в принципе снижает надежность системы, так как новые модули, функционирующие в режиме ядра, могут являться источником фатальных

ошибок. Наконец, третий вариант – драйверы выполняются в виде модулей режима процесса, но ОС предоставляет для них сервисные системные вызовы, выполняющие для драйверов операции режима ядра. Этот вариант допускает изменение драйверов и не снижает надежность, но является несколько менее эффективным.

Во многих современных ОС применяется не двух-, а многоуровневая структура драйверов. Нижний уровень классического двухуровневого драйвера сохраняется и называется аппаратным драйвером. Функции же верхнего уровня разделяются между драйверами нескольких слоев. Каждый более высокий уровень выполняет более абстрактную обработку данных и в меньшей степени зависит от конкретного устройства. Каждый уровень представляет собой отдельный драйвер и может меняться независимо от других. Многоуровневая структура позволяет минимизировать изменения состава драйверов при изменении состава устройств.

## **6.2. Интерфейсы устройств**

Здесь мы рассматриваем взаимодействия ОС с устройствами, лежащие на уровне "интерфейса оборудования" (см. рисунок 1.2).

При всем многообразии внешних устройств ЭВМ и способов управления ими их программные интерфейсы могут быть сведены к трем основным моделям, определяющимся способом подключения устройств к ЭВМ:

- регистры устройств;
- контроллеры ввода-вывода;
- прямой доступ к памяти;

- каналы ввода-вывода;
- процессоры ввода-вывода.

Устройство может быть подключено к процессору через регистры устройства, как показано на рисунке 6.1. Такое подключение применяется для устройств, которые имеют простое управление, и обмен с ними ведется небольшими порциями данных (байт, слово, двойное слово). Устройство может иметь большое число регистров, которые, однако, сводятся к трем основным типам: регистры состояния – для передачи в процессор информации о состоянии, регистры управления – для передачи на устройство команд, регистры данных – для обмена данными между процессором и устройством. Регистры управления и состояния, как правило, являются однонаправленными, регистры данных могут быть как одно-, так и двунаправленными. Регистры устройств являются расширением адресного пространства ЭВМ. Расширение это может быть как явным – с доступом при помощи команд работы с памятью типа MOV, так и неявным – с отдельной адресацией портов ввода-вывода и доступом при помощи специальных команд типа IN/OUT.

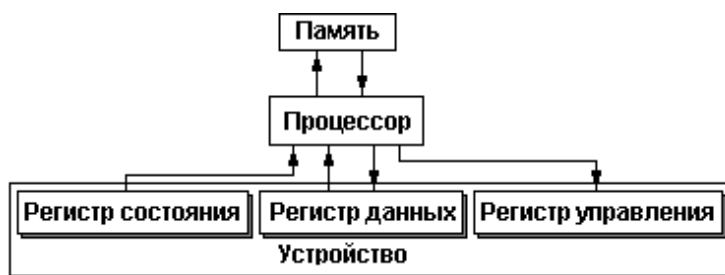


Рисунок 6.1 Прямое подключение устройства

Сколько-нибудь сложные по управлению устройства подсоединяются к ЭВМ через контроллеры ввода-вывода (устройства управления), причем один контроллер может обслуживать несколько однотипных устройств, как показано на рисунке 6.2. С точки зрения

программного интерфейса это подключение ничем не отличается от предыдущего варианта, регистры контроллера выглядят для программы так же, как и регистры устройств.

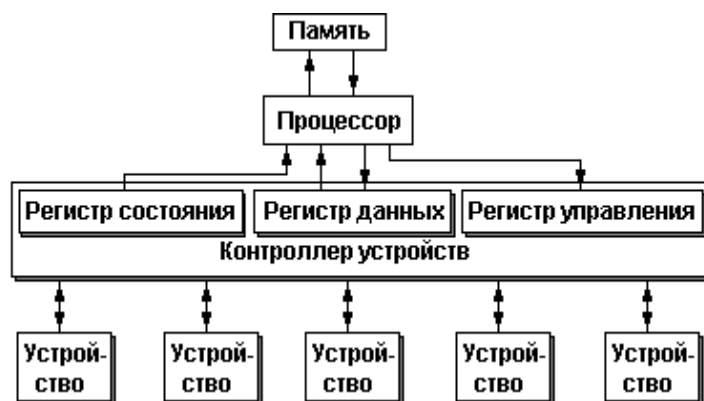


Рисунок 6.2 Подключение через контроллер

Быстродействие устройств много ниже быстродействия центрального процессора, поэтому обычно после выдачи команды на устройство программа должна дожидаться ее завершения. Программа может убедиться в завершении операции одним из двух способов: опросом или прерыванием. Опрос предполагает периодическое чтение регистра состояния устройства и проверку в нем признака завершения операции. Крайним случаем опроса является занятое ожидание – когда программа опрашивает устройство практически непрерывно, ничем другим не занимаясь. Помимо того, что при этом непроизводительно расходуется процессорное время, занятое ожидание еще и небезопасно: при отсутствии сигнала окончания от устройства (например, при сбое последнего) программа может "зависнуть" в состоянии занятого ожидания, а если она при этом была в непрерываемом состоянии – заблокировать работу всей системы.

Более действенным способом сигнализации об окончании операции является прерывание от устройства. При большом разнообразии

механизмов прерываний в разных архитектурах вычислительных систем все они обеспечивают сохранение вектора состояния прерванного процесса и идентификацию устройства, приславшего прерывание. При использовании прерываний ОС после выдачи команды ввода-вывода на устройство переводит процесс в состояние ожидания. Прерывание, присланное устройством, обрабатывается ядром ОС, которое при этом разблокирует процесс, ожидающий завершения операции.

Для устройств, обмен с которыми ведется большими порциями информации, применяется прямой доступ к памяти (ПДП), показанный на рисунке 6.3. Контроллер ПДП работает параллельно с центральным процессором и обменивается данными прямо с оперативной памятью, минуя центральный процессор. Сам контроллер ПДП выглядит для программы как устройство с доступом через регистры. Программа должна его запрограммировать, записав в его регистры адрес области оперативной памяти, с которой происходит обмен и размер блока данных, а затем запустить операцию, которая инициирует прямой обмен. Регистр данных контроллера ПДП при этом используется только для передачи управляющей информации. Об окончании обмена программа может узнать либо по прерыванию, либо опрашивая регистр состояния контроллера. Контроллер ПДП обычно содержит собственную буферную память для сглаживания разницы в быстродействии устройства и оперативной памяти. Напомним, что аппаратура ПДП обычно не обеспечивает динамическую трансляцию адресов. Поэтому ОС, получив от процесса запрос на выполнение операции ввода-вывода через ПДП, фиксирует в реальной памяти ту часть виртуального адресного пространства программы, с которой происходит обмен, – до окончания обмена. Отметим также, что непрерывное виртуальное адресное пространство процесса может отображаться в несмежные страничные кадры реальной памяти, поэтому

ОС может вводить-выводить непрерывный с точки зрения процесса блок данных за несколько операций обмена.

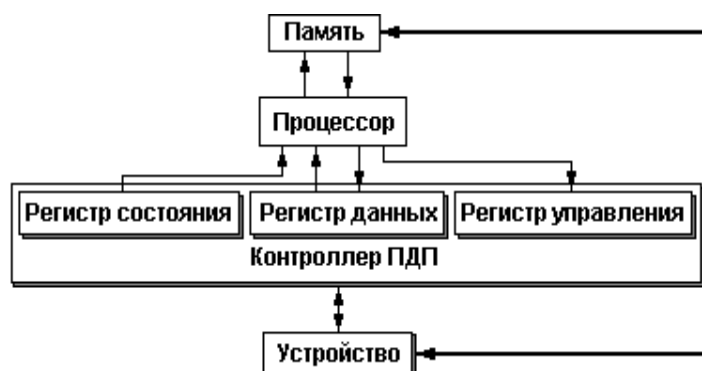


Рисунок 6.3 Подключение через ПДП

В сущности, и контроллер обычного устройства, и контроллер ПДП представляют собой специализированные процессорные устройства, но более полно это качество присуще каналу ввода-вывода. Каналы представляют собой специализированные процессоры, имеющие свою систему команд и работающие параллельно с центральным процессором, но использующие ту же оперативную память. В отличие от контроллеров, которые являются специализированными по типам устройств, каналы являются универсальными процессорами ввода-вывода, к одному каналу могут быть одновременно подсоединены контроллеры разных устройств. Работа канала во многом похожа на работу контроллера ПДП: канал программируется, а затем запускается операция, в ходе которой канал обеспечивает прямой обмен с оперативной памятью, минуя центральный процессор. Подключение через канал показано на рисунке 6.4.



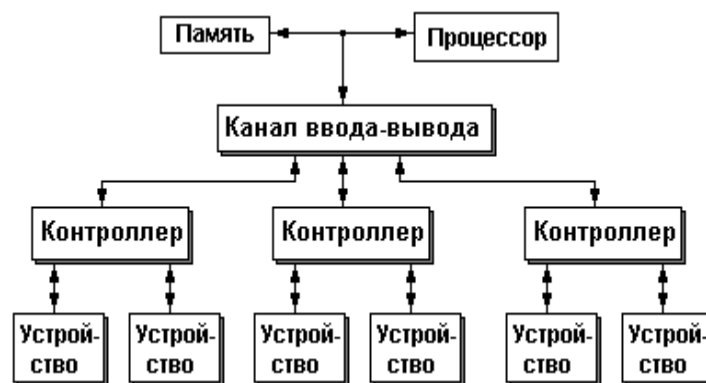


Рисунок 6.4 Подключение через канал ввода-вывода

Идея канала ввода-вывода, впервые реализованная в System/360 фирмы IBM, была впоследствии воплощена в ряде других архитектур. Ввиду своей продуктивности эта идея без концептуальных изменений была перенесена и в System/370, и в System/390, и отказа от нее в перспективе также не предвидится. В последующем изложении мы опираемся именно на эти реализации каналов ввода-вывода.

Существует несколько типов каналов (селекторный, мультиплексный, байт-мультиплексный), предназначенных для подключения устройств с разными скоростями обмена, но для программиста все они выглядят одинаково. Средства программирования канала, во-первых, гораздо более мощные и гибкие, чем контроллера ПДП, во-вторых, позволяют унифицировать программирование ввода-вывода для различных устройств.

Выполнение операции ввода-вывода подразумевает совместное (параллельное или квазипараллельное) функционирование нескольких "субъектов": основной программы, выполняющейся на центральном процессоре (далее – программа ЦП), канала и устройства (далее – канал), аппаратного механизма прерываний и программы обработки прерываний. Программа ЦП формирует в оперативной памяти программу канала, сообщает системе ввода-вывода ее адрес, назначает Блок управления событием, в котором будет сделана отметка о завершении операции, и

выдает команду "Начать ввод-вывод", адресуя канал и устройство. Канал проверяет готовность канала/устройства и начинает выполнение канальной программы. При этом команда "Начать ввод-вывод" завершается, и программа ЦП продолжает свое выполнение. Когда у программы ЦП возникает необходимость дождаться завершения операции, она опрашивает Блок управления событием. Если в нем есть отметка о выполнении, программа продолжает выполняться, в противном случае – переводится в состояние ожидания до появления отметки в Блоке управления событием. Канал при завершении операции сообщает системе ввода-вывода информацию о своем состоянии и инициирует прерывание по вводу-выводу. Аппаратный механизм прерывания сохраняет текущее состояние программы и обеспечивает передачу управления на программу обработки прерываний по вводу-выводу. Программа обработки прерываний распознает канал и устройство, пославшие прерывания, и передает управление на соответствующую ветвь обработки. Из информации о состоянии канала определяется причина прерывания. Выполняются действия по обработке соответствующей ситуации. Диагностическая информация может быть также записана в область памяти, доступную для программы ЦП. Если прерывание сообщает об окончании операции, обработчик прерывания делает отметку в Блоке управления событием. Обработчик прерывания возвращает управление в прерванную программу. Программа ЦП, если это необходимо, анализирует диагностическую информацию о результатах выполнения операции.

Программа канала размещается в оперативной памяти и представляет собой массив канальных команд. Каждая команда канальной программы – структура данных фиксированной длины, содержащая код операции, адрес области памяти, с которой происходит обмен, код контроля доступа к памяти, признаки режима выполнения, объем передаваемых или принимаемых данных.

Различаются несколько типов операций: "чтение" (передача данных из устройства в память), "запись" (передача данных из памяти в устройство), "управление" (выполнение специфических операций на устройстве, например, перемотка магнитной ленты) – эти команды специфичны для устройств, устройство может иметь несколько модификаций одной команды. Общей для всех устройств является команда "уточнить состояние" – передача в память информации о состоянии устройства. Команда "переход в канале" на устройство не передается, она изменяет последовательность выполнения канальных команд. Код контроля доступа к памяти позволяет предотвратить вторжение процесса в операциях ввода-вывода в области памяти, принадлежащие другим процессам или ОС.

Среди признаков режима выполнения наибольший интерес представляют три. Признак "цепочка команд" определяет продолжение программы канала в следующей команде. Признак "цепочка данных" задает выполнение следующей канальной команды как продолжение текущей: поле кода операции в ней игнорируется, но все остальные поля обновляются. Если отсутствуют признаки "цепочка команд" или "цепочка данных", канальная команда считается последней в программе. Признак "программно управляемое прерывание" задает генерацию прерывания при выборке каналом команды, содержащей этот признак, что дает возможность синхронизировать работу программы на центральном процессоре с выполнением канальной программы.

Взаимодействие между субъектами выполнения операции ввода-вывода происходит также через фиксированные адреса памяти, в которые записывается управляющая и диагностическая информация: Адресное слово канала, Слово состояния канала, Слово состояния программы.

Как и контроллер ПДП, канал не выполняет динамическую трансляцию адресов, кроме того, в системах, где процесс работает только в

своей виртуальной памяти, не имея дела с реальными адресами, ОС транслирует программу канала, размещая результат трансляции в недоступной для процесса области. В этом случае процесс не может модифицировать программу канала в ходе ее выполнения.

Контроллер ПДП и канал ввода-вывода являются специализированными процессорами. Следующим шагом в интеллектуализации контроллеров ввода-вывода являются процессоры ввода-вывода – универсальные процессоры, выполняющие функции контроллера ввода-вывода. Впервые процессоры ввода-вывода были применены в вычислительной системе CDC-6600, считающейся первым суперкомпьютером. С тех пор процессоры ввода-вывода перестали быть прерогативой суперсистем и постепенно внедряются в системы ординарные (например, AS/400). Интерфейс процессора ввода-вывода не похож на интерфейс устройства в обычном понимании. Взаимодействие ОС с процессором ввода-вывода происходит через механизм обмена сообщениями, поддерживаемый микроядром. Данные могут передаваться как в составе сообщения, так и выбираться процессором ввода-вывода непосредственно из оперативной памяти. Взаимодействие через сообщения, естественно, занимает больше времени, чем управление вводом-выводом через контроллер или канал, но выигрыш в эффективности получается за счет переноса некоторых (иногда весьма значительных по объему) операций обработки данных в процессор ввода-вывода. Являясь универсальным процессором, процессор ввода-вывода работает под управлением своей собственной мини-ОС и программа управления устройством имеет статус приложения в этой ОС.

### **6.3. Управление устройствами**

Внешние устройства ЭВМ отличаются большим разнообразием в форматах управляющей информации и в алгоритмах управления. Даже в канальной модели интерфейса разные устройства имеют разные модификации команд чтения/записи/управления, не говоря уже о том, что транзакция на устройстве обычно подразумевает выполнение нескольких команд и последовательности их бывают совершенно различными для разных устройств.

Модули ОС, которые осуществляют трансляцию однотипных для всех устройств обращений к ним из процессов и из других модулей ОС в специфические для устройства управляющие воздействия и управляют выполнением этих воздействий, называются драйверами. (Мы не согласны с теми авторами, которые называют драйверами любые программы управления вводом-выводом, драйверы в нашем понимании обязательно включаются в состав ОС и обязательно соответствуют спецификациям данной ОС.) Каждому типу устройства соответствует свой драйвер. Драйвер устройства имеет два основных уровня, как показано на рис.6.5. Первый (верхний) уровень принимает системные вызовы от процессов и формирует на основании каждого вызова запрос. Этот же уровень выстраивает запросы в очередь и поддерживает упорядоченность этой очереди в соответствии с принятой дисциплиной обслуживания. Второй (нижний) уровень драйвера выбирает из очереди первый запрос и обслуживает его: формирует управляющие воздействия и передает их на устройство, обрабатывает прерывания от устройства и сообщает ядру ОС о наступлении событий, связанных с вводом-выводом.



Рисунок 6.5 Структура драйвера

Как мы сказали, верхний уровень драйвера определяет очередность, в которой обслуживаются запросы от разных процессов. Для реализации политики обслуживания драйвер должен учитывать как приоритеты процессов, так и доступность устройств. Приоритетность запросов может оцениваться по разным стратегиям, из которых наиболее распространенной является та, согласно которой запрос процесса, имеющего наивысший процессорный приоритет, имеет наивысший приоритет в очереди драйвера и другие стратегии, ставящие целью повышение эффективности обмена с устройством, пример такой стратегии мы рассматриваем ниже применительно к драйверам дисковых накопителей.

Доступность устройства мы рассмотрим на примере канальной модели подключения. Путь от процессора к устройству включает в себя три "станции": канал, контроллер, устройство. Каждая из станций пути может быть свободна или занята независимо от других. На приведенной на рисунке 6.4 древовидной структуре различные уровни этой структуры характеризуются различным временем своего участия в операции ввода-вывода. Канал участвует только в передаче данных. Например, при выводе канал может быть занятым только на время передачи из памяти в буфер контроллера, после чего канал освобождается и может обслуживать другой контроллер, а первый контроллер тем временем передает данные на устройство. После передачи данных на устройство освобождается

контроллер, а устройству может потребоваться еще некоторое время, чтобы обработать полученные данные. Поэтому события освобождения канала, контроллера и устройства индицируются разными признаками в состоянии канала. В наборе команд ввода-вывода есть отдельные команды для проверки состояния устройства, контроллера и канала. Адрес устройства в схеме подключения, подобной той, которая представлена на рисунке 6.4, адрес должен состоять из идентификатора (номера) канала, идентификатора контроллера в канале, идентификатора устройства в контроллере. Процесс обращается к устройству по некоторому своему идентификатору – виртуальному адресу, который может быть подобен реальному, а может представлять собой и логическое имя устройства. В простейшем случае трансляция адреса устройства производится по таблице перекодировки. Хотя, пока в современных системах предпочтение отдается именно древовидной структуре подключения, возможна и более сложная структура, допускающая подключение устройства к нескольким контроллерам, а контроллера – к нескольким каналам. Реальный адрес устройства может формироваться, таким образом, динамически. В IBM System/390 эти функции переданы аппаратной подсистеме ввода-вывода.

Для принятия решений о доступности устройств ОС поддерживает таблицы дескрипторов, отражающие состояние станций пути (три таблицы – по числу типов станций). Для канала дескриптор включает в себя: идентификатор канала; состояние (занят/свободен); список контроллеров, подключенных к каналу; список запросов к каналу. Для контроллера: идентификатор контроллера; состояние; список каналов, к которым подключен контроллер; список устройств, подключенных к контроллеру; список запросов к контроллеру. Для устройства: идентификатор устройства; состояние; список контроллеров, к которым подключено устройство; список запросов к устройству.

Логически являясь частью ОС, драйверы, тем не менее, оформляются как отдельные модули. Поскольку каждый драйвер однозначно связан с устройством определенного типа (а возможно, и данной модификации), то и состав набора драйверов зависит от конфигурации аппаратных средств. Кроме того, обязательно должна быть обеспечена возможность подключения к системе новых внешних устройств без внесения изменений в ОС. При модульности драйверов это достигается простым добавлением нового драйвера к системному программному обеспечению. Драйверы загружаются в память либо при загрузке системы, либо (реже) – динамически, при возникновении потребности в них. Выбор драйверов для загрузки выполняется либо по явным указаниям в процедуре инициализации ОС, либо неявно – по имеющимся таблицам конфигурации системы либо полностью автоматически – путем опроса при загрузке всех установленных устройств, опознания их и подключения соответствующих драйверов.

## **6.4. Примеры драйверов устройств**

Приводимые ниже примеры показывают, что на драйверы некоторых устройств часто возлагаются дополнительные функции помимо непосредственного управления вводом-выводом.

### **Драйвер системных часов**

Вычислительные системы имеют один или два таймера. Обязательным является линейный таймер, генерирующий прерывания центрального процессора через фиксированные интервалы времени. Возможен также программируемый таймер, работающий независимо от линейного и генерирующий однократное прерывание через заданный интервал времени от момента задания. Прерывания такого таймера иногда



называются сигналом тревоги. Если программируемый таймер отсутствует, он может моделироваться при помощи интервального таймера и программных средств.

Драйвер линейного таймера осуществляет обработку его прерываний и в типовом случае может выполнять следующие действия по каждому прерыванию:

- модифицировать системные структуры данных службы времени и даты;
- увеличивать счетчик виртуального времени активного процесса;
- если планирование процессов ведется с квантованием времени, уменьшать счетчик кванта активного процесса и, если счетчик обратился в ноль, вызывать планировщик;
- если не используется программируемый таймер – уменьшать счетчик тревоги и, если он обратился в ноль, вызывать системную задачу, ожидающую этого сигнала. Линейный таймер может поддерживать целый список таких сигналов тревоги, используемых разными процессами и системными службами, временные выдержки могут задаваться для обеспечения протоколов обмена, измерения производительности системы и т.п.

### Драйвер клавиатуры

Этот драйвер предназначен для ввода символов с клавиатуры терминала. В большинстве аппаратных архитектур нажатие любой клавиши на клавиатуре вызывает прерывание. Обработчик этого прерывания в типовом случае выполняет:

- чтение кода клавиши и перевод его в код символа;
- запоминание кодов символов в своем буфере;

- распознавание специальных клавиш или/и комбинаций клавиш (например, Ctrl+Break) и вызов специальных их обработчиков;
- обработку специальных клавиш редактирования содержимого буфера (например, Backspace).

Большинство драйверов позволяют пользователю терминала производить упреждающий ввод данных – до того, как на них поступит запрос из программы. Введенные данные становятся доступными для чтения при нажатии клавиши ввода (например, Enter). Код этой клавиши сохраняется в буфере как признак конца строки. При поступлении запроса на чтение данных с клавиатуры драйвер выбирает из буфера строку – до признака конца строки. Если этот признак отсутствует, то процесс, выдавший запрос, блокируется до появления законченной строки в буфере драйвера. Некоторые драйверы запоминают введенные строки в стеке и обрабатывают также специальные клавиши, позволяющие выбирать строки из стека.

### Драйверы дисковых запоминающих устройств

Обычной функцией такого драйвера является перевод виртуального адреса на диске в реальный (физический). Физический адрес на диске состоит из трех компонент: головка, дорожка, сектор (в дисковых архитектурах без разбиения на сектора – смещение на дорожке). Драйвер же формирует для процессов виртуальный диск, представляемый, как линейная последовательность секторов, виртуальным адресом является номер сектора.

Интересной функцией дискового драйвера может быть планирование запросов на ввод-вывод с целью повышения эффективности обмена. В соответствии со структурой физического адреса доступ к данным на диске состоит из трех этапов – выборка составляющих этого адреса: выбора головки, выбора дорожки и выбора сектора. Выбор головки чтения/записи

производится простым переключением электронных ключей практически мгновенно. Выбор дорожки – самый времяемкий этап: он требует механического перемещения головок к требуемой дорожке; время этого перемещения зависит от расстояния перемещения. Выбор сектора на дорожке требует ожидания момента, когда требуемый сектор окажется под головкой (за счет вращения диска), время выбора сектора много меньше времени выбора дорожки.

Драйвер упорядочивает очередь запросов таким образом, чтобы минимизировать среднее время поиска дорожки. Обсуждение стратегий обслуживания мы далее ведем, исходя из предположения о случайном распределении запросов по пространству диска. Обслуживание очереди по дисциплине FCFS, очевидно, приведет к хаотическому перемещению головок и в результате – к невысокой пропускной способности драйвера и значительным механическим нагрузкам на дисковод. Из дисциплин обслуживания, позволяющих повысить пропускную способность, наиболее известными являются следующие:

- SSTF (shortest seek time first – с наименьшим временем поиска – первый) – обслуживается запрос к ближайшей дорожке; эта стратегия обеспечивает весьма высокую пропускную способность, но при высоких нагрузках с высокой вероятностью допускает бесконечное откладывание запросов, обращенных к крайним на диске дорожкам;
- Scan (сканирование) – головка движется в одном направлении, применяя на этом направлении SSTF, то есть обслуживается ближайший запрос на выбранном направлении, когда в этом направлении не остается запросов, направление меняется; стратегия обеспечивает высокую пропускную способность и исключает бесконечное откладывание, но при высоких нагрузках

время ожидания запросов, обращенных к крайним дорожкам, существенно превышает среднее;

- N-Scan (многошаговое сканирование) – головка движется в одном направлении, применяя на этом направлении FCFS, обслуживаются те запросы на выбранном направлении, которые поступили на момент начала движения, запросы, поступившие после этого момента, будут обслужены при обратном движении; стратегия обеспечивает лучшие показатели справедливости обслуживания при некотором увеличении среднего времени обслуживания;
- C-Scan (циклическое сканирование) – такая модификация стратегии Scan, в которой головка движется всегда в одном направлении, а после обслуживания последнего на направлении запроса скачком перемещается к самому дальнему запросу; стратегия полностью исключает дискриминацию крайних дорожек даже при высоких нагрузках.

Авторы, приводящие результаты исследования функционирования этих стратегий на моделях [12, 36], рекомендуют стратегию Scan при малых нагрузках и C-Scan – при больших.

Совместно с любым методом сокращения времени выбора дорожки может применяться алгоритм минимизации задержки от вращения диска SLTF (shortest latency time next – с наименьшим временем задержки – первый): при наличии нескольких запросов к одной дорожке они упорядочиваются таким образом, чтобы все они могли быть обслужены за один оборот диска.

Другим примером функций, возлагаемых на драйвер дисков может быть поддержка RAID-технологий – использование избыточных дисковых устройств для обеспечения возможности восстановления данных при сбоях. В настоящее время имеется широкий спектр реализаций RAID-

технологий – от полного переноса их на аппаратуру ввода-вывода до полной реализации их в драйвере.

## **6.5. Потоки и многоуровневые драйверы**

В целом ряде случаев данные, передаваемые из процесса на устройство или в обратном направлении, должны быть дополнительно преобразованы. Такими преобразованиями могут быть: кеширование данных, сжатие данных, криптографическое кодирование данных, согласование с форматами сетевых протоколов и т.д.

Выполнение этих функций может быть заложено непосредственно в драйвер устройства. Однако такой вариант снижает мобильность драйвера, так как не во всех применениях данной вычислительной системы и данной ОС эти функции могут быть необходимы, требования к этим функциям могут меняться, что повлечет за собой необходимость создания нового драйвера. Даже для разных процессов могут понадобиться разные способы модификации данных. Решение этой проблемы содержится в идее потоков.

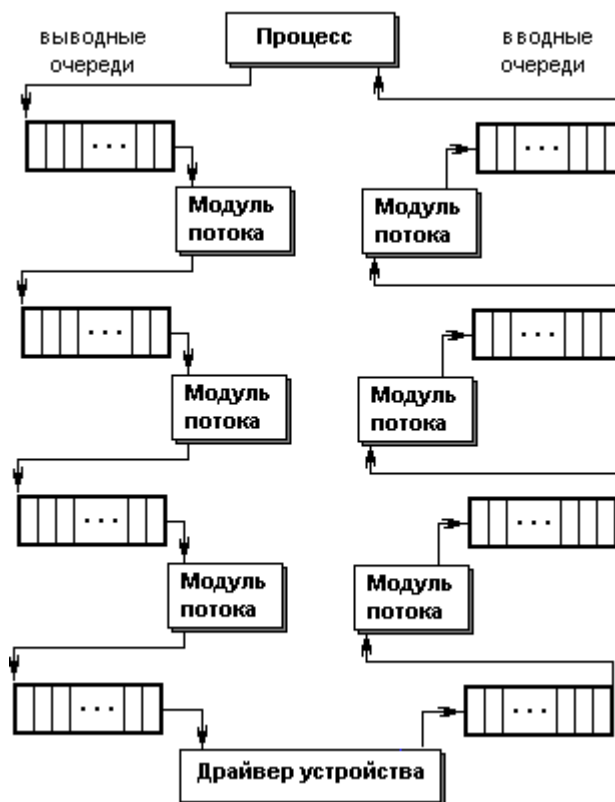


Рисунок 6.6 Поток ввода-вывода

Поток (stream) представляет собой цепочку очередей, через которые проходят данные, передаваемые от процесса драйверу устройства или в обратном направлении. Каждая очередь в этой цепочке обрабатывается одним программным модулем – модулем потока, как показано на рисунке 6.6. Данные движутся по цепочке очередей от процесса к драйверу или обратно, по пути проходя обработку в модулях потока. Модуль потока выбирает данные из своей очереди, выполняет их обработку и передает данные в очередь к следующему модулю потока. Когда модуль завершает обработку очередной порции информации, он запускает на выполнение следующий модуль, который обрабатывает эти данные или заносит их в свою очередь. Вставляя в поток новые модули (и, соответственно, новые очереди) можно обеспечивать сколь угодно сложную дополнительную обработку данных.

Потоки прозрачны для пользовательских процессов, но последним предоставляется возможность вставлять в потоки собственные модули.

Поток может быть однонаправленным или дуплексным, то есть содержать две параллельные цепочки очередей – вводную и выводную, как и показано на рисунке 6.6.

Развитием идеи потоков является многоуровневая структура драйверов, которая применяется в ряде современных ОС. Обработка данных проходит в этом случае через цепочку отдельных драйверов. Места драйверов в цепочке распределяются таким образом, чтобы каждый драйвер более низкого уровня осуществлял обработку, более привязанную к конкретному устройству. Драйверы высоких уровней могут обслуживать несколько устройств одного типа, но разных моделей. Непосредственно аппаратно-зависимым является только драйвер самого нижнего уровня (аппаратный драйвер). В некоторых ОС даже аппаратный драйвер не осуществляет прямого взаимодействия с устройством, а использует для этого сервис ОС. В многоуровневой структуре драйверов может быть оставлено место и для пользовательских драйверов, выполняющих специфическую обработку данных.

## **6.6. Интерфейс процесса**

Как мы уже отметили выше, ОС конструирует для процессов виртуальные устройства высокого уровня. Характерным примером таких устройств являются файлы. Процесс может работать с каждым файлом, как с отдельным устройством, хотя на самом деле файл представляет собой лишь часть дискового пространства. В ОС Unix концепция файла была впервые введена как универсальная метафора устройства. Это позволяет свести все многообразие управления вводом-выводом к единой форме

системных вызовов. Ниже приведен набор системных вызовов для работы с устройствами, являющийся практически общепринятым.

Открыть устройство:

```
devHandle = open(devNname, mode)
```

Этот вызов сообщает драйверу, что процесс будет работать с данным устройством. Вызов является частным случаем вызова `getResource`, и при его выполнении могут производиться действия по предупреждению или обнаружению тупиков. Процесс может быть заблокирован, если требуемое устройство занято. Могут проверяться права доступа данного процесса к данному устройству. На устройстве могут выполняться какие-то специфические для устройства действия – начальные установки. Параметр `mode` определяет направление обмена данными с устройством: чтение, запись, чтение/запись, запись в конец. Никакие другие операции с устройством невозможны, если для него не выполнена операция открытия. Манипулятор (`handle`), возвращаемый системным вызовом `open`, служит идентификатором открытого устройства во всех последующих операциях с ним.

Закрыть устройство:

```
close(devHandle)
```

Вызов сообщает драйверу, что процесс закончил работу с устройством. Ресурс-устройство открепляется от процесса, разблокируются другие процессы, ожидающие освобождения этого устройства. На устройстве могут выполняться специфические заключительные операции.

Читать данные из устройства в память:

```
read(devHandle, vAddr, counter)
```

Это запрос на передачу данных из устройства, идентифицируемого манипулятором `devHandle` в задаваемую виртуальным адресом `vAddr` область адресного пространства процесса. Если на устройстве нет того объема информации, который задан счетчиком `counter`, может быть



передан меньший объем. Вызов обычно возвращает действительный объем переданной информации.

Писать данные из памяти в устройство:

```
write(devHandle, vAddr, counter)
```

Запрос на передачу данных из заданной области виртуального адресного пространства процесса на устройство.

Позиционировать устройство:

```
seek(devHandle, position)
```

Запрос на установку устройства в заданную позицию. Применяется для устройств, имеющих внутреннюю адресацию. По умолчанию вызовы `read` и `write` устанавливают устройство в позицию, следующую за прочитанными или записанными данными.

Управление вводом-выводом:

```
ioctl(devHandle, command, parameters)
```

Запрос на выполнение специфических для устройства операций, не вписывающихся в перечисленные выше системные вызовы. Операция определяется командой `command`, например: чтение, запись, управление и т.п. Третий параметр задает дополнительные параметры, специфичные для каждой операции.

При реализации системных вызовов `read` и `write` перед разработчиком ОС возникает вопрос: блокировать или не блокировать процесс, выдавший системный вызов? Возможна синхронная и асинхронная организация ввода-вывода. При синхронной организации процесс, выдавший запрос на чтение/запись данных, требующий физического обмена данными с устройством, безусловно блокируется. Когда процесс возобновляется, он может быть уверен, что данные, которые он запросил (`read`), уже находятся в его рабочей области, и он может их использовать, а данные, которые он передал (`write`), уже пересланы, и он может записывать в область, в которой они находились, другую

информацию. Альтернативной является асинхронная организация ввода-вывода, при которой системный вызов `read/write` только инициирует операцию, далее процесс продолжает выполняться параллельно со вводом-выводом. Параллельное выполнение позволяет повысить эффективность работы как самого процесса, так и всей системы, так как снимает необходимость в переключении процессов, но синхронизация все равно необходима. Если, например, процесс запросил ввод данных, то прежде, чем он начнет их использовать, он должен убедиться, что ввод завершился. Ответственность за такую синхронизацию перекладывается на процесс. Для предоставления процессу возможности синхронизировать свои действия с выполнением операций ввода-вывода ОС обеспечивает процессу виртуальные прерывания, поддерживаемые системными вызовами, описываемыми далее.

Ждать завершения операции на устройстве:

```
wait(devHandle, delay)
```

Вызов блокирует процесс – переводит его в состояние ожидания до тех пор, пока не поступит виртуальное прерывание, сигнализирующее о завершении операции на устройстве, определяемом `devHandle`. Если операция к моменту вызова уже завершилась, процесс не блокируется. Параметр `delay` задает максимально допустимое время, которое процесс может ожидать. Если это время выходит, системный вызов `wait` заканчивается, возвращая признак ошибки. Это время может быть установлено и бесконечно большим.

Установить обработчик виртуального прерывания от устройства:

```
setHandler(devHandle, procAddr)
```

Возможность, предоставляемая этим вызовом, обеспечивает полное сходство виртуального прерывания с реальным. При поступлении виртуального прерывания от устройства выполнение процесса прерывается и управление передается на процедуру с адресом `procAddr`,

которую процесс назначил обработчиком виртуального прерывания. Все обработчики виртуальных прерываний должны соответствовать несложным системным спецификациям. Обычно обработчик имеет параметры, через которые ему передается дополнительная информация о виртуальном прерывании. Еще раз подчеркнем, что речь здесь идет именно о виртуальных, а не о реальных прерываниях. Виртуальное прерывание обеспечивается не аппаратными средствами, а моделируется для процесса операционной системой.

Асинхронная организация ввода-вывода предоставляет пользователю больше возможностей для повышения эффективности выполнения процесса, но вместе с тем и больший простор для ошибок. Характерной, например, может быть такая ошибка, появление которой должно быть предусмотрено в ОС. При записи данных, например, правильной является такая последовательность системных вызовов:

```
write...wait...write...wait... ,
```

то есть следующий блок информации не выводится, пока не будет закончен вывод предыдущего. Что произойдет, если последовательность вызовов будет такой:

```
write...write...wait...wait... ?
```

Если ОС связывает с операцией ввода-вывода двоичный флаг, то первый вызов `write` установит этот флаг в состояние "занято", второй – не изменит значение флага. Окончание первой операции вывода сбросит флаг в "свободно", что будет обработано первым вызовом `wait`, окончание второй операции вывода не повлияет на состояние флага, и второй вызов `wait` в любом случае найдет флаг в состоянии "свободно". Момент окончания второй операции, таким образом, будет "утерян". ОС должна либо расценивать подобную последовательность вызовов как ошибку процесса, либо поддерживать ее, создавая для каждой операции свой флаг.

## 6.7. Буферизация

Хотя блокировки и не влияют на результат выполнения процесса и не отражаются на его виртуальном времени, они сказываются на реальном времени его выполнения. Для интерактивных процессов они могут стать основным фактором, определяющим время реакции процесса. Невыгодны блокировки и для ОС, так как каждая блокировка – это переключение процессов, а, следовательно, накладные расходы. Одним из способов, позволяющим избежать блокировок (или, по крайней мере, уменьшить их количество) является буферизация данных. Для устройства ввода-вывода назначается буферная область в оперативной памяти. Обмен данными происходит между процессом и буферной областью, а обмен между буферной областью и устройством выполняет ОС независимо от выполнения процесса. Если, например, выполнение системного вызова `write` будет включать в себя только пересылку данных в оперативной памяти, то блокировка процесса на время выполнения этого вызова не нужна. Буферизация, таким образом, сглаживает различия в скоростях работы производителя и потребителя информации и позволяет избежать излишних блокировок.

Особенно существенный эффект буферизация может дать при последовательном вводе-выводе, так как при вводе ОС может предсказать, какой блок информации понадобится следующим и произвести его упреждающее чтение в буфер. Еще один полезный эффект буферизации – ОС имеет возможность сгруппировать данные в буфере и производить обмен с внешним устройством большими блоками.

Богатые вариантами средства организации ввода-вывода с использованием буферизации были впервые реализованы в OS/360 [21] и унаследованы следующими поколениями ОС мейнфреймов. При

синхронной организации ввода-вывода (в OS/360 – "методы доступа с очередями") буферизация прозрачна для процесса, она полностью обеспечивается ОС. При асинхронной организации ("базисные методы доступа") процесс сам может организовать буферизацию. Устройству назначается буферная область, которая форматируется как буферный пул. В методах с очередями это назначение происходит автоматически, программист может только изменить размер пула, если его не устраивает принятый по умолчанию. В базисных методах есть специальные системные вызовы для выделения, освобождения, форматирования буферного пула и связывания пула с устройством. Буферизация не обязательно означает дополнительную пересылку данных в оперативной памяти. Методы с очередями предоставляют три альтернативных режима управления буферизацией:

- режим пересылки – данные пересылаются в оперативной памяти между рабочей областью процесса и буфером;
- режим указания – данные не пересылаются; при вводе процесс получает от ОС адрес буфера, содержащего введенные данные, и может использовать его как рабочую область, при выводе процесс получает от ОС адрес свободного выводного буфера и использует его как свою рабочую область, формируя в ней данные для вывода;
- режим подстановки – данные не пересылаются; при вводе процесс получает от ОС адрес заполненного вводного буфера, а свою рабочую область "передает" во вводной буферный пул; при выводе процесс "передает" в выводной пул свою рабочую область, заполненную выводными данными, а взамен получает свободный буфер из пула.

Два интересных примера буферизации мы возьмем из ОС Unix.

В Unix буферизация обмена с дисковыми накопителями (кеширование) является тотальной, через кеш проходят все данные, которыми ОС и процессы обмениваются с дисками, и под кеш отводится значительная часть оперативной памяти. Кеширование дает значительный эффект и при случайном (непоследовательном) обмене, характерном для многозадачной многопользовательской ОС. В дескрипторе каждого буфера в кеше имеются поля:

- состояние буфера (свободен / содержит правильную информацию / грязный / заблокирован / занят в операции ввода/вывода/ожидается каким-либо процессом);
- адрес на внешней памяти блока информации, содержащегося в буфере;
- указатели, связывающие буфера в хеш-очереди (см.дальше) и в список свободных буферов.

Список свободных буферов организован как простая FIFO-очередь: ядро ОС выбирает буфер из головы списка, освобождающийся буфер включает в конец списка. Таким образом, буфер, попавший в список свободных, еще некоторое время находится в этом списке, сохраняя свое содержимое, и может быть выбран из списка, если к нему будет обращение. При обращении к дисковому вводу-выводу основным параметром является адрес блока на внешней памяти. ОС сначала ищет в кеше соответствующий блоку буфер и только при неудачном поиске в кеше назначает свободный буфер и производит физическое обращение к диску. Для ускорения поиска в кеше все буфера распределяются по нескольким спискам, именуемым хеш-очередями. Номер списка определяется как результат весьма простой функции хеширования, аргументом которой является адрес на внешней памяти. Применение хеширования позволяет равномерно распределить буфера по хеш-очередям. Каждый буфер может входить только в одну хеш-очередь, но

также может быть включен и в список свободных. Процесс, выдавший запрос на ввод-вывод, блокируется в случаях:

- если буфера нет в хеш-очереди и список свободных блоков пуст;
- если буфер есть в хеш-очереди, но занят.

При освобождении блока разблокируются все процессы, ждущие освобождения этого или любого блока.

Все современные ОС (OS/2, Windows 95, Windows NT) в той или иной степени применяют тотальное кеширование при обмене с дисками.

Второй пример – буферизация ввода в драйвере терминала. Задача такой буферизации – обеспечить накопление данных и возможность упреждающего ввода с терминала. Для этой буферизации характерны: последовательный ввод, сравнительно небольшие объемы данных в буфере, непостоянство длины информации, содержащейся в буфере. Буфер представляет цепочку блоков, каждый из которых имеет постоянную длину. В каждом блоке имеются следующие поля:

- указатель на следующий блок в цепочке;
- смещение первого символа в поле данных;
- смещение последнего символа в поле данных;
- поле данных для хранения N символов.

Пример буфера при N=8 показан на рисунке 6.7. Ядро ОС хранит указатели на первый и последний блоки цепочки и ведет список свободных блоков (очередь LIFO). Ядро обеспечивает:

- назначение драйверу свободного блока;
- возвращение блока в список свободных;
- выбор первого символа из буфера (при этом возможно освобождение блока);
- добавление символа в конец буфера (при этом возможно выделение нового блока).



Рисунок 6.7 Буфер терминала для Unix

## КОНТРОЛЬНЫЕ ВОПРОСЫ

1. Охарактеризуйте методы виртуализации устройств в ОС. Приведите примеры их применения.
2. Почему даже при закреплении устройства за процессом устройство все равно остается виртуальным?
3. Чем вы объясните столь долгое и успешное существование концепции каналов ввода-вывода?
4. Чем объясняется двухуровневая (как минимум) структура драйвера устройства?
5. В некоторых современных ОС драйверами называются также и модули ОС, не имеющие отношения к управлению устройствами. Чем может быть объяснен такой подход?
6. Назовите те функции, которые вы считаете целесообразным добавить в драйвер клавиатуры для обеспечения большего удобства пользователю.
7. Каким образом можно обеспечить выдачу процессу "сигнала тревоги" через заданный интервал времени, если в системе нет программируемого интервального таймера?



8. В чем цель стратегии драйвера диска?
9. Сопоставьте потоки и многоуровневые драйверы. В чем их сходство и различия?
10. Какие цели преследует буферизация ввода-вывода?
11. Сопоставьте по эффективности три режима буферизации (пересылка, указание, подстановка), описанные в разделе 6.7.
12. Для чего нужен системный вызов `ioctl`? Приведите примеры устройств, для которых этот вызов совершенно необходим.

## **Глава 7. Файловые системы**

### **7.1. Иерархическая модель файловой системы**

Файл — именованная совокупность данных. Это определение относится к виртуальному файлу: каким он представляется пользователю. Определение виртуального файла не конкретизирует, где именно, на каких носителях находятся данные, из каких элементов эта совокупность состоит и каковы отношения между элементами. Отсутствие детализации в определении виртуального файла делает его удобной универсальной метафорой любых внешних по отношению к процессу данных. В ОС Unix впервые было введено представление всех внешних устройств как виртуальных файлов. Это представление прочно укоренилось, и в современных ОС имеется тенденция более широкого использования файловой метафоры. В таких системах, например, имена всех внешних по отношению к процессу именованных данных (семафоры, каналы-транспортеры, очереди и т.д.) формируются по соглашениям именования файлов. Многие современные ОС поддерживают так называемую виртуальную файловую систему (VFS), которая позволяет включить в

единое пространство имен и в единую структуру именования не только физические файлы, находящиеся на устройствах с различной структурой хранения, но и все ресурсы, имеющие внешние имена.

Понятие же физического файла связывают с данными, хранящимися на внешней памяти. Устройства внешней памяти предназначены для длительного хранения данных. Файл, созданный на внешней памяти, может существовать на ней сколь угодно долго, пока не будет уничтожен явно заданной операцией уничтожения. Характерным является то, что файл продолжает существовать и после завершения создавшего его процесса, данные файла могут быть многократно прочитаны, модифицированы, полностью заменены этим же или другим процессом. Физический файл есть набор записей на устройстве внешней памяти, сгруппированных таким образом, чтобы управлять доступом к ним, их чтением и модификацией.

Файловой системой (ФС) называется та часть ОС, которая обеспечивает перевод виртуального представления файла в физическое. Этот перевод выполняется поэтапно, что позволяет представить ФС в виде иерархической модели, показанной на рисунке 7.1.

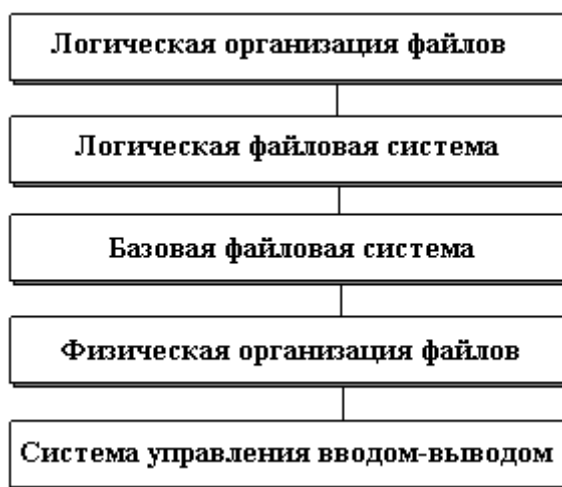


Рисунок 7.1 Иерархическая модель файловой системы

Подсистема логической организации файлов обеспечивает API процессов в соответствии с той логической структурой, которую имеет виртуальный файл.

Логическая ФС выполняет перевод символьного имени файла в некоторый внутрисистемный идентификатор файла. Этот перевод включает в себя поиск по справочникам. Идентификатор обычно представляет собой некоторую простую структуру данных, адресующую дескриптор файла, который используется на следующем уровне иерархии.

Базовая ФС выполняет открытие и закрытие файлов и сопровождение открытых файлов. Базовая ФС находит по идентификатору дескриптор файла – системную структуру данных, содержащую информацию о местонахождении файла, его характеристиках и состоянии. При открытии файла этот уровень создает расширенный дескриптор файла, отслеживающий в ходе работы процессов с файлом его текущее состояние. Базовая ФС, возможно, также контролирует соблюдение прав доступа к файлу.

Подсистема физической организации файлов выполняет перевод виртуальных файловых адресов в реальные адреса на носителях. Этот уровень отслеживает размещение файлов на внешней памяти и управляет распределением пространства внешней памяти.

Система управления вводом-выводом (СУВВ) занимается формированием управляющих воздействий на устройства внешней памяти, их выполнением и обработкой прерываний. Этот уровень обеспечивается драйверами устройств, рассмотренными нами в предыдущем разделе, поэтому здесь мы этот уровень рассматривать не будем. Отметим только, что управление устройствами (любыми, а не только устройствами внешней памяти) является вырожденным случаем иерархической модели, в котором отсутствует уровень логической ФС, а функции физической организации и управления вводом-выводом реализованы в драйвере.

В следующих разделах мы рассмотрим подробнее уровни ФС, двигаясь сверху вниз по иерархии.

## 7.2. Логическая организация файлов. Интерфейсы

Вне зависимости от логической структуры виртуального файла для получения процессом доступа к данным файла должен быть выполнен системный вызов `open`:

```
handle = open(fileName, mode)
```

Здесь `fileName` – символьное имя, идентифицирующее файл для пользователя, соглашения об именовании файлов обсуждаются в следующем разделе. `mode` – режим, задает способ обработки файла (чтение, запись, запись в конец, чтение/запись, синхронное/асинхронное выполнение операций ввода/вывода, параметры буферизации, возможность совместного использования файла процессами и т.д.). Возвращаемый манипулятор `handle` используется процессом для идентификации файла во всех последующих операциях с ним.

Системный вызов:

```
close(handle)
```

закрывает файл, т. е., заканчивает работу процесса с файлом. Как правило, ОС обеспечивает при завершении процесса автоматическое закрытие всех открытых им файлов.

Синтаксис и семантика системных вызовов, связанных с чтением/записью и поиском данных в файле, зависит от логической структуры файла. Логические структуры виртуальных файлов прежде всего можно подразделить на байт–ориентированные и записе–ориентированные.

Байт–ориентированные файлы представляются как линейная последовательность байтов без какого-либо дополнительного

структурирования. API байт-ориентированных файлов обеспечивается тремя основными системными вызовами:

```
read(handle, vAddr, byteCounter);  
write(handle, vAddr, byteCounter);  
seek (handle, offset, form).
```

Здесь `vAddr` – адрес той области в виртуальном адресном пространстве процесса, с которой происходит обмен. За одну операцию обмена читается или пишется заданное `byteCounter` количество байтов. К байт-ориентированным файлам обеспечивается произвольный доступ, для чего вводится понятие файлового курсора – номера того байта файла, который будет читаться/записываться следующим. По умолчанию при открытии файла курсор устанавливается на начало файла и сдвигается при каждой операции `read/write` на `byteCounter`. Но системный вызов `seek` дает процессу возможность установить курсор в произвольную позицию, причем смещение `offset` может задаваться как от начала файла, так и от его конца или от текущей позиции курсора (это задается параметром `from`). Возможен также сервисный системный вызов, возвращающий текущее положение курсора. Системный вызов `read` должен адекватно реагировать на попытку чтения данных за концом файла. Стандартным решением является возврат этим вызовом числа прочитанных байт – при достижении конца файла возвращаемое значение будет меньше, чем `byteCounter`.

Безусловно, байт-ориентированная организация является наиболее универсальной из всех возможных, так как на неструктурированную последовательность байтов приложение может наложить любую собственную логическую структуру. Приведем пример, наверняка известный читателю: MS DOS поддерживает только байт-ориентированную организацию файлов, но системы программирования

Pascal (приложения в MS DOS) обеспечивают работу с файлами, состоящими из записей – тип данных `file of...`. Байт-ориентированные файлы являются единственной логической файловой структурой, поддерживаемой ОС Unix, и "с подачи" этой системы они включены в стандарт на переносимость ОС.

Альтернативной файловой организацией является записе-ориентированная. Записе-ориентированные файлы поддерживает фирма IBM в нескольких поколениях своих ОС для ЭВМ средней и большой мощности. В таких файлах единицей обмена является запись – порция данных, состоящая из одного или нескольких байтов и, возможно, имеющая свою внутреннюю структуру. Имеется целый ряд методов логической организации записе-ориентированных файлов.

Последовательные файлы – файлы, ориентированные на обработку запись за записью: от первой записи до последней. Синтаксис и семантика системных вызовов `read` и `write` для таких файлов такие же, как и для байт-ориентированных, но объем данных задается количеством записей, а не байтов. Произвольное позиционирование файлового курсора для последовательных файлов невозможно, но система может предоставлять системные вызовы для сдвига на запись вперед или назад. Последовательные файлы могут состоять из записей фиксированной или переменной длины. В последнем случае либо длина входит в состав записи, как одно из ее полей, либо запись содержит специальный код – признак конца записи.

Файлы прямого доступа – предполагается, что для каждой записи файла имеется логическая идентификация или ключ (`key`). Доступ производится к произвольной записи файла с указанием ее ключа. Системные вызовы для чтения записи имеют вид:

```
keyRead (handle, key, vAdd);  
keyWrite (handle, key, vAdd).
```

Выполнение такого вызова включает в себя позиционирование файлового курсора на запись, определяемую ключом `key`, и собственно обмен. Ключ может входить в состав записи как одно из ее полей (например, поле фамилии в списке личного состава) или не содержаться в составе записи (например, ключ – порядковый номер записи).

Файлы комбинированного доступа допускают как прямой доступ по ключу (`keyRead/keyWrite`), так и последовательный (`read/write`) в порядке возрастания ключей. Системный вызов, например, для чтения по ключу 'Коваль' обеспечит считывание записи, относящейся к сотруднику Ковалю, последующие системные вызовы последовательного чтения будут считывать записи, относящиеся к сотрудникам, следующим за Ковалем в принятом (например, в алфавитном) порядке.

Какой организации отдать предпочтение: байт– или записе–ориентированной? Как мы уже отмечали, байт–ориентированная организация гибче, но при ней задача структурирования файла перекладывается на приложения – а задача эта не всегда простая. Фирма IBM, не желая, с одной стороны, отказываться от преимуществ записе–ориентированной организации, а с другой, – желая обеспечить совместимость своих ОС со стандартами, обеспечивает поддержку обеих структур, и такое решение представляется нам оптимальным.

Интересной, хотя пока несколько экзотической формой файлов являются файлы отображаемого доступа. Такие файлы при открытии отображаются в виртуальное адресное пространство процесса. Операция `open` может возвращать дескриптор сегмента в виртуальной памяти, и в дальнейшем все манипулирование с данными файла будет выполняться, как манипулирование с данными в памяти. Хотя такое решение является весьма элегантным, его реализация требует значительного расширения разрядности представления виртуального адреса и реализуется пока только в 64-разрядных клонах Unix.

### 7.3. Логическая файловая система. Каталоги

Логическая ФС рассматривает файл как единицу хранения. Удачным нам представляется сравнение логической ФС с библиотекарем, который не должен знать содержание книги (файла), но обязан уметь быстро найти требуемую книгу в книгохранилище. Как бы продолжая это сравнение, файловые справочники ОС носят названия каталогов (catalog) или оглавлений (directory).

Файловые справочники (и это естественно) располагаются на тех же носителях, что и файлы данных. Первоначально с каждой единицей сменного носителя (пакет магнитных дисков, бобина магнитной ленты) жестко связывался свой единственный справочник, и смена носителя вызывала смену справочника. Такая единица сменного носителя получила название тома (volume). Дальнейшее развитие вычислительных систем привело с одной стороны к увеличению объемов несменяемых носителей, так что несколько их единиц могут составлять один том, с другой – к практике разбиения одного физического носителя на несколько логических томов. Таким образом, мы определяем том как часть внешней памяти вычислительной системы, имеющую собственный справочник (каталог). Ниже мы рассмотрим вначале логическую структуру хранения файлов на отдельном томе, а затем – интеграцию томов.

Элемент каталога содержит как минимум символьное имя файла и адрес его дескриптора, иногда дескриптор файла может непосредственно входить в элемент каталога.

Прежде чем переходить к структурам каталогов, остановимся на именовании файлов. Каждый файл имеет символьное имя, которое позволяет пользователю обращаться к данному конкретному файлу. Каждая ОС устанавливает собственные соглашения о формировании имен.



Как правило, эти соглашения допускают или даже требуют разделения имени файла на составные части – компоненты. При записи символьного имени его компоненты обычно разделяются точкой. Разные ОС накладывают разные ограничения на количество и длины компонентов имени: от двухкомпонентных имен с компонентами ограниченной длины до неограниченного их числа и практически неограниченной длины. Некоторые ОС резервируют одну из составляющих имени файла в качестве описателя типа информации, хранящейся в файле. Даже если ОС не предъявляет дополнительных требований к именованию, пользователи применяют составные имена файлов как средство структурирования хранения своей информации. Для ОС, работающих с многокомпонентными именами, распространенной является практика применения "символа-джокера", обычно – '\*' или '&'. Употребление джокера вместо одного из компонентов имени обеспечивает выполнение системных вызовов, адресованных логической ФС, как групповых операций, – над всеми файлами с любыми значениями этой составляющей имени.

Каталоги файлов выполняют две функции: обеспечивают поиск файлов и структурирование хранения информации.

Простейшей структурой каталога является плоский (flat) каталог. Информация обо всех файлах сведена в одну таблицу и поиск файла сводится к поиску в этой таблице – линейному или двоичному, если таблица упорядочена. Такая структура каталога не допускает, чтобы файлы имели одинаковые имена. Плоская структура не выполняет второй функции каталога – структурирования хранения и может быть эффективной только для ОС, работающих с томами носителей небольшого объема, когда том содержит файлы, принадлежащие одному пользователю. Если том содержит файлы, принадлежащие разным пользователям, то возможны конфликты имен – назначение разными пользователями

одинаковых имен для своих файлов. Даже если такие конфликты предотвращаются (за счет многокомпонентных имен), то остается существенный недостаток: каталог открыт для чтения всем пользователям, а в некоторых случаях даже список имен файлов пользователя может быть его конфиденциальной информацией.

Простейшим решением разделения файлов между пользователями (или по другим признакам) является двухуровневая структура каталога. В этой структуре каждый пользователь имеет собственный каталог. Каталог именуется по правилам именования файлов, его имя также может быть составным. В такой структуре возникают понятия полного и локального имен файла. Полное имя файла состоит из имени каталога и имени файла в каталоге. Обычно эти имена разделяются наклонной чертой '`\`' или '`/`'. Вторая часть полного имени является локальным именем файла. На томе не может быть двух файлов, имеющих одинаковые полные имена, но файлы с одинаковыми локальными именами могут находиться в разных каталогах. Для сокращения именования файлов процесс может сообщить ОС устанавливаемое по умолчанию имя каталога. Каталог, имя которого устанавливается по умолчанию, называется рабочим, или текущим. К файлам в текущем каталоге процесс может обращаться по локальным именам, к файлам в других каталогах – только по полным именам.

Нетрудно распространить двух- или трехуровневую структуру каталога на произвольное число уровней – каталог приобретает древовидную или иерархическую структуру. Пример такой структуры показан на рисунке 7.2. Иерархическая структура каталогов практически исключает возможность конфликта имен. Полное имя файла складывается из перечисления через '`/`' всего пути (path) от корневого каталога, имеющего имя '`/`', до данного файла, например:

```
/system/utility/disks/anti/test1
```

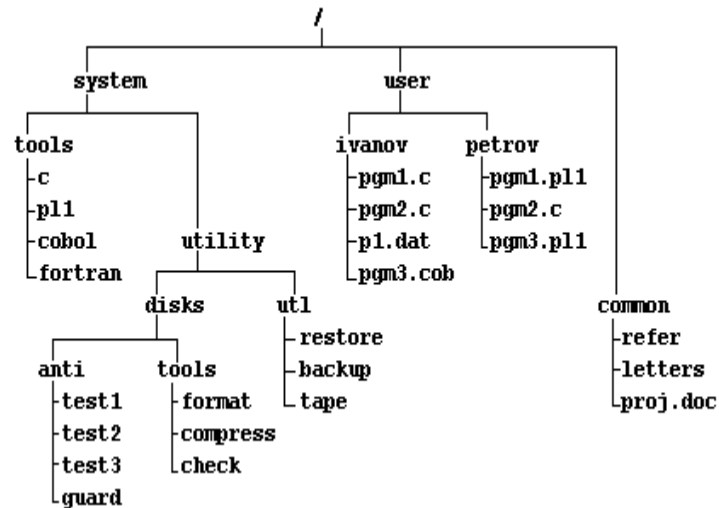


Рисунок 7.2 Пример иерархической структуры каталогов

Иерархические каталоги хорошо известны пользователям MS DOS, поэтому мы будем останавливаться прежде всего на тех их свойствах, которые в этой системе неизвестны или непопулярны.

Обычно в каждый узел древовидной структуры включается запись о каталоге, являющемся по отношению к данному родительским. В такой записи имя каталога-родителя обозначается каким-либо специальным символом или комбинацией символов, например: '..'. Это позволяет обращаться к файлам в другом каталоге, задавая путь с отправной точкой либо из корневого каталога, либо из текущего (рабочего). Так, если рабочим является каталог /users/ivanov (см. рисунок 7.2), то возможно обращение:

```
../../petrov/pgm3.c
```

В каталог также включается запись о нем самом, обычно обозначаемая, как '.'. Если рабочим является каталог /system/utility (см. рисунок 7.2), то возможно обращение:

```
./disks/anti/test.1
```

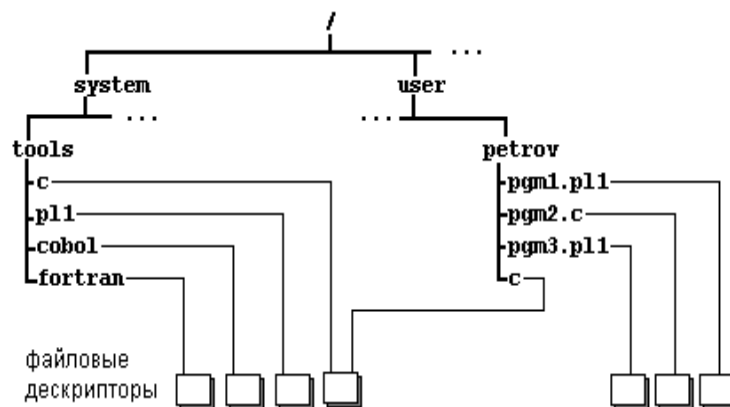
Древовидный каталог является элегантной и в большинстве случаев удобной структурой, но в некоторых случаях может быть полезным внесение в идеальную древовидную структуру некоторые нарушения. Но

прежде чем обсуждать такие нарушения, рассмотрим вопрос о составе элемента каталога.

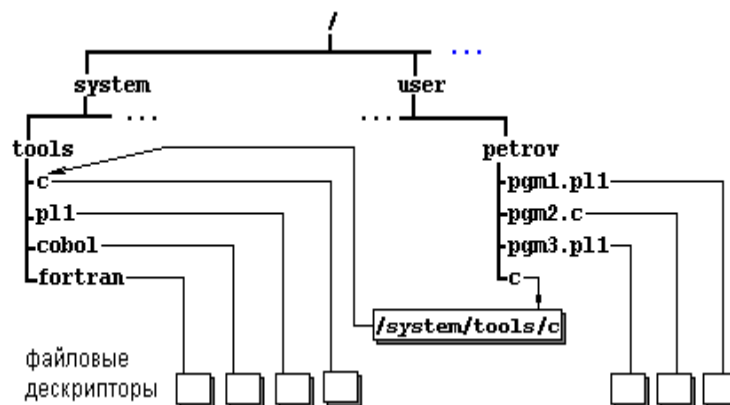
Каждый элемент каталога описывает файл или подкаталог (subdirectory). С точки зрения ниже лежащих уровней ФС подкаталог является таким же файлом, как и файлы пользователей. Выше мы уже говорили о том, что элемент каталога содержит символьное имя файла или подкаталога и указатель на файловый дескриптор. Вся остальная информация о файле содержится в дескрипторе файла. В некоторых системах сам файловый дескриптор также входит в состав элемента каталога. Но хранение дескриптора файла отдельно от каталога логически вытекает из иерархической структуры ФС, поскольку поиск файла (работа с каталогами) и обработка файла (работа с дескриптором) выполняются разными уровнями ФС. Дескрипторы файлов могут храниться вместе с данными файлов или в специально отведенной области. Раздельное хранение каталогов и дескрипторов обеспечивает ряд дополнительных возможностей в логической ФС, первая из которых – алиасы.

Алиасами (alias) или связями (link) называются элементы каталогов, указывающие на один и тот же файловый дескриптор. Алиасы могут находиться в одном и том же подкаталоге, в этом случае альтернативные имена обязательно должны быть разными, или в разных подкаталогах – тогда имена могут быть одинаковыми. Два разных элемента каталога, таким образом, указывают на один и тот же физический файл. Если при обращении к файлу по одному из альтернативных имен в данных файла были сделаны изменения, то при чтении файла по другому имени эти изменения будут найдены в файле. Как правило, алиасы создаются для того, чтобы включить в рабочий каталог пользователя файлы, находящиеся в других каталогах, но часто используемые данным пользователем. Например, если обычным рабочим каталогом для нас является /user/petrov, но нам часто приходится обращаться к файлу

`/system/tools/c`, то удобно создать для этого файла альтернативный элемент каталога `/user/petrov/c`, это позволит нам в дальнейшем обращаться к этому файлу из нашего рабочего каталога по локальному имени. Аlias для рассмотренного примера показан на рисунке 7.3.а. Отметим, что если для файла создан алиас, то оба альтернативные имени файла, старое и новое, являются равноправными. Нельзя говорить, что файл принадлежит к тому каталогу, в котором он был создан, и только присоединен к другому каталогу, – файл в равной степени принадлежит обоим каталогам. При удалении файла по одному из альтернативных имен удаляется только соответствующий элемент в каталоге, физический же файл (и его дескриптор) продолжает существовать, он будет уничтожен, когда будет удалена последняя ссылка на него.



а). алиас



б). косвенный файл

### Рисунок 7.3 Альтернативное имя для файла

Косвенным файлом (indirect file) или символьной связью (symbolic link) называется элемент каталога, который ссылается на другой элемент каталога. Ссылка производится обычно путем указания полного символьного имени каталога. Так, в приведенном выше примере элемент каталога `/users/petrov/c` может вместо адреса дескриптора содержать символьную строку `'/system/tools/c'` или ссылаться на "псевдодескриптор", содержащий эту символьную строку. Такой косвенный файл показан на рисунке 7.3.б. При обращении к файлу по имени `'/users/petrov/c'` ФС в процессе поиска, дойдя до этого места, продолжит поиск по пути, который указан в косвенной ссылке. Нетрудно обеспечить и многоуровневые косвенные ссылки. Подобно косвенным файлам, могут быть и косвенные каталоги. Принципиальное отличие косвенных файлов от алиасов в том, что имена косвенных файлов имеют неравные права с основным именем. Только один элемент каталога (основной) ссылается на физический файл, остальные же – на элемент каталога. Поэтому удаление физического файла возможно только по основному имени, удаления же по косвенным именам удаляют только элементы каталогов. Если файл удален по основному имени, то косвенные ссылки на него, как правило, остаются в каталогах, и обращения по косвенным именам приведут к ошибкам. Задача чистки каталогов от неактуальных косвенных имен может возлагаться либо на пользователей – владельцев каталогов, либо на администратора системы, в распоряжении которого должны быть соответствующие утилиты.

Раздельное хранение каталогов и дескрипторов предоставляет, как мы показали, дополнительные возможности, а также создает почву для возникновения дополнительных ошибок – "беспризорных" файлов, то есть файлов, на которые нет ссылок ни в каком каталоге. В распоряжении

системного администратора должны быть утилиты, позволяющие исправлять такие ошибки (как, например, fsck и fsdb в Unix).

Внешняя память вычислительной системы может состоять из нескольких томов, каждый из которых имеет свой каталог. Если компьютер работает в составе сети, то в его пространство внешней памяти могут также включаться тома, расположенные в других узлах сети. В персональных системах пользователь при начале работы автоматически получает доступ ко всем наличным томам. В многопользовательской системе пользователю предоставляется только один определенный том (или несколько томов), а остальные должны быть подключены (mount – монтированы) пользователем явным образом. При работе с несколькими томами структура хранения информации может представлять собой "лес" или "дерево". В первом случае каждый том представляется как отдельное дерево каталогов и полное имя файла включает в себя имя тома (OS/2, Windows, CMS). Во втором случае новый том подключается к основному и выглядит, как ветвь в общем дереве каталогов (Unix, OS/400). Если каталоги всех томов имеют одинаковую логическую структуру (а стандартной является структура иерархическая), то в одно дерево могут быть объединены даже тома, имеющие различную физическую структуру. С логической точки зрения нет разницы в том, находится ли монтируемый том на этом же компьютере или в другом узле сети. Кроме того, операция монтирования может быть реализована таким образом, чтобы давать пользователю доступ не ко всему тому (к корневому каталогу), а только к одной ветви его дерева каталогов.

## **7.4. Логическая файловая система. Системные вызовы**

В API ФС зачастую трудно установить, к какой части ФС адресован тот или иной системный вызов, так как большинство вызовов проходят

обработку на всех уровнях ФС. Тем не менее, большинство системных вызовов мы рассматриваем именно вместе с логической ФС, так как существенной составляющей этих вызовов является именно работа с каталогами. Не следует, однако, забывать, что даже если функции вызова ограничиваются только работой с каталогом, для его выполнения тоже требуется обращение к нижним уровням иерархии ФС, так как каталог – это тоже файл, который имеет свою физическую структуру и который тоже надо читать и записывать.

Для рассмотрения мы разобьем все системные вызовы на следующие группы:

- вызовы, работающие с каталогами;
- вызовы, работающие с файлами;
- вызовы, работающие с томами.

Вызовы, работающие с каталогами

Установить рабочий (текущий) каталог:

```
setCurrentDirectory(dirName)
```

При помощи этого вызова процесс сообщает ОС, какой каталог является для него рабочим. В дальнейшем допустимы обращения к файлам в этом каталоге по локальным именам. В ходе своего выполнения процесс может неоднократно менять свой рабочий каталог. Имя каталога `dirName` задается в виде символьной строки, содержащей путь, отправной точкой которого может быть либо корневой каталог, либо – текущий. Логическая ФС (совместно с нижними уровнями ФС) обеспечивает движение по этому пути. В API ОС может быть включен также информационный вызов `getCurrentDirectory`, возвращающий полное имя текущего каталога.

Создать подкаталог:

```
createDirectory(dirName)
```



При помощи этого вызова процесс может создать новый подкаталог. Обычно имя каталога `dirName` задается локальным и новый подкаталог создается в текущем каталоге, но может быть допущено и задание полного имени. Этот вызов может рассматриваться как специальный случай вызова `createFile`.

Удалить подкаталог:

```
removeDirectory(dirName)
```

Конструктор ОС должен особо определить реакцию системы на применение этого вызова к непустому каталогу: то ли завершать в этом случае вызов с ошибкой, то ли удалять каталог со всем его содержимым.

Вызовы, работающие с файлами

Создать файл:

```
createFile(fileName, parameters)
```

Вызов создает новый физический файл в текущем каталоге, если имя задано в локальной форме или в другом - если задано полное имя. Другие параметры – `parameters` – задают атрибуты, заносимые в дескриптор создаваемого файла.

Создать алиас:

```
createAlias(fileName, aliasName)
```

Вызов создает новый элемент каталога, ссылающийся на тот же дескриптор физического файла.

Создать косвенный файл:

```
createIndirect(fileName, indirectName)
```

Вызов создает новый элемент каталога, ссылающийся на старый элемент каталога.

Удалить файл:

```
deleteFile(fileName)
```

Вызов удаляет элемент каталога, соответствующий заданному имени. Если имя является именем косвенного файла или если у файла имеются

альтернативные имена, то удаляется только элемент каталога, в противном случае уничтожаются также и физический файл, и его дескриптор.

Переместить файл:

```
moveFile(oldName, newName )
```

Вызов перемещает файл в другой каталог. Одно из имен может быть локальным (то есть исходный или целевой каталог может быть текущим), другое – обязательно должно быть полным. Данная операция не требует перемещения физического файла или его дескриптора, а только элемента каталога. Вызов может быть реализован как комбинация двух вызовов, описанных выше: `createAlias` – в новом каталоге и `deleteFile` – в старом. Как частный случай этого вызова может рассматриваться вызов `renameFile` – переименовать файл, но в целях повышения эффективности его реализация может быть выполнена путем исправления данных в элементе каталога, остающемся на том же месте.

Копировать файл:

```
copyFile(oldName, newName )
```

Вызов копирует файл в другой каталог или в тот же каталог под новым именем. В отличие от вызова `moveFile` копируется физический файл – данные файла и файловый дескриптор, а для копии создается новый элемент каталога. Далее старый файл и копия существуют независимо друг от друга.

Вызовы, работающие с томами.

Монтировать том:

```
mount(entrName, extName)
```

Вызов подключает к ФС новый том. `entrName` задает идентификацию тома в ФС – это логическое имя тома или имя подкаталога, которым представляется том в едином дереве каталогов. `extName` идентифицирует

том вне ФС – это может быть адрес устройства, на котором том установлен, сетевой адрес узла при удаленном доступе и т.п.

Снять том:

```
unMount (entrName )
```

Вызов отключает от ФС ранее монтированный том.

## **7.5. Базовая файловая система**

### **ДЕСКРИПТОР ФАЙЛА**

Выше мы уже неоднократно упоминали дескриптор файла и читатель, наверное, уже сделал правильный вывод о том, что эта структура данных является ключевой при переводе виртуального представления файла в реальное. Дескриптор однозначно связан с физическим файлом, но храниться может как вместе с файлом, так и отдельно от него. В любом случае, однако, дескриптор файла является прозрачным для процессов пользователей, он обрабатывается только ОС. Дескриптор содержит информацию о физической природе файла и информацию о его использовании и защите. Естественно, что формат дескриптора определяется спецификациями конкретной ОС, и мы можем дать только приблизительное перечисление его полей. В общем случае в дескриптор файла могут входить следующие составляющие:

- тип файла (файл данных, косвенный файл, каталог, файл-устройство и т.п.);
- тип данных файла (текст, объектный модуль, программа и т.п.);
- сведения об организации файла (для перехода от логической структуры к физической);
- размер файла;
- план (layout) размещения файла на устройстве внешней памяти;

- список прав доступа;
- время (создания, последней модификации, последнего доступа);
- счетчик алиасов файла;
- и т.д.

Некоторые из входящих в состав дескриптора подструктуры данных мы рассмотрим ниже более подробно. В основном содержимое дескриптора формируется при создании файла или каталога (системные вызовы `createFile`, `createDirectory`), некоторые поля изменяются при работе с файлом. Предоставляем читателю самому проследить, как системные вызовы, описанные выше, работают с дескрипторами файлов/каталогов.

Наиболее важным для дескриптора файла является системный вызов `open`: он производит активизацию дескриптора. Для файлов на устройствах внешней памяти при этом производится считывание дескриптора в оперативную память и построение его расширения, называемого дескриптором открытого файла. Для устройств, представляемых в виде виртуальных файлов, файловых дескрипторов на внешней памяти не существует, но в оперативной памяти открытие такого виртуального файла вызывает построение дескриптора открытого файла, отличающегося от дескриптора открытого файла на внешней памяти, как правило, содержимым полей, но не их форматом. Для каталогов системный вызов `open` не выполняется явным образом, но при выполнении функций поиска по заданному пути в логической ФС активизируются дескрипторы всех каталогов, входящих в путь.

В расширение дескриптора открытого файла в оперативной памяти могут входить:

- счетчик открытий файла (файл может одновременно использоваться несколькими процессами);

- замок (lock) разделяемого доступа;
- режим обработки;
- режим буферизации;
- текущее положение файлового курсора;
- идентификация устройства, на котором расположен файл;
- информация, зависящая от типа файла.

Так, например, при подключении тома к ФС в виде ветви общего дерева каталогов, в памяти создается дескриптор для псевдокаталога, которым представлен подключенный том. Этот дескриптор содержит признак, описывающий его как "точку монтирования", и ссылку на элемент специальной системной таблицы монтирования. Элемент же таблицы содержит описание тома и указатели как на корневой каталог тома, так и обратно – на "точку монтирования", что позволяет проходить эту точку при движении по дереву в обоих направлениях.

Очевидно, что некоторые поля дескриптора должны изменяться автоматически при работе с файлом. Все изменения, выполненные в дескрипторе в ходе работы с открытым файлом, запоминаются в его копии на внешней памяти при выполнении системного вызова `close`.

Обычно ОС предоставляют также в составе своего API тот или иной набор системных вызовов, сводящихся к двум типам: `getFileInfo`, `setFileInfo` – получить или установить информацию о файле. Выполнение вызовов этой группы позволяет прочитать/записать значения отдельных полей файловых дескрипторов.

## **УПРАВЛЕНИЕ ДОСТУПОМ**

В некоторых ОС одной из важнейших функций базовой ФС является контроль за доступом пользователей к файлам. Мы говорим "в некоторых", так как в ряде ОС проблема контроля доступа решается на общесистемном уровне и доступ к файлам – ее частный случай. Такие ОС рассматриваются

нами в главе 10, здесь же мы остановимся на случае, когда доступ к файлам контролируется базовой ФС, и рассмотрим его на примере ОС Unix.

В Unix возможны следующие режимы доступа к файлам: *r* – чтение, *w* – запись, *x* – выполнение. Возможны также их комбинации.

Пользователи подразделяются на следующие категории:

- владелец – пользователь, который создал файл;
- группа – пользователи, входящие в ту же группу, что и владелец файла;
- все остальные пользователи.

Для каждого файла определяется допустимый режим доступа для каждой из этих трех групп. Так, например, если для файла доступ закодирован в виде:

`rwxr-x--x`,

то это означает, что владелец имеет право читать, писать и выполнять файл (*rw**x*), остальные члены группы владельца имеют право читать и выполнять файл (*r*–*x*), все другие пользователи – только исполнять (*--x*).

Идентификатор владельца и группы владельца входят в состав файлового дескриптора. Для кодировки прав доступа достаточно трех 3-битных позиционных кодов, которые также включаются в дескриптор.

Каждая активизация файлового дескриптора базовой ФС включает в себя проверку прав доступа. В ходе проверки определяется идентификатор владельца процесса, открывающего файл. Этот идентификатор сравнивается с идентификатором владельца файла и с идентификатором группы владельца файла. В зависимости от результатов сравнения определяется категория пользователя, открывающего файл, и выбирается соответствующий 3-битный код доступа. Режим доступа, запрашиваемый при открытии, сравнивается с кодом доступа и при несоответствии их происходит отказ в доступе.

Как мы отмечали выше, с точки зрения базовой ФС, каталоги практически ничем не отличаются от файлов, следовательно, все, что говорилось выше о правах доступа применимо и к каталогам.

Поскольку логическая ФС при поиске обращается к базовой ФС для чтения информации из каталогов, а базовая ФС производит при этом проверку прав доступа, то файл может быть доступен для данного пользователя только, если для него доступны все подкаталоги, входящие в путь к файлу.

Как определяется доступ к алиасам и косвенным файлам? Что касается алиасов, то ответ зависит от того, связываются ли права доступа с именем или с файлом. До сих пор мы говорили о том, что права доступа хранятся в файловом дескрипторе, поскольку физическому файлу соответствует единственный файловый дескриптор, права доступа к файлу по всем алиасам будут одинаковы. Возможно, однако, хранение прав доступа не в дескрипторе, а в элементе каталога, в этом случае разные алиасы могут обеспечивать разные права доступа к одному файлу. Возможны два варианта реализации прав доступа к косвенному файлу. В первом варианте в псевдодескрипторе, содержащем ссылку, указываются права доступа, как для обычного файла, при соответствии конкретного режима доступа этим правам проверка прав доступа заканчивается. Во втором случае проверка прав доступа продолжается и для всех каталогов, входящих в путь, указанный в ссылке.

## **7.6. Физическая структура файлов**

До этого уровня ФС оперировала с виртуальными адресами в файле – относительными номерами байт или записей. Уровень физической структуры выполняет перевод реальных адресов в физические адреса на носителе. На этот уровень ложатся все задачи, связанные с управлением и распределением реальной внешней памяти. Внешняя память может включать в себя как устройства произвольного доступа (в дальнейшем –

диски), так и устройства последовательного доступа (в дальнейшем – ленты). Мы сосредоточимся здесь только на дисковой памяти, имея в виду то обстоятельство, что задачи управления памятью на лентах составляют лишь узкое подмножество задач, возникающих при управлении дисками.

Несколько принципиально важных положений являются общими для любых способов управления дисковой памятью.

1) Дисковая память состоит из блоков, являющихся единицами распределения дискового пространства (например, секторов). Каждый блок имеет уникальный номер (адрес), его идентифицирующий. В каждый блок может быть записана любая информация достаточно сложной структуры, в том числе и содержащая ссылки на другие блоки.

2) Каждый физический диск описывается дескриптором диска, который содержит информацию о количестве и размере блоков на диске и о свободном пространстве на диске. Дисковый дескриптор записывается на известное заранее место на диске (чаще всего – в первый блок).

3) Каждый файл в составе своего дескриптора имеет план своего размещения (layout) на физическом пространстве диска.

4) Информация, записываемая на диск, может быть избыточной для обеспечения возможности ее восстановления при сбоях.

5) Дисковое пространство распределяется блоками фиксированной длины. Даже в тех дисковых архитектурах, которые допускают чтение/запись блоками переменной длины, размер единицы распределения, как правило, все равно фиксирован, например, дорожка. Возможно объединение в единицу распределения нескольких смежных блоков, такой прием носит название кластеризации (clustering), а порции распределения, состоящие из нескольких блоков, называются кластерами. Кластеризация может быть как симметричной – с заранее установленным размером кластера, так и асимметричной – с размером кластера, выбираемым для каждого распределения.



б) Поиск на диске управляющих структур ФС и свободных блоков может оказаться слишком времяемким. Поэтому те управляющие структуры, обращение к которым происходит наиболее часто, обычно копируются в оперативную память. Это создает дополнительные проблемы, связанные с надежностью функционирования ОС. При крахе системы изменения в управляющих структурах могут быть не перенесены из кеша на внешнюю память. Специальные системные процессы ОС обычно обеспечивают периодическое (например, раз в минуту) сохранение управляющих структур на внешней памяти.

Выше мы указывали на наличие в файловом дескрипторе плана размещения файла. Отметим, что поскольку дескриптор представляет собой одинаковую для всех файлов структуру данных, то размер его должен быть фиксированным. Не во всех случаях план размещения файла может иметь фиксированную длину. Если не удастся добиться фиксированной длины плана, то переменная его часть может быть вынесена из файлового дескриптора. Вид представления плана зависит от способа распределения файлам дисковой памяти.

Распределение дисковой памяти может быть классифицировано как смежное или не смежное. В первом случае файлу распределяется непрерывный участок внешней памяти, во втором – файл может быть разбросан по пространству диска.

Смежное распределение весьма привлекательно по нескольким причинам. Во-первых, план размещения в этом случае получается простейшим: он состоит только из номера начального блока и количества блоков. Во-вторых, и это более важно, смежное размещение обеспечивает высокую степень локализации обращений к дорожкам при обработке файла и, следовательно, высокую эффективность обмена.

Однако, и недостатки смежного распределения более чем серьезны. Во-первых, как и при распределении оперативной памяти блоками

переменной длины, здесь возникают все проблемы фрагментации пространства памяти (внешних дыр). Во-вторых, такое распределение требует, чтобы необходимый размер дискового пространства указывался при создании файла – эта информация не всегда может быть предоставлена пользователем, создающим файл. В-третьих, расширение файла за пределы установленного размера невозможно – для файлов с длительным сроком существования эта проблема может стать критической. Перечисленные недостатки могут быть не радикально преодолены, но могут быть несколько сглажены, если допустить, чтобы файл занимал не один, а несколько участков дискового пространства. При создании такого файла задаются требования к размеру первичного и вторичного распределения. Файлу выделяется непрерывный участок памяти в соответствии с первичным размером. Если далее оказывается, что файл не помещается в первичный участок, ему выделяется дополнительный участок (экстент – extent), размер которого задается требованием ко вторичному распределению. Экстент занимает непрерывную область на диске, но необязательно смежную с первичным участком. Затем может быть выделен второй экстент и т.д. План файла в его дескрипторе в этом случае представляется массивом пар "начальный адрес – длина", каждый элемент которого соответствует одному экстенту. Размер этого массива в дескрипторе ограничивает допустимое число экстенгов.

Очевидно, что смежное распределение является негибким. Это в первую очередь послужило причиной полного отказа от него в ОС Unix и в MS DOS. Однако, в связи с увеличением объемов дисковых накопителей вопросы повышения эффективности обмена приобретают все большее значение и локализация файлов является хорошим средством повышения этой эффективности. В последнее время наметилась тенденция к размещению файлов в смежных областях диска, пусть даже и в рамках

несмежной модели распределения (см., например описание ФС HPFS, NTFS, Veritas, JFS в Части II).

Практически все системы, использующие несмежную модель, имеют в своем наборе утилит такие, которые производят реорганизацию дисков, переписывая файлы таким образом, чтобы они располагались в смежных блоках.

Несмежное распределение допускает, чтобы файл состоял из большого числа участков (отдельных блоков) и план файла должен описывать размещение каждого из блоков файла. Можно выделить три основных подхода к представлению плана файла в несмежной модели:

- списки блоков;
- карта размещения;
- списки указателей на блоки.

Метод списков блоков предполагает, что в файловом дескрипторе содержится только указатель на первый выделенный файлу блок. В самом блоке на фиксированном месте содержится указатель на второй блок, во втором – на третий и т.д. Таким образом, блоки выстраиваются в односвязный линейный список. С учетом того, что список располагается на внешней памяти, этот метод может применяться только для файлов с последовательным доступом.

Метод карты размещения хорошо известен программистам в MS DOS по Таблице размещения файлов FAT. ФС для каждого диска поддерживает таблицу, каждая строка которой описывает один блок на диске. В списке связываются не сами блоки, а соответствующие им элементы карты. Карта размещения обеспечивает достаточно простой и эффективный метод управления распределением, но сосредоточение информации о размещении всех файлов в одной структуре данных делает эту структуру узким местом ФС с точки зрения безопасности.

Метод списков указателей на блоки предполагает наличие для каждого файла перечня номеров блоков, ему распределенных. Этот

перечень записывается в отдельный блок, специально выделяемый для этой цели, а файловый дескриптор содержит указатель на этот блок. Если в блоке не хватает места для всего перечня, то последний в нем номер адресует блок, содержащий продолжение перечня и т.д. При применении этого метода "в чистом виде" список блоков, как всякий линейный список, может обрабатываться только последовательно, доступ к блокам файла, имеющим большие виртуальные номера, может быть значительно замедлен. Поэтому на практике применяются различные модификации этого метода. Классическим примером такого подхода является ФС s5 первых версий ОС Unix [33].

Учет дискового пространства, выделенного файлу (план размещения), ведется в s5 следующим образом. Как показали исследования, на любом носителе всегда есть большое число файлов, объем которых не превышает 1 Кбайт. Для таких файлов нет никакой необходимости выделять отдельные блоки для размещения плана, а тем более – увязывать эти блоки в какие-либо списки. Поэтому непосредственно в файловом дескрипторе содержится массив из 10 номеров первых блоков файла. При размере блока 512 байт первые 5 Кбайт файла адресуются непосредственно из файлового дескриптора. Одиннадцатый элемент этого массива содержит адрес блока, в котором записано еще 128 номеров следующих блоков файла. Таким образом, доступ к следующим 64 Кбайтам файла производится путем косвенной адресации из дескриптора через этот блок адресов. Двенадцатый элемент содержит адрес блока, в котором записано еще 128 номеров блоков, каждый из которых адресует еще 128 блоков данных файла – это обеспечивает доступ к следующим 8 Мбайтам путем двухуровневой косвенной адресации. Наконец, тринадцатый элемент массива в дескрипторе обеспечивает доступ через трехуровневую косвенную адресацию еще к 2 Гбайтам файла. Структура плана размещения показана на рисунке 7.4.



"масштаба". Наряду с обязательной картой свободных блоков – самой "мелкомасштабной" может существовать карта свободных дорожек и карта свободных цилиндров. Наличие "крупномасштабных" карт позволяет ускорить поиск больших непрерывных свободных участков.

Списки свободных блоков организуются точно так же, как и списки блоков, принадлежащих файлу. Дескриптор диска содержит указатель на начало списка свободных блоков. Последовательная природа списка не оказывает влияния на эффективность его обработки, так как список свободных блоков может обрабатываться по дисциплине LIFO – выборка блока происходит из начала списка, и новый свободный блок также добавляется в начало списка. Изящный пример работы со свободными блоками показывает та же ФС s5, пример относится к категории "ленивых политик", свойственных этой ОС Unix.

В дескрипторе диска (суперблоке) содержится массив из 10 номеров свободных блоков. Поскольку суперблок в процессе работы хранится в оперативной памяти, первые 10 свободных блоков могут быть выбраны без обращений к внешней памяти. Если этот "запас" свободных блоков исчерпан, то используется последний элемент этого массива, который ссылается на свободный блок, содержащий еще 10 номеров свободных блоков. Эти новые номера заново заполняют массив в суперблоке и т.д. При необходимости включить в список новые свободные блоки их номера сохраняются в массиве суперблока, пока в нем есть свободные места. Если же свободные места исчерпаны, номера свободных записываются в свободный блок на внешней памяти, который включается в список, а массив в суперблоке освобождается.

## 7.7. Пример

Мы описали работу каждого уровня ФС по отдельности. Их совместное функционирование будет удобнее всего разобрать на конкретном примере. Рассмотрим такую программу:

```
1 main() {
2   filehandler xfile;
3   xfile = open ("/ivanow/work/testfile.tst",
WR_ONLY);
4   seek (xfile,1000);
5   write (xfile,"Example",7);
6   close (xfile);
7 }
```

Введенный нами тип данных `filehandler` предназначен для представления манипулятора файла.

Далее мы рассматриваем выполнение программы по шагам, причем в идентификации шагов первая цифра – номер строки исходного текста, вторая – собственно номер шага.

**Шаг 3.1.** Системный вызов `open` переключает контекст с процесса на ядро ОС. ОС передает этот вызов подсистеме логической организации, а та после соответствующих проверок – логической ФС.

**Шаг 3.2.** Логическая ФС начинает поиск файла по указанному пути, для чего выполняет синтаксический разбор строки имени файла. Поиск начинается с каталога `'/'` – корневого. Указатель на дескриптор корневого каталога у логической ФС уже имеется: дескриптор корневого

каталога может находиться на фиксированном месте на диске, или же указатель на него может храниться в дескрипторе диска и считываться в память вместе с дескриптором диска. (Также логическая ФС постоянно имеет в своем распоряжении и указатель на дескриптор рабочего каталога, так как поиск может начинаться и с него. Первоначально этот указатель указывает на дескриптор корневого каталога, а затем изменяется системными вызовами `setCurrentDirectory`). Теперь логическая ФС обращается к базовой ФС с запросом на активизацию дескриптора корневого каталога.

**Шаг 3.3.** Базовая ФС передает на следующие уровни адрес дескриптора корневого каталога. На уровне СУБВ проверяется, не считан ли уже блок, содержащий этот дескриптор, в буферную память. Для корневого каталога весьма велика вероятность того, что блок, содержащий его дескриптор, уже есть в памяти. Если это так, то СУБВ выбирает из кеша буфер, содержащий требуемый дескриптор. Если же такого буфера в кеше нет, то СУБВ находит в кеше свободный буфер, если нет и свободного буфера, освобождает буфер, используя для выбора, например, дисциплину LRU. Затем СУБВ формирует запрос на ввод и ставит его в очередь. На время выполнения обмена процесс, выдавший вызов, блокируется. Когда прерывание от устройства сигнализирует об окончании ввода, процесс продолжает выполнение на нижнем уровне иерархии ФС. Происходит возврат из СУБВ в верхние уровни, и базовая ФС получает дескриптор корневого каталога.

Заметим, что в некоторых случаях здесь может происходить две операции обмена и две блокировки процесса. Если СУБВ выбирает в кеше буфер для освобождения, а этот буфер "грязный", то сначала содержимое этого буфера выводится на диск, а уже затем в него вводится блок.

**Шаг 3.4.** Базовая ФС проверяет права доступа, находит свободное место в системной таблице открытых файлов и формирует в последней



дескриптор открытого файла для корневого каталога. Режим открытия в этом дескрипторе устанавливается RD\_ONLY, позиция файлового курсора – 0.

**Шаг 3.5.** Логическая ФС далее формирует запрос на чтение первого элемента каталога. Допустим, что элемент каталога состоит из 16-байтного имени и 4-байтного адреса. Базовая ФС получает, таким образом, запрос на 20 байт. Она передает на следующий нижний уровень дескриптор открытого файла и счетчик. Система физической организации по плану размещения файла и позиции файлового курсора определяет физический адрес блока, в котором находится требуемая информация, и передает запрос СУБВ. (Это определение может потребовать обращений к СУБВ для чтения информации о размещении файла).

**Шаг 3.6.** СУБВ находит требуемый блок в кеше или организует его ввод с диска (на время обмена процесс блокируется) и возвращает адрес буфера. Система физической организации выделяет из буфера нужные 20 байт и они возвращаются базовой ФС и далее – логической ФС. Базовая ФС увеличивает на 20 позицию файлового курсора.

Отметим, что в некоторых случаях для выполнения одного запроса система физической организации может формировать два и более запросов к СУБВ – если требуемая информация переходит из блока в блок.

**Шаг 3.7.** Логическая ФС сравнивает поле имени в полученном элементе каталога с первой составляющей строки поиска – *ivanov*. Скорее всего результат сравнения в первый раз будет отрицательным. Если имя не совпадает, логическая ФС формирует запрос на чтение следующего элемента каталога и повторяются шаги 5, 6 и 7. Поскольку файловый курсор модифицирован, будут читаться следующие 20 байт и скорее всего СУБВ найдет их уже в кеше.

**Шаг 3.8.** Когда элемент с именем *ivanov* будет найден, логическая ФС дает команды базовой ФС освободить дескриптор открытого файла для

корневого каталога и открыть подкаталог `ivanov`. Указатель на дескриптор подкаталога уже выбран логической ФС, далее открытие подкаталога происходит по тому же сценарию, что и корневого открытия каталога.

**Шаг 3.9.** Когда наконец активизируется файловый дескриптор для последней составляющей пути – файла `testfile.tst`, режим открытия устанавливается `WR_ONLY`. К этому моменту все каталоги, открывавшиеся в процессе поиска, уже закрыты. Указатель на элемент системной таблицы открытых файлов помещается в контекст процесса – в таблицу файлов процесса. Индекс элемента в этой таблице возвращается процессу в качестве манипулятора открытого файла и сохраняется в переменной `xFile`.

**Шаг 4.10.** Следующий системный вызов из программы – `seek`. По манипулятору файла ядро выбирает дескриптор открытого файла. Выполнение системного вызова ограничивается уровнем базовой ФС. Последняя устанавливает файловый курсор в позицию 1000, определяемую параметром вызова. Если это определяется спецификациями ФС, то позиция курсора сравнивается с размером файла и ограничивается этим размером. На более низкие уровни этот вызов не передается.

**Шаг 5.11.** Следующий системный вызов – `write`. По манипулятору выбирается дескриптор открытого файла. Базовая ФС передает на следующий уровень запрос на запись. Подсистема физической структуры вычисляет номер блока, в который должна быть произведена запись. Последующие действия, выполняемые этой подсистемой, зависят от того, выделен уже файлу этот блок или нет. В первом случае формируется запрос для СУБВ на чтение этого блока, СУБВ читает блок или находит соответствующий буфер в кеше. Во втором случае подсистема физической структуры находит свободный блок на диске. (Поиск свободного блока

может потребовать обращений к СУБВ для чтения информации о распределении свободного пространства.) СУБВ выделяет свободный буфер в кеше и связывает его с выделенным блоком. Подсистема физической структуры записывает на определенное место в буфер заданные 8 байт и формирует для СУБВ запрос на вывод блока. СУБВ пока только помечает буфер как "грязный".

**Шаг 6.12.** При выполнении системного вызова `close` базовая ФС копирует часть дескриптора открытого файла на диск (обращаясь для этого к нижним уровням ФС) и освобождает его место в таблице открытых файлов. Дальнейшие действия ФС зависят от того, придерживается она "активной" или "ленивой" дисциплины. "Активная" ФС просматривает весь план размещения файла и формирует для СУБВ запросы на поиск в буферном кеше блоков, принадлежащих файлу. Если такие блоки найдены в кеше, и они помечены как "грязные", СУБВ выводит их на диск и снимает с них пометку. Таким образом, при закрытии файла все обновления его данных записываются на диск. "Ленивая" ФС оставляет "грязные" блоки файла в кеше. Эти блоки попадут на диск, когда потребуется освободить занятые ими буферы кеша.

## **7.8. Целостность данных и файловой системы**

Контроль доступа является лишь одной из составляющих защиты файлов. Другими составляющими является обеспечение сохранения или восстановления целостности информации при аппаратных сбоях и при параллельном доступе к ней. Методы защиты от потерь целостности, происходящих по причине аппаратных и программных ошибок, можно подразделить на основывающиеся на избыточности и основывающиеся на копировании.

Избыточность, поддерживаемая программно, на уровне ОС, обеспечивается, как правило, только для особо важной системной информации: дескрипторов дисков и файлов, таблиц размещения, каталогов и т.п. При записи одной и той же информации в два разных блока на диске, даже если сбой произойдет во время записи, один из блоков будет содержать корректный вариант информации – то ли до ее изменения, то ли после. Избыточность может вноситься как простым дублированием управляющих структур данных, так и внесением избыточности в сами эти структуры. Так, в структуры данных, которые связываются в списки, часто вводятся дополнительные указатели, позволяющие получить доступ к одной и той же структуре данных разными путями. Причем, при штатной работе используется только один путь, а остальные могут быть использованы для восстановления списка при нарушении указателей основного пути. Например, если сделать список свободных блоков на диске двунаправленным, то по обратным прошивкам можно восстановить этот список при потере указателя на его начало в дескрипторе диска. Другой пример: алгоритмы функционирования логической и базовой ФС не нуждаются в том, чтобы имя файла, имеющееся в элементе каталога, дублировалось в дескрипторе файла. Однако такое дублирование чрезвычайно облегчит задачу восстановления доступа к файлам при разрушении системы каталогов.

Начнем с того, что весьма часто нарушение целостности происходит по вине пользователя – ошибочное удаление файлов или внесение в них неправильных изменений. Профилактикой этих ошибок является создание обратных копий (backup) файлов, к которым можно вернуться, если испорчена или уничтожена текущая копия. Концепция обратных копий может быть расширена до сохранения нескольких поколений версий файлов: при изменении файла создается его новая версия, а старая версия сохраняется. Создание новых версий может производиться разными способами:

- вручную – пользователь сам копирует файл, версию которого он хочет сохранить, под новым именем;
- специальными утилитами – некоторые программы (например, текстовые редакторы) автоматически создают новую версию файла, который они изменяют (ВАК-файлы, хорошо известные пользователям MS DOS);
- общей утилитой, которая проверяет версии всех файлов (или заданной группы файлов) и создает новые версии для изменившихся файлов;
- автоматически – самой ОС при открытии любого файла для записи.

Множественность версий может быть прозрачна для пользователя: все версии могут иметь одинаковое имя. Один элемент каталога может содержать массив ссылок на дескрипторы разных версий или дескрипторы версий могут быть увязаны в список. Естественно, что по умолчанию всегда выбирается последняя версия и только специальным системным вызовом обеспечивается возврат к предыдущим версиям. Количество сохраняемых версий должно ограничиваться: это может быть либо общее для всех файлов ограничение, либо устанавливаемое индивидуально для каждого файла. При превышении числом версий файла установленного предела ОС автоматически выполняет удаление самой старой версии.

Профилактикой случайного удаления файла может быть неполное удаление. Такое удаление предполагает сохранение физического файла на диске, а следовательно, и возможность его последующего восстановления. Элемент каталога, соответствующий не полностью удаленному файлу, может помечаться специальным признаком или файл может переноситься в специальный каталог для удаленных файлов. Такой подход должен обеспечиваться двумя системными вызовами: `deleteFile` для неполного

удаления и `purgeFile` – для полного. Неполное удаление создает, однако проблему дискового пространства: неполное удаление файла не освобождает места на диске. ОС может периодически проверять срок хранения не полностью удаленных файлов и физически удалять те из них, которые не были восстановлены за установленный срок, или же выполнять такую проверку только тогда, когда дает о себе знать нехватка дискового пространства.

Средством, практически гарантирующим минимизацию потерь при порче информации, является восстановление по резервной копии. Без рекомендаций о сохранении копий своих файлов на дискетах не обходится ни одно руководство по работе на персональных компьютерах, но выполнение этих рекомендаций требует от пользователя некоторой самодисциплины, которая в массе пользователей персональных ЭВМ, увы, не свойственна. В любых неперсональных системах и при работе над любыми неперсональными проектами копирование информации является не возможностью по выбору, а необходимостью. Для хранения резервных копий применяются магнитные ленты – носители, отличающиеся большим временем доступа, но и большой емкостью и малой стоимостью. Отчасти забытые во времена "персонального бума", они сейчас опять восстанавливают свои позиции архивных носителей.

Схема копирования должна обеспечивать минимизацию времени поиска и восстановления информации как при локальных (в пределах одного или нескольких файлов) потерях, так и при полном разрушении структур ФС. Рекомендуемая схема включает в себя несколько уровней выгрузки (`dump`), обычно – три: полные, повторные и инкрементные выгрузки. Полная выгрузка выполняется редко (например, раз в год) и включает в себя все имеющиеся на диске файлы. Повторные выгрузки производятся несколько чаще (например, ежемесячно); в них включаются только файлы, изменившиеся со времени предыдущей повторной или

полной выгрузки. Инкрементные выгрузки выполняются часто (ежедневно) и включают в себя файлы, изменившиеся со времени предыдущей выгрузки – инкрементной или повторной. Для возможности восстановления состояния информации на начало текущего дня при полном крахе системы необходимо сохранять:

- носитель последнего полного дампа;
- все носители повторных дампов, сделанных со времени последнего полного;
- все носители инкрементных дампов, сделанных со времени последнего повторного.

В таком порядке и используются носители при выполнении восстановления.

Альтернативная схема архивирования, известная как "функция линейки" (ruller function), продиктована стремлением минимизировать используемое число архивных носителей. Если предположить, что архивирование делается ежедневно, и в нашем распоряжении имеется  $T$  носителей, то период между двумя полными дампами должен составлять  $2^T$  дней. В день  $n$  в таком периоде для использования выбирается носитель с номером  $t$ , причем  $t$  должно обеспечивать максимальное значение  $2^t$ , на которое номер дня  $n$  делится без остатка. Копируются все файлы, изменившиеся со времени последнего использования этого носителя.

Создание архивных копий должно выполняться системными утилитами, которые могут также вести учет архивированных файлов: поддерживать на внешней памяти (и сохранять на отдельных архивных носителях) таблицу файлов с указанием для каждого файла сведений о том, когда сделана его архивная копия и на каком носителе она находится. Наличие такой таблицы позволяет выполнить быстрое восстановление при локальных сбоях.

При наличии аппаратной избыточности в дисковой памяти ФС может (как правило, в качестве дополнительной возможности) поддерживать правила чтения/записи, обеспечивающие отказоустойчивость и возможность восстановления информации, а также повышение быстродействия дискового ввода-вывода за счет параллельной работы двух и более дисков.. Эти правила известны как технологии RAID (Redundant array of inexpensive disks – избыточный массив недорогих дисков). Определены 5 уровней технологии RAID

RAID 0 – разделение данных по дискам; отдельные блоки данных записываются на разные свободные диски. Надежная защита не обеспечивается, так как при выходе из строя одного диска все данные могут быть потеряны.

RAID 1 – дублирование дисков; данные записываются на оба диска по одному и тому же каналу. При порче одного из дисков данные могут быть восстановлены по другому, но при сбое канала все данные будут потеряны.

RAID 2 – разделение данных по дискам с чередованием битов и контрольной суммой; метод неэффективный как по надежности, так и по быстродействию. На практике не применяется.

RAID 3 – разделение данных по дискам с чередованием битов и проверкой четности; метод более надежный, чем RAID 2, но столь же медленный. На практике не применяется.

RAID 4 – разделение данных по дискам с чередованием блоков и проверкой четности; быстродействие выше, чем RAID 3, но все же недостаточно для практического применения.

RAID 5 – разделение блоков по дискам, чередование блоков, распределенная четность; требует не менее 3 дисков, каждый следующий блок данных помещается на другом физическом диске. В массиве блоков, имеющих одинаковые физические адреса на разных дисках, один сектор



используется как контрольный. Размещение контрольного сектора "скользящее", как показано на рисунке 7.5. Контрольный сектор содержит результат операции XOR между содержимым секторов с тем же адресом всех остальных дисков. При потере информации на одном из дисков эта информация может быть восстановлена путем выполнения XOR между контрольным сектором и секторами всех уцелевших рабочих дисков.

Адрес	Диск 1	Диск 2	Диск 3	Диск 4	Диск 5
1	блок 1	блок 2	блок 3	блок 4	контроль
2	контроль	блок 5	блок 6	блок 7	блок 8
3	блок 9	контроль	блок 10	блок 11	блок 12
4	блок 13	блок 14	контроль	блок 15	блок 16
5	блок 17	блок 18	блок 19	контроль	блок 20
6	блок 21	блок 22	блок 23	блок 24	контроль

Рисунок 7.5 Технология RAID 5

Промышленное применение в настоящее время имеют технологии RAID 0, 1 и 5.

Резервное копирование и даже аппаратная избыточность позволяют уменьшить потери, но не избежать их полностью. В предыдущей главе мы упоминали о том, что все современные файловые системы используют кеширование дискового ввода-вывода, а, следовательно, в них возникает ситуация отложенной или ленивой записи (lazy write), когда данные, уже записанные процессом в файл, на самом деле находятся в буферной оперативной памяти. При сбое системы такие данные могут быть потеряны. Наиболее опасно то, что отложенной записи подвергаются не только данные, но и метаданные файловой системы: дескрипторы дисков и файлов, каталоги и т.п. При потере изменений в метаданных недоступными могут стать все данные на диске. Поэтому в тех ФС, к целостности которых предъявляются высокие требования, ведется протоколирование операций над метаданными – запись информации по повторению или откату операции над метаданными в файл протокола (log)

или журнала (journal). При сбое системы восстановление или откат операций производится по журналу. Поскольку журнал также подвергается отложенной записи, периодически производится фиксация контрольной точки – принудительная запись журнала на диск. При сбое системы, таким образом, гарантируется восстановление до последней контрольной точки.

Причиной нарушения целостности может являться одновременный доступ к файлу двух и более процессов. Для предотвращения таких нарушений ОС накладывает "замки" (lock) на файлы: замок разделяемого доступа для файла, открытого для чтения, и замок монопольного доступа для файла, открытого для записи. Замок может входить в дескриптор открытого файла. Если новый замок конфликтует с замком, наложенным ранее начавшейся операцией, то новая операция должна быть заблокирована. Обычно ОС не поддерживает сложного управления синхронизацией операций, ограничиваясь замками файлов. Наиболее часто в ОС применяется так называемая двухфазная дисциплина, требующая, чтобы замок на файл накладывался при его открытии и снимался при его закрытии. Замыкаться могут не только файлы, но и целые диски или каталоги или наоборот – отдельные записи или байты файла. Задача обеспечения синхронизации является частным случаем задачи взаимного исключения, подробно рассматриваемой в главе 8.

Большинство ОС обеспечивает только разделение доступа к каждому отдельному файлу и восстановление целостности метаданных ФС, перепоручая управление сложными транзакциями и целостностью пользовательских данных промежуточному программному обеспечению – менеджерам транзакций (IBM CICS, MS Transaction Server и др.) и системам управления базами данных (Oracle, IBM DB2 и др.). В распоряжении процессов должны быть средства API для формирования структуры транзакций – системные вызовы или обращения к тому

программному пакету, в среде которого функционирует процесс. Примером такого средства могут быть предложения языка SQL: COMMIT (совершить) и ROLLBACK (откат).

## **7.9. Загружаемая файловая система**

Задачи переносимости программного обеспечения, в том числе и системного, и функционирования программных изделий в среде распределенной обработки данных включают в себя требования к обеспечению единого пользовательского интерфейса различных ФС. Вытекающим отсюда следствием может быть совместное использование в одной операционной среде томов данных, обслуживаемых различными файловыми системами.

Иерархическая структура ФС дает возможность провести в иерархии уровней некоторую разграничительную линию, выше которой будут располагаться абстрактная ФС – структуры данных и алгоритмы, общие для любых ФС, а ниже – конкретная ФС со специфическими структурами и алгоритмами. Наиболее вероятно эта граница может проходить в базовой ФС, связывающей логическое представление файла с его физическим представлением. Такой подход был впервые применен в ОС Unix в связи с концепцией сетевых файловых систем. Проведение разграничительной линии на уровне базовой ФС вызывает расщепление дескриптора открытого файла на две части. Первая часть имеет общий для всех файлов формат и содержит общие для всех файлов поля, обработка которых не зависит от конкретной ФС. Это могут быть поля типа файла, размера, временные отметки и другие данные. Вторая часть – частный дескриптор – для конкретной ФС. В ней содержится план размещения файла, сведения об организации и т.д. Общая таблица томов в ядре указывает для каждого

тома тип конкретной ФС, управляющей этим томом. Для каждого типа конкретной ФС ядро хранит таблицу операций – таблицу входных точек процедур, выполняющих для данной ФС стандартные функции (open, close, read и т.д.). При обращении к ФС абстрактная ФС, выполнив общие операции, определяет том и тип конкретной ФС на этом томе, выбирает соответствующую конкретной ФС таблицу входных точек и вызывает требуемую процедуру для конкретной ФС.

В системах с многоуровневыми драйверами драйвер ФС является одним из уровней, которые проходят данные на пути от процесса к устройству. При этом драйвер ФС не зависит не только от верхнего уровня – абстрактной ФС, но и от нижнего уровня – аппаратных драйверов.

Если конкретная ФС удовлетворяет спецификациям, диктуемым общей ФС, то конкретная ФС может быть загружаемой – включаемой в ядро при установке тома, управляемого конкретной ФС.

## **КОНТРОЛЬНЫЕ ВОПРОСЫ**

1. Поясните различие между виртуальным и физическим файлом.
2. Охарактеризуйте основные компоненты иерархической модели файловой системы. Какие преимущества дает иерархическая модель?
3. В чем различие между байториентированными и записеориентированными файлами? Назовите достоинства и недостатки той и другой модели.
4. В чем отличие логической структуры каталогов в MD DOS – Windows – OS/2 от структуры каталогов в Unix?
5. В чем достоинства и недостатки отделения дескриптора файла от элемента каталога?

6. Какую информацию о файле должен содержать его дескриптор, хранимый в файловой системе? Какую информацию должен содержать дескриптор открытого файла?

7. В чем сходство и различие каталогов и файлов (на логическом и на физическом уровнях)?

8. В чем сходство и различие алиасов и косвенных файлов?

9. Обязательно ли закрытие файла при завершении открывшего его процесса? Обязательна ли запись данных файла на диск при закрытии файла?

10. В чем отличие смежного размещения файлов в современных файловых системах от смежного размещения файлов в старых файловых системах?

11. Какими методами может быть обеспечено преимущественно смежное размещение файла на внешней памяти?

12. В чем отличие целостности файловой системы от целостности данных? Какую целостность, и какими методами обеспечивают современные файловые системы?

## **Глава 8. Параллельное выполнение процессов**

### **8.1. Постановка проблемы**

Параллельными называются процессы, в которых "интервалы времени выполнения перекрываются за счет использования разных ресурсов одной и той же вычислительной системы или за счет перераспределения одного и того же набора ресурсов." [8] При рассмотрении параллельности мы несколько расширим (только в пределах этой главы) понятие процесса: будем понимать под процессом любую

последовательность действий. Процесс у нас имеет строго последовательную структуру, он выполняется операция за операцией, команда за командой. Но в один и тот же момент времени в системе могут выполняться несколько таких последовательностей. Такое расширение понимания процесса позволит нам включить в рассмотрение и такие последовательности действий, которые формально процессами не являются: отдельные нити одного процесса, модули ядра ОС, обработчики прерываний, процессы на внешних устройствах и т.д. С концептуальной точки зрения не имеет значения, является ли одновременность выполнения реальной (достигаемой за счет аппаратного распараллеливания) или виртуальной (достигаемой за счет разделения времени), хотя это различие, как мы увидим ниже, может сказываться на механизмах реализации.

Процессы, которые мы рассматриваем в этой главе, являются слабо связанными. Это означает, что за исключением достаточно редких моментов явной связи процессы выполняются независимо друг от друга. Взаимная независимость процессов запрещает нам при решении задач управления параллельным выполнением делать какие-либо предположения о соотношении скоростей выполнения процессов. За счет перераспределения ресурсов эти скорости вообще не могут считаться постоянными. Единственным оправданным предположением в этом отношении может быть предположение о наихудшем (наименее удобном для нас) соотношении скоростей.

Поскольку параллельные процессы разделяют ресурсы вычислительной системы, задачи управления параллельным выполнением есть задачи управления ресурсами. Как и в случае монопольно используемых ресурсов, методы решения этих задач составляют широкий спектр от самых консервативных (требующих значительного снижения уровня мультипрограммирования) до самых либеральных (допускающих параллельное выполнение большого числа процессов).

Основной задачей управления параллельным выполнением является задача взаимного исключения (mutual exclusion, сокращенно – mutex): два процесса не могут выполнять одновременный доступ к разделяемым ресурсам. Обратим внимание на то обстоятельство, что взаимное исключение обеспечивается уже на аппаратном уровне. Если процессор одновременно получает два запроса, то он выполняет их последовательно по тем или иным правилам арбитража. Каждая процессорная команда обладает свойством атомарности: она или выполняется полностью, или вообще не выполняется. Так же атомарно каждое обращение к памяти. Арбитражные свойства аппаратуры являются основой для построения более сложных механизмов взаимного исключения.

Последовательность операций в процессе, которая выполняет такую единицу работы (транзакцию) с разделяемым ресурсом, во время которой никакой другой процесс не должен иметь доступа к этому ресурсу, составляет критическую секцию. Чтобы процесс мог обеспечить безопасность выполнения своей критической секции, в его распоряжении должны быть средства ("скобки критической секции") которыми он эту критическую секцию обозначит и защитит. Назовем эти скобки `csBegin` и `csEnd` и посвятим дальнейшее рассмотрение взаимного исключения механизмам реализации этих скобок. Для доказательства правильности каждой реализации для нее необходимо показать следующее:

- в любой момент времени не более, чем один процесс может находиться в своей критической секции;
- решение о том, какому процессу первому надлежит войти в критическую секцию, не может откладываться до бесконечности;
- остановка процесса вне своей критической секции не влияет на другие процессы.

В некоторых случаях возникает необходимость в приостановке выполнения процесса до наступления некоторого внешнего по отношению к

нему события – такая задача называется задачей синхронизации. Если рассматривать события как ресурсы (потребляемые), то задача синхронизации является частным случаем задачи взаимного исключения. Как частный случай, она допускает частные решения.

Типичным примером задачи синхронизации является обеспечение выполнения "графа синхронизации", подобного тому, который представлен на рисунке 8.1. Граф синхронизации задает порядок выполнения действий. В программной реализации мы можем представлять действия A, B, ..., I как отдельные процессы и выполнять явную синхронизацию или использовать неявную синхронизацию, представляя в виде процессов тройки ABC, DEF, GHI или ADG, BEH, CFI, и вводить явные точки синхронизации внутри процессов.

Более сложные задачи синхронизации, такие как "читатели–писатели" и "производители–потребители" мы рассмотрим отдельно ниже.

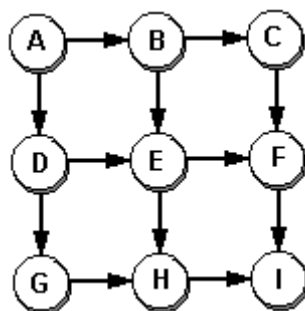


Рисунок 8.1 Пример графа синхронизации

## 8.2. Взаимное исключение запретом прерываний

Большинство компьютерных архитектур предусматривают в составе своей системы команд команды запрета прерываний (иногда – селективного запрета). В микропроцессорах Intel-Pentium, например, такими командами являются CLI (запретить прерывания) и STI (разрешить прерывания). Такие команды и могут составить "скобки критической секции": запрет прерываний при входе в критическую секцию



и разрешение – при выходе из нее. Поскольку вытеснение процесса возможно только по прерыванию, процесс, находящийся в критической секции, не может быть прерван. Этот метод, однако, обладает большим числом недостатков:

- пока процесс находится в критической секции, не могут быть обработаны никакие внешние события, в том числе и события, требующие обработки в реальном времени;
- исключаются не только конфликтующие критические секции, которые могут обращаться к разделяемым данным, но и все другие процессы вообще; дисциплина, таким образом, является весьма консервативной;
- процесс, находящийся внутри критической секции, не может перейти в ожидание (кроме занятого ожидания), так как механизм ожидания обеспечивается прерываниями;
- программист должен быть весьма внимателен к тому, чтобы все возможные варианты выхода из критической секции обеспечивались разрешением прерываний;
- вложение критических секций невозможно;
- обеспечение критической секции запретом прерываний базируется на аппаратной атомарности команд, оно неприменимо в мультипроцессорных системах с реальной параллельностью, так как программа, выполняющаяся на одном процессоре, не может запретить прерывания другого процессора.

### **8.3. Взаимное исключение через общие переменные**

Следующая группа решений базируется на непрерываемости памяти. Представляя эти алгоритмы, мы в основном следуем первоисточнику [11]

и приводим в качестве примеров как правильные, так и неправильные или неудачные варианты решений.

Почти все примеры мы даем для двух процессов с номерами 0 и 1, их нетрудно обобщить на произвольное число процессов.

#### Вариант 1: общая переменная исключения.

Введем булевскую переменную `mutex`, которая должна получать значение `true` (1), если входение в критическую секцию запрещено, или `false` (0), если входение разрешено. Попытка организовать "скобки критической секции" представлена следующим программным кодом:

```
1    static char mutex = 0;
2    void csBegin ( void ) {
3        while ( mutex );
4        mutex = 1;
5    }
6    void csEnd ( void ) {
7        mutex = 0;
8    }
```

При вхождении в функцию `csBegin` процесс попадает в цикл ожидания (строка 3), в котором находится до тех пор, пока состояние переменной исключения не разрешит ему войти в критическую секцию. Выйдя из этого цикла, процесс устанавливает эту переменную в 1, запрещая тем самым другим процессам входить в их критические секции. Процесс, который выполнялся в критической секции, при выходе из последней сбрасывает переменную исключения в 0, разрешая этим другим процессам входить в их критические секции.

Это решение базируется на непрерывности доступа к памяти – к переменной `mutex`, но оно является НЕПРАВИЛЬНЫМ. Рассмотрим такой случай. Пусть процесс А вошел в свою критическую секцию и установил `mutex=1`. Пока процесс А выполняется внутри своей критической секции, два других процесса – В и С – также подошли к своим критическим секциям и обратились к функции `csBegin`. Поскольку переменная `mutex` установлена в 1, процессы В и С зацикливаются в строке 3 кода функции `csBegin`. Когда процесс А выйдет из своей критической секции и установит `mutex=0`, другой процесс, например В, выйдет из цикла строки 3. Но имеется вероятность того, что прежде, чем процесс В успеет выполнить строку 4 кода и этим запретить вход в критическую секцию другим процессам, выйдет из цикла строки 3 и процесс С. Таким образом, два процесса – В и С – входят в критическую секцию, задача взаимного исключения не выполняется.

Хотя это решение и не выполняет взаимного исключения, оно может быть приемлемо для решения некоторых частных задач. Например, граф синхронизации, представленный на рисунке 8.1, может быть обеспечен, если мы введем массив `done`, каждый элемент которого свяжем с определенным событием. События пронумерованы, и номер события является параметром функции, сигнализирующей о завершении события, – `finish` и функции ожидания события – `waitFor`:

```
1    static char done[9]
2        = {0,0,0,0,0,0,0,0,0};
3    void finish ( int event ) {
4        done[event] = 1;
5    }
6    void waitFor ( int event ) {
```

```
7     while ( ! done[event] );
8 }
```

Теперь работа процессов может быть синхронизирована таким образом (функциями типа `workX()` представлена работа, выполняемая процессом X):

```
processA() {
    /* работа процесса А */
    workA();
    /* отметка о завершении процесса А */
    finish(0);
}

processB() {
    /* ожидание завершения процесса А */
    waitFor(0);
    /* работа процесса В */
    workB();
    /* отметка о завершении процесса В */
    finish(1);
}

. . .

processE() {
    /* ожидание завершения процесса В */
    waitFor(1);
    /* ожидание завершения процесса D */
    waitFor(3);
    /* работа процесса Е */
    workE();
    /* отметка о завершении процесса Е */
    finish(4);
}
```

```
    finish(4);  
}  
.  
.  
.
```

Можно сократить запись, например, так (используя естественную последовательность, заложенную в строках графа):

```
processABC() {  
    workA(); finish(0);  
    workB(); finish(1);  
    workC(); finish(2);  
}  
processDEF() {  
    waitFor(0); workD(); finish(3);  
    waitFor(1); workE(); finish(4);  
    waitFor(2); workF(); finish(5);  
}  
processGHI() {  
    waitFor(3); workG(); finish(6);  
    waitFor(4); workH(); finish(7);  
    waitFor(5); workI(); finish(8);  
}
```

или иным образом (запишите самостоятельно) – с использованием последовательности в столбцах.

## ВАРИАНТ 2: ПЕРЕМЕННАЯ-ПЕРЕКЛЮЧАТЕЛЬ

Введем общую переменную `right`, значением ее будет номер процесса, который имеет право входить в критическую секцию.

Реализация "скобок критической секции" для двух процессов с номерами 0 и будет следующей:

```
1    static int right = 0;
2    void csBegin ( int proc ) {
3        while ( right != proc );
4    }
5    void csEnd( int proc ) {
6        if ( proc == 0) right = 1;
7        else right = 0;
8    }
```

Процесс, вызвавший функцию `csBegin`, закидывается до тех пор, пока не получит права на вход. Разрешение входа производится другим процессом при выходе его из своей критической секции.

Данный алгоритм обеспечивает разделение процессов 0 и 1. Два процесса могут одновременно войти в функцию `csBegin`, но при этом они только читают переменную `right`. Одновременное вхождение двух процессов в функцию `csEnd` невозможно. При обобщении алгоритма на большее число процессов первый процесс переключает право на второй, второй – на третий и т.д., последний – на первый. Если процессы используют разные группы разделяемых данных, то каждая группа может быть защищена своим переключателем, таким образом, не запрещается одновременный доступ к разным разделяемым данным. Это делает политику более либеральной, но при наличии двух и более групп разделяемых данных возможно возникновение тупика, так как группа разделяемых данных – тот же монополярный ресурс. Для предотвращения тупика возможно введение иерархии ресурсов, как было описано в главе 5.

Существенный недостаток этого алгоритма в том, что он жестко определяет порядок, в котором процессы входят в критическую секцию. В нашем примере для двух процессов процессы 0 и 1 могут входить в нее только поочередно. Если предположить, что скорости процессов существенно разные, например, процессу 0 требуется вдвое чаще входить в критическую секцию, чем процессу 1, то частота вхождения процесса 0 снизится до частоты процесса 1, причем снижение скорости процесса 0 будет обеспечиваться за счет занятого ожидания в строке 3. Таким образом, не выполняется пункт 3 условия правильности решения: если один из процессов остановится вне своей критической секции, то он заблокирует все остальные процессы.

### Вариант 3: неальтернативные переключатели.

Введем для каждого процесса свою переменную, отражающую его нахождение в критической секции. Эти переменные сведены у нас в массив `inside`. Элемент массива `inside[i]` имеет значение 1, если *i*-й процесс находится в критической секции, и 0 – в противном случае.

Для примеров этого варианта введем функцию, определяющую номер процесса-конкурента:

```
int other (int proc ) {  
    if ( proc == 0 ) return 1;  
    else return 0;  
}
```

Первое решение в этом варианте:

```
1    static char inside[2] = { 0,0 };  
2    void csBegin ( int proc ) {  
3        int competitor; /* конкурент */
```

```

4      competitor = other ( proc );
5      while ( inside[competitor] );
6      inside[proc] = 1;
7  }
8  void csEnd (int proc ) {
9      inside[proc] = 0;
10 }

```

Здесь и во всех последующих решениях параметр `proc` функций `csBegin` и `csEnd` – номер процесса, желающего войти в свою критическую секцию или выйти из нее.

Процесс находится в занятом ожидании (строка 5) до тех пор, пока его конкурент находится в своей критической секции. Когда конкурент снимает свой признак пребывания в критической секции, наш процесс устанавливает свой признак (строка 6) и таким образом запрещает вход в секцию конкуренту.

Решение, однако, не гарантирует взаимного исключения. Возможен случай, когда два процесса одновременно выполняют каждую строку 5 своего кода и, следовательно, войдут в свои критические секции одновременно.

В следующем решении мы меняем местами установку своего признака входа и проверку признака конкурента:

```

1      static char inside[2] = { 0,0 };
2      void csBegin ( int proc ) {
3          int competitor;
4          competitor = other ( proc );
5          inside[proc] = 1;

```



```

6      while ( inside[competitor] );
7      }
8      void csEnd (int proc ) {
9          inside[proc] = 0;
10     }

```

Теперь процессы не могут одновременно войти в критические секции. Но возникает другая опасность: если процессы одновременно выполняют строку 5, то они заблокируют друг друга, так как оба процесса уже установят свои признаки, но будут зациклены в занятом ожидании, выйти из которого им не разрешит установленный признак конкурента.

Новое решение:

```

1      static char inside[2] = { 0,0 };
2      void csBegin ( int proc ) {
3          int competitor;
4          competitor = other ( proc );
5          do {
6              inside[proc] = 1;
7              if ( inside[competitor] ) inside[proc] = 0;
8          } while ( ! inside[proc] );
9      }
10     void csEnd (int proc ) {
11         inside[proc] = 0;
12     }

```

Процесс устанавливает свой признак вхождения (строка 6). Но если он обнаруживает, что признак вхождения конкурента тоже установлен (строка 7), то он свой признак сбрасывает. Эти действия будут

повторяться до тех пор, пока наш процесс не сохранит свой признак взведенным (строка 8), а это возможно только в том случае, если признак конкурента сброшен.

Это решение не может быть принято вот по какой причине. Возможно такое соотношение скоростей процессов, при котором они будут одновременно выполнять строку 7, и одновременно сбрасывать свои признаки. Такая "чрезмерная уступчивость" процессов приведет к бесконечному откладыванию решения о входе в критическую секцию.

### АЛГОРИТМ ДЕККЕРА

Эффективное и универсальное решение проблемы взаимного исключения носит название алгоритма Деккера и выглядит для двух процессов таким образом:

```
1  static int right = 0;
2  static char wish[2] = { 0,0 };
3  void csBegin ( int proc ) {
4      int competitor;
5      competitor = other ( proc );
6      while (1) {
7          wish[proc] = 1;
8          do {
9              if ( ! wish[competitor] ) return;
10             }
11             while ( right != competitor );
12             wish[proc] = 0;
13             while ( right == competitor );
14         }
```

```

15     }
16     void csEnd ( int proc ) {
17         right = other ( proc );
18         wish[proc] = 0;
19     }

```

Алгоритм предусматривает, во-первых, общую переменную `right` для представления номера процесса, который имеет преимущественное (но не абсолютное) право на вход в критическую секцию. Во-вторых, массив `wish`, каждый элемент которого соответствует одному из процессов и представляет "желание" процесса войти в критическую секцию. Процесс заявляет о своем "желании" войти в секцию (строка 7). Если при этом выясняется, что процесс-конкурент не выставил своего "желания" (строка 9), то происходит возврат из функции, т.е. процесс входит в критическую секцию независимо от того, кому принадлежало преимущественное право на вход. Если же в строке 9 выясняется, что конкурент тоже выставил "желание", то проверяется право на вход (строка 10). Если право принадлежит нашему процессу, то повторяется проверка "желания" конкурента (строки 8 - 10), пока оно не будет отменено. Конкурент вынужден будет отменить свое "желание", потому что он в этой ситуации перейдет к строке 11, где процесс, не имеющий преимущественного права, должен это сделать. После отмены своего желания процесс ждет, пока преимущественное право не вернется к нему (строка 12), а затем вновь повторяет заявление "желания" и т.д. (строки 6 - 13). Таким образом, процесс в функции `csBegin` либо повторяет цикл 7 - 14, либо выходит из функции и входит в критическую секцию (10).

При выходе из критической секции (функция `csEnd`) процесс передает преимущественное право входа конкуренту (строка 16) и отказывается от своего желания (строка 17).

По собственному опыту признаем, что понимание этого алгоритма дается не очень просто. Рекомендуем для лучшего его понимания записать в две колонки две копии функции `csBegin`, соответствующие двум процессам, и промоделировать ход их параллельного выполнения с разными скоростями и разными сдвигами в фазах выполнения между процессами.

Приведем также обобщение алгоритма Деккера на N процессов:

```
1  static char wish[N+1] = { 0, ..., 0 };
2  static char claimant[N+1] = { 0, ..., 0 };
3  static int right = N;
4  void csBegin ( int proc ) {
5      int i;
6      claimant[proc] = 1;
7      do {
8          while ( right != proc ) {
9              wish[proc] = 0;
10             if(!claimant[right]) right=proc;
11         }
12         wish[proc] = 1;
13         for (i = 0; i<N; i++ )
14             if ((i!=proc) && wish[i]) break;
15     }
16     while (i<N);
17 }
18 void csEnd ( int proc ) {
19     right = N;
20     wish[proc] = claimant[proc] = 0;
21 }
```

Ограничимся здесь только общими замечаниями к этому алгоритму. Процессы нумеруются от 0 до  $N-1$ . Мы вводим два массива для переменных состояния, размеры массивов на 1 больше числа процессов. Последние элементы каждого из массивов соответствуют несуществующему  $N$ -му процессу, который используется как абстрактный "другой" процесс. Понятие "конкурент" здесь заменяется понятием "претендент" (claimant). Процесс становится претендентом, входя в функцию csBegin (строка 6). В отличие от "желания" "претензия" процесса не снимается до тех пор, пока она не будет удовлетворена (строка 18). Если преимущественное право на вход в критическую секцию принадлежит другому процессу, но этот другой процесс не является претендентом, то наш процесс забирает это право себе (строки 7 - 11). При выполнении этих действий наш процесс, однако, отказывается от своего "желания", давая тем самым возможность участвовать в состязании за захват секции другим процессам (строка 9). Получив право, процесс заявляет о своем "желании" (строка 12). В последующем цикле for проверяются "желания" других процессов (строки 13 - 14). Если есть другие "желающие", то повторяется получение права и т.д. (строки 7 - 15). Если же в цикле for другие желающие не выявлены (строка 15), наш процесс входит в критическую секцию. При выходе из секции процесс сбрасывает свои "претензию" и "желание" (строка 18) и передает право несуществующему  $N$ -му процессу (строка 19).

## АЛГОРИТМ ПИТЕРСОНА

Более компактная и изящная модификация алгоритма Деккера известна как алгоритм Питерсона. Вот его вариант для двух процессов:

```

1  static int right;
2  static char wish[2] = { 0,0 };
3  void csBegin ( int proc ) {
4      int competitor;
5      if ( proc == 0 ) competitor = 1;
6      else competitor = 0;
7      wish[proc] = 1;
8      right = competitor;
9      while ( wish[competitor] && ( right ==
10 competitor );
11 }
12 void csEnd ( int proc ) {
13     wish[proc] = 0;
14     }

```

При входе в критическую секцию процесс заявляет о своем "желании" (строка 7) и отказывается от своего преимущественного права (строка 8). Процесс будет ожидать, если его конкурент заявил свое "желание" и имеет преимущественное право (строка 9). Если нет интереса конкурента или если независимо от интереса конкурента наш процесс имеет преимущественное право, то наш процесс входит в критическую секцию. Если наш процесс отказался от своего права в строке 8, то как же это право может к нему вернуться? Право нашего процесса может быть восстановлено конкурентом, когда последний тоже войдет в функцию csBegin своего кода и выполнит строку 8. При выходе из критической секции процесс просто снимает свой интерес и тогда его конкурент, возможно, ожидающий в строке 8, получает возможность выхода из цикла строки 9 по первой части условия.

Общие положительные свойства алгоритмов, основывающихся на неальтернативных переключателях (Деккера и Питерсона), следующие:

- они корректны как для одно-, так и для многопроцессорных систем;
- они либеральны, так как позволяют более быстрым процессам входить в свои критические секции чаще, чем медленным;
- они не ограничивают количество обслуживаемых ими процессов;
- они позволяют процессам сколь угодно долго задерживаться вне своей критической секции.

Но существуют и сложности:

- решения не просты для понимания и ошибиться в их реализации очень легко;
- процессы используют занятое ожидание при входе в критическую секцию.

## 8.4. Команда `testAndSet` и блокировки

Взаимное исключение при помощи переменных-переключателей базируется на атомарности обращений к памяти. Как мы показали выше, это делает решение универсальным как для одно-, так и для многопроцессорных систем. Но большинство архитектур компьютеров имеет в составе своей системы команд специальные команды с расширенной атомарностью обращений к памяти, при помощи которых можно реализовать взаимное исключение и быстрее, и проще. Общее название таких команд: `testAndSet` – проверить и установить. Действия такой команды могут быть описаны функцией:

```
int atomic testAndSet ( char *lock ) {  
    char var;
```

```
    var = *lock;  
    *lock = 1;  
    return var;  
}
```

(Здесь и далее мы, следуя правилам языка C, в котором параметры передаются по значению, вынуждены передавать в функции указатели, чтобы функции могли изменять значения параметров).

Команда проверяет (возвращает) значение некоторой переменной, а затем устанавливает ее значение в 1. Введенный нами описатель функции `atomic` показывает, что функция непрерываемая, во время ее выполнения никакой другой процесс не имеет доступа к той памяти, с которой работает функция (к переменной `lock`). Функция, как мы видим, выполняет два обращения к переменной `lock`, но оба они выполняются как одна транзакция.

Возможно, первые включения команд типа `testAndSet` в системы команд диктовались иными соображениями, но сейчас возможность выполнения подобных команд является обязательной для процессоров, претендующих на возможность использования в многопроцессорных комплексах. В микропроцессорах Intel-Pentium, например, имеются следующие команды, которые представляют собой "вариации на тему" `testAndSet`:

- XCHG – перестановка;
- BTS – проверка и установка бита;
- BTR – проверка и сброс бита;
- BTC – проверка бита и установка противоположного значения;
- CMPXCHG – сравнить и заменить;
- XADD – заменить и сложить.



Так, для реализации "канонической" функции `testAndSet` при помощи команды `XCHG` нужны две команды:

```
MOV  al,1
XCHG AL,LOCK
```

Переменная `lock` устанавливается в 1, а ее прежнее значение сохраняется в регистре `AL`.

Непрерываемость операций с памятью при использовании этих команд в многопроцессорных системах обеспечивается сигналом `LOCK#`, появляющимся на выходе микропроцессора. Причем команда `XCHG`, когда она использует операнд, расположенный в памяти, активизирует сигнал `LOCK#` автоматически. Для других названных команд активизация этого сигнала может явным образом задаваться программистом – при помощи префикса `LOCK`.

Наличие в арсенале программиста команды `testAndSet` позволяет ему организовать простую и надежную защиту критической секции при помощи переменной-замка:

```
1  void csBegin ( char *lock ) {
2      while ( testAndSet( lock ) );
3  }
4  void csEnd ( char *lock ) {
5      *lock = 0;
6  }
```

Команда `testAndSet` (строка 2) будет возвращать 1 до тех пор, пока другой процесс находится в критической секции, защищенной замком. Как только этот другой процесс выйдет из критической секции и установит замок в 0 (строка 5), наш процесс тут же вновь установит его в 1

и выйдет из цикла. Вследствие атомарности `testAndSet` никакой другой процесс не сможет изменить состояние замка между теми моментами, когда наш процесс считает его нулевое значение и установит его значение в 1.

Каждый ресурс, к которому происходит совместный доступ (например, каждая разделяемая переменная), может быть защищен своим замком. Это обеспечит запрет только конфликтующих критических секций, позволяя разным процессам одновременно использовать разные разделяемые ресурсы.

При наличии двух и более групп ресурсов, со своим замком каждая, возможны тупики. Как мы помним из главы 5, наиболее либеральной политикой их предотвращения, не требующей априорной информации, является иерархическая – ее рекомендуется применять и в данном случае.

Замки, обеспечиваемые командой `testAndSet`, обладают, следовательно, такими свойствами:

- они применимы для любого числа процессов и любого числа процессоров;
- они просты для понимания и для верификации;
- они либеральны, так как не запрещают другим процессам выполняться и даже одновременно входить в неконфликтующие критические секции;
- они допускают бесконечное откладывание, так как выбор процесса, допускаемого к замку, определяется оборудованием, а мы ничего не знаем о его критериях арбитража;
- они используют занятое ожидание.

## 8.5. Семафоры

Все рассмотренные выше методы используют занятое ожидание: если процесс не может войти в критическую секцию, он закидывается на опросе переменных состояния. Следующие методы используют блокировку ожидающего процесса – перевод его из списка процессов, планируемых на выполнение (готовых) в список ожидающих (заблокированных). Этим экономится процессорное время, в противном случае попусту растрачиваемое в занятом ожидании, а затраты сводятся к переключению процессов.

Такая возможность обеспечивается:

- введением специальных целочисленных общих переменных, которые называются семафорами;
- добавлением к набору элементарных действий, из которых строятся процессы, операций над семафорами: V-операции и P-операции.

V-операция есть операция с одним операндом, который должен быть семафором. Выполнение операции состоит в увеличении значения аргумента на 1, это действие должно быть атомарным.

P-операция есть операция с одним операндом, который должен быть семафором. Выполнение операции состоит в уменьшении значения аргумента на 1, если только это действие не приведет к отрицательному значению операнда. Выполнение P-операции, т.е. принятие решения о том, что момент является подходящим для уменьшения аргумента, и последующее его уменьшение должно быть атомарным.

Атомарность P-операции и является потенциальной задержкой: если процесс пытается выполнить P-операцию над семафором, значение которого в данный момент нулевое, данная P-операция не может завершиться пока другой процесс не выполнит V-операцию над этим семафором. Несколько процессов могут начать одновременно P-операцию

над одним и тем же семафором. Тогда при установке семафора в 1 только одна из Р-операций завершится, какая именно – мы обсудим позже.

Защита разделяемых ресурсов теперь выглядит следующим образом. Каждый ресурс защищается своим семафором, значение которого может быть 1 – свободен или 0 – занят. Процесс, выполняющий доступ к ресурсу, инициирует Р-операцию (эквивалент `csBegin`). Если ресурс занят – процесс задерживается в своей Р-операции до освобождения ресурса. Когда ресурс освобождается, Р-операция процесса завершается и процесс занимает ресурс. При освобождении ресурса процесс выполняет V-операцию (эквивалент `csEnd`).

Э.Дейкстра [11], вводя семафорные примитивы для синхронизации и взаимного исключения, исходил из гипотезы о том, что Р- и V-операции реализованы в нашей вычислительной системе аппаратно. На самом же деле, в составе любого набора команд таких операций нет – и это оправданно. Программная реализация семафоров позволяет нам включить в них блокировку и диспетчеризацию процессов, чего нельзя было бы делать на аппаратном уровне.

В соответствии с общей методикой нашего подхода, сосредотачивающей внимание на механизмах реализации взаимного исключения, рассмотрим реализацию семафоров и операций над ними. Сам семафор может быть представлен следующим образом:

```
typedef struct {  
    int value;  
    char mutex;  
    process *waitList;  
} semaphore;
```

Здесь `value` – та самая целочисленная переменная, которая представляет значение семафора в приведенном выше определении. `mutex` – переменная взаимного исключения, которая обеспечивает, как мы увидим ниже, атомарность операций над семафорами. `waitList` – указатель на список процессов, ожидающих установления этого семафора в 1. (Здесь мы предполагаем линейный однонаправленный список, но очередь ожидающих процессов может быть представлена и любым другим образом.)

Мы начинаем рассмотрение с так называемых двоичных семафоров, для которых допустимые значения – 0 и 1. Если семафор защищает критическую секцию, то начальное значение поля `value` – 1. Начальные значения других полей: `mutex = 0; waitList = NULL`.

Операции над семафорами можно представить в виде следующих функций:

```
void P ( semaphore *s ) {
    csBegin (&s->mutex);
    if (!s->value) block(s);
    else {
        s->value--;
        csEnd (&s->mutex);
    }
}

void V ( semaphore *s ) {
    csBegin (&s->mutex);
    if(s->waitList!= NULL) unBlock(s);
    else s->value++;
    csEnd (&s->mutex);
}
```

В нашей реализации вы видите "скобки критической секции" как элементарные операции. Они обеспечивают атомарность выполнения семафоров и могут быть реализованы любым из описанных выше корректных способов. Здесь мы ориентируемся на команду `testAndSet` с использованием поля семафора `mutex` в качестве замка, но это может быть и любая другая корректная реализация (в многопроцессорных версиях Unix, например, используется алгоритм Деккера). Вопрос: в чем же мы выигрываем, если в `csBegin` все равно используется занятое ожидание? Дело в том, что это занятое ожидание не может быть долгим. Этими "скобками критической секции" защищается не сам ресурс, а только связанный с ним семафор. Выполнение же семафорных операций происходит быстро, следовательно, и потери на занятое ожидание будут минимальными.

Если при выполнении Р-операции оказывается, что значение семафора нулевое, выдается системный вызов `block`, который блокирует активный процесс – переводит его в список ожидающих, в тот самый список, который связан с данным семафором. Важно, что процесс прервется именно в контексте строки 3 своей Р-операции и впоследствии он возобновится в том же контексте. Поскольку заблокированный таким образом процесс не успеет закончить критическую секцию, это должен сделать за него системный вызов `block`, чтобы другие процессы получили доступ к семафору.

Когда процесс выполняет V-операцию (освобождает ресурс), проверяется очередь ожидающих процессов и разблокируется один из них. В системном вызове `unBlock` можно реализовать любую дисциплину обслуживания очереди, в том числе и такую, которая предупреждает возможность бесконечного откладывания процессов в очереди. Если

разблокируется какой-либо процесс, то значение семафора так и остается нулевым, если же очередь ожидающих пуста, то значение семафора увеличивается на 1.

Укажем еще вот на какую особенность. После того, как процесс разблокирован, он включается в список готовых. Может случиться так, что этот процесс будет выбран планировщиком и активизирован даже раньше, чем процесс, освободивший ресурс, закончит свою V-операцию. Поскольку разблокированный процесс восстанавливается в контексте своей Р-операции, то получится, что два процесса одновременно выполняют семафорные операции. В данном случае ничего страшного в этом нет, потому что для разблокированного процесса уже снято взаимное исключение (это было сделано при его блокировании), и этот процесс после разблокирования уже не изменяет значения семафора. Запомним, однако, эту особенность, которая в других примитивах взаимного исключения может приобретать более серьезный характер.

Итак, общие свойства решения задачи взаимного исключения с помощью семафоров таковы:

- дисциплина либеральна по тем же соображениям, что и предыдущая;
- метод справедлив для любого числа процессов и процессоров;
- когда процесс блокируется, он не расходует процессорное время на занятое ожидание;
- возможность бесконечного откладывания зависит от принятой дисциплины обслуживания очереди ожидающих процессов, при дисциплине FCFS бесконечное откладывание исключается;
- сами семафоры (но не защищаемые ими ресурсы) представляют собой монопольно используемые ресурсы, следовательно, могут порождать тупики; для борьбы с тупиками возможно применение

любого из известных нам методов, предпочтителен – иерархический;

- как и все методы, рассмотренные выше, семафоры требуют от программиста корректного применения "скобок", в роли которых выступают P- и V-операции.

Для решения задачи взаимного исключения достаточно двоичных семафоров. Мы, однако, описали тип поля `value` как целое число. В приведенном нами выше определении Дейкстры речь тоже идет о целочисленном, а не о двоичном значении. Семафор, который может принимать неотрицательные значения, большие, чем 1, называется общим семафором. Такой семафор может быть очень удобен, например, при управлении не единичным ресурсом, а классом ресурсов. Начальное значение поля `value` для такого семафора устанавливается равным числу единиц ресурса в классе. Каждое выделение единицы ресурса процессу сопровождается P-операцией, уменьшающей значение семафора. Семафор, таким образом, играет роль счетчика свободных единиц ресурса. Когда этот счетчик достигнет нулевого значения, процесс, выдавший следующий запрос на ресурс, будет заблокирован в своей P-операции. Освобождение ресурса сопровождается V-операцией, которая разблокирует процесс, ожидающий ресурс или наращивает счетчик ресурсов.

Общие семафоры могут быть использованы и для простого решения задачи синхронизации. В этом случае семафор связывается с каким-либо событием и имеет начальное значение 0. (Событие может рассматриваться как ресурс, и до наступления события этот ресурс недоступен). Процесс, ожидающий события, выполняет P-операцию и блокируется до установки семафора в 1. Процесс, сигнализирующий о событии, выполняет над семафором V-операцию. Для графа синхронизации, например, показанного на рисунке 8.1, мы свяжем с каждым действием графа одноименный



семафор. Тогда каждое действие (например, E) должно быть оформлено следующим образом:

```
procE () {  
    /*ожидание событий В и D*/  
    P(B); P(D);  
    . . .  
    /* сигнализация о событии E для двух  
    ожидающих его действий (F и H) */  
    V(E); V(E);  
}
```

Ниже мы рассмотрим применение семафоров для более сложного варианта задачи синхронизации.

## 8.6. "Производители–потребители"

Пусть мы имеем два циклических процесса, которые мы назовем "производитель" и "потребитель". Производитель в каждой итерации своего цикла вырабатывает (производит) порцию информации, которую он помещает в общий для обоих процессов буфер. Предположим для начала, что емкость буфера неограничена. Потребитель в каждой итерации своего цикла выбирает из буфера порцию информации, выработанную производителем, и обрабатывает (потребляет) ее. Потребитель не должен начинать обработку порции, пока ее производство не будет закончено. Задача состоит в синхронизации действий производителя и потребителя таким образом, чтобы не допустить потерь и искажений информации, во-первых, и голодания процессов, во-вторых.

Решение достигается при помощи единственного общего семафора, играющего роль счетчика числа порций в буфере (здесь и далее мы предполагаем структуру семафора, определенную в предыдущем разделе):

```
1  static semaphore *portCnt =
2      { 0, 0, NULL };
3  static ... buffer ...;
4  /* процесс-производитель */
5  void producer ( void ) {
6      while (1) {
7          < производство порции >
8          < добавление порции в буфер >
9          V(portCnt);
10     }
11 }
12 /* процесс-потребитель */
12 void consumer ( void ) {
14     while (1) {
15         P(portCnt)
16         < выборка порции из буфера >
17         < обработка порции >
18     }
19 }
```

Исходное значение семафора portCnt – 0. Производитель каждую итерацию своего цикла заканчивает V-операцией, увеличивающей значение счетчика. Потребитель каждую свою итерацию начинает P-операцией. Если буфер пуст, то потребитель задержится в своей P-операции до появления в буфере очередной порции. Таким образом, если

потребитель работает быстрее производителя, он будет время от времени простаивать, если производитель работает быстрее – в буфере будут накапливаться порции.

В реальных задачах такого рода (например, кольцевой буферизации, рассмотренной нами в главе 6) буфер всегда имеет некоторую конечную емкость. Если производитель работает быстрее, то при заполнении буфера он должен приостанавливаться, ожидая освобождения места в буфере. Это легко обеспечить, введя новый семафор `freeCnt`, выполняющий роль счетчика свободных мест в буфере:

```
1  static semaphore *portCnt =
2      { 0, 0, NULL },
3  *freeCnt = { BSIZE, 0, NULL },
4  static ... buffer [BSIZE];
5  /* процесс-производитель */
6  void producer ( void ) {
7      while (1) {
8          < производство порции >
9          P(freeCnt);
10         < добавление порции в буфер >
11         V(portCnt);
12     }
13 }
14 /* процесс-потребитель */
15 void consumer ( void ) {
16     while (1) {
17         P(portCnt)
18         < выборка порции из буфера >
19         V(freeCnt);
```

```

20      < обработка порции >
21      }
22      }

```

Попытаемся теперь обобщить решение для произвольного числа производителей и потребителей. Но в таком обобщении мы сталкиваемся с еще одной существенной особенностью. В приведенных выше решениях мы допускали одновременное выполнение операций <добавление порции в буфер> и <выборка порции из буфера>. Очевидно, что при любой организации буфера производитель и потребитель могут одновременно работать только с разными порциями информации. Иначе обстоит дело с многочисленными производителями и потребителями. Два производителя могут попытаться одновременно добавить порцию в буфер и выбрать для этого одно и то же место в буфере. Аналогично два потребителя могут попытаться выбрать из буфера одну и ту же порцию. В следующем примере мы для наглядности представляем буфер в виде кольцевой очереди с элементами (порциями) одинакового размера:

```

1      typedef ... portion; /* порция информации */
2      static portion buffer [BSIZE];
3      static int wIndex = 0, rIndex = 0;
4      static semaphore *portCnt = { 0, 0, NULL },
5      *freeCnt = { BSIZE, 0, NULL },
6      *rAccess = { 1, 0, NULL },
7      *wAccess = { 1, 0, NULL };
8      /* имеется NP
9      аналогичных процессов-производителей */
10     void producer ( void ) {
11         portion work;

```

```

12     while (1) {
13         < производство порции в work >
14         P(wAccess);
15         P(freeCnt);
16         /* добавление порции в буфер */
17         memcpy(buffer+wIndex,&work,
18                 sizeof(portion) );
19         if ( ++wIndex == BSIZE ) w_index = 0;
20         V(portCnt);
21         V(wAccess);
22     }
23 }
24 /* имеется NC
25     аналогичных процессов-потребителей */
26 void consumer ( void ) {
27     portion work;
28     while (1) {
29         P(rAccess);
30         P(portCnt)
31         /* выборка порции из буфера */
32         memcpy(&work, buffer+rIndex,
33                 sizeof(portion) );
34         if ( ++rIndex == BSIZE ) rIndex = 0;
35         V(freeCnt);
36         V(rAaccess);
37         < обработка порции в work>
38     }
39 }

```

Мы оформляем обращения к буферу как критические секции, защищая их семафорами `rAccess` и `wAccess`. Поскольку конфликтовать (пытаться работать с одной и той же порцией в буфере) могут только однотипные процессы, мы рассматриваем буфер как два ресурса: ресурс для чтения и ресурс для записи, и каждый такой ресурс защищается своим семафором. Таким образом, запрещается одновременный доступ к буферу двух производителей или двух потребителей, но разрешается одновременный доступ одного производителя и одного потребителя.

## **8.7. Конструкции критических секций в языках программирования**

Семафоры являются удобными и эффективными примитивами, при помощи которых решаются задачи взаимного исключения и синхронизации, поэтому они широко используются во многих ОС. API ОС, обеспечивающий для пользователя работу с семафорами, мы рассмотрим в следующей главе, а здесь остановимся на некоторых возможных расширенных механизмах в системном программном обеспечении (прежде всего – в системах программирования), которые будут использовать семафоры в скрытой от пользователя форме.

Некоторые неудобства использования критических секций могут быть преодолены введением специальных конструкций в язык программирования. Например, если у нас в программе есть разделяемая переменная `x`, то удобно защитить ее конструкцией типа:

```
shared int x;  
.  
.  
.  
section ( x ) {
```

```
< операторы, работающие с переменной x >  
}
```

Конструкция `section` определяет критическую секцию. Вместо специальных "скобок критической секции" используется заголовок `section` и обычные операторные скобки. Последнее дает возможность проверять правильность оформления критической секции на синтаксическом уровне – на этапе трансляции программы, а не ее выполнения, – и предупреждает возможность появления самой распространенной ошибки программистов – непарных скобок. Определение переменной `x` со специальным описателем `shared` позволяет выявить (опять на этапе компиляции) все попытки доступа к ней вне критической секции.

Реализация этой конструкции очевидна: переменная `x` защищается скрытым семафором, над которым производится P-операция при входе в секцию и V-операция – при выходе из нее.

Отчасти такая конструкция позволяет решить и проблему тупиков. Если описать в программе иерархию разделяемых переменных, то можно спокойно разрешить программисту делать вложенные критические секции и проверять правильность вложения на этапе компиляции. Этот метод, однако, не универсален: если в критической секции есть обращение к процедуре, а в последней – другая критическая секция, то правильность такого вложения компилятор проверить не сможет. Другой путь – запретить вложенные секции, но разрешить в заголовке секции указывать не одну разделяемую переменную, а целый их список. Этот вариант более прост и надежен, но он использует дисциплину залпового выделения ресурсов и, следовательно, более консервативен.

Критические секции как языковые конструкции обеспечивают взаимное исключение, но не синхронизацию процессов. Инструментом для

синхронизации может быть оператор типа `await`, операндом которого является логическое выражение. Этим оператором процесс блокируется до тех пор, пока операнд не примет значение "истина". Поскольку ожидание связано с внешним событием, в логическом выражении должна участвовать хотя бы одна разделяемая переменная, следовательно, применение `await` может быть разрешено только в критической секции.

В качестве примера приведем решение задачи "производители–потребители" для процессов, использующих буфер, организованный в виде стека. Такая организация буфера диктует нам необходимость запретить одновременный доступ к нему любых двух процессов.

```
1  shared struct {
2    portion buffer [BSIZE];
3    int stPtr;
4  } stack = { {...}, 0 };
5  void producer ( void ) {
6    portion work;
7    while (1) {
8      < производство порции в work >
9      section ( stack ) {
10         await ( stack.stRtr < BSIZE );
11         memcpy ( stack.buffer +
12                 stack.stPtr++,
13                 &work, sizeof(portion) );
14     }
15 }
16 }
17 void consumer ( void ) {
18     portion work;
```



```

19     while (1) {
20         section ( stack ) {
21             await( stack.stPtr > 0 );
22             memcpy ( &work, stack.buffer +
23                     --stack.stPtr,
24                     sizeof(portion) );
25         }
26         < обработка порции в work>
27     }
28 }

```

При реализации возможности `await` мы должны решить проблему исключения. В приведенном выше примере процесс-производитель, заблокированный в строке 10, ждет уменьшения значения указателя стека (если стек полон). Это значение может быть уменьшено процессом-потребителем в строке 23, но эта строка находится в критической секции потребителя, а последний не может войти в свою критическую секцию, так как производитель уже вошел в свою, – в строке 9. Одним из способов разрешения этого противоречия является запрещение употребления `await` где-либо, кроме самого начала критической секции, возможно, даже включение `await`-условия в заголовок секции. Исключение в этом случае начинает работать только после выхода из `await`. Естественно, что сама проверка условия должна выполняться как атомарная операция (защищаться скрытым семафором).

Можно разрешить `await` где угодно в критической секции, но на время блокировки процесса в `await` снимать взаимное исключение. Конечно, такой вариант дает программисту более гибкий инструмент, но перекладывает на него ответственность за целостность, так как за время, на

которое исключение снимается, разделяемые данные могут быть изменены.

Еще один вопрос, который мы должны решить при реализации `await`: если процесс заблокировался, то в какие моменты следует проверять условие разблокирования? Вопрос неспроста, так как для вычисления условия, являющегося операндом `await`, необходимо активизировать контекст заблокированного процесса. Если проделывать это слишком часто, то накладные расходы на переключение неоправданно возрастают. Для варианта, в котором мы жестко привязываем `await` к началу критической секции, общая переменная, входящая в условие, может быть изменена только другим процессом и ее новое значение станет доступным при выходе этого другого процесса из его критической секции. Выход другого процесса из критической секции – единственное событие, по которому имеет смысл переключаться в контекст заблокированного процесса и проверять условие в этом варианте. Если же мы разрешаем употребление `await` где угодно в критической секции, то добавляется еще один тип события – блокировка другого процесса в операторе `await` его критической секции.

## 8.8. Мониторы

Монитор представляет собой набор информационных структур и процедур, которые используются в режиме разделения. Некоторые из этих процедур являются внешними и доступны процессам пользователей, их имена представляют входные точки монитора. Пользователь не имеет доступа к информационным структурам монитора и может воздействовать на них, только обращаясь к входным точкам. Монитор, таким образом, воплощает принцип инкапсуляции данных. В терминах объектно-

ориентированного программирования ресурс, обслуживаемый монитором, представляет собой объект, а входные точки – методы работы с этим объектом. Особенностью монитора, однако, является то, что в его состав входят так называемые "процедуры с охраной", которые не могут выполняться двумя процессами одновременно.

Не являясь средством более мощным или гибким, чем рассмотренные выше примитивы, мониторы, однако представляют значительно более удобный инструмент для программиста, избавляя его от необходимости формировать критические секции, обеспечивая более высокий уровень интеграции данных и предупреждая возможные ошибки во взаимном исключении.

Если в задаче "производители–потребители" процессы программируются пользователем, то вид этих процессов может быть таким:

```
1  #include <monitor.h>
2  /* процесс-производитель
3   (может быть отдельным модулем) */
4  void producer ( void ) {
5      portion work;
6      while (1) {
7          < производство порции в work >
8          putPortion ( &work );
9      }
10 }
11 /* процесс-потребитель
12 (может быть отдельным модулем) */
13 void consumer ( void ) {
14     portion work;
```

```

15     while (1) {
16         getPortion ( &work );
17         < обработка порции в work>
18     }
19 }

```

Обратим внимание на то, что процессы, во-первых, никоим образом не заботятся о разделении данных, во-вторых, не используют никакие общие данные. Такая "беззаботная" работа процессов, однако, должна быть поддержана монитором, входные точки которого описаны в файле `monitor.h`, а определение его имеет такой вид:

```

1     /* монитор производителей-потребителей
2     (отдельный модуль) */
3     #define BSIZE ...
4     /* буфер */
5     static portion buffer [BSIZE];
6     /* индексы буфера для чтения и записи*/
7     static int rIndex = 0, wIndex = 0;
8     /* счетчик заполнения */
9     static int cnt = 0;
10    /* события НЕ_ПУСТ, НЕ_ПОЛОН */
11    static event nonEmpty, nonFull;
12    /* процедура занесения порции в буфер*/
13    void guard putPortion ( portion *x ) {
14        /* если буфер полон -
15        ожидать события НЕ_ПОЛОН */
16        if ( cnt == BSIZE ) wait (nonFull);
17        /* запись порции в буфер */

```

```

18     memcpy ( buffer + wIndex,
19             x, sizeof(portion) ) ;
20     /* модификация индекса записи */
21     if ( ++wIndex == BSIZE ) wIndex = 0;
22     cnt++; /* подсчет порций в буфере */
23     /* сигнализация о том,
24        что буфер НЕ_ПУСТ */
25     signal (nonEmpty);
26 }
27 void guard getPortion ( portion *x ) {
28     if ( cnt == 0 ) wait (nonEmpty);
29     memcpy ( x, buffer + rIndex,
30             sizeof(portion) ) ;
31     if ( ++rIndex == BSIZE ) rIndex = 0;
32     cnt++;
33     signal (nonFull);
34 }

```

В реализации монитора нам пришлось прибегнуть к некоторым новым обозначениям. Во-первых, функции монитора даны с описателем `guard` (охрана). Это означает, что они должны выполняться в режиме взаимного исключения. В литературе часто употребляется образное сравнение мониторов с комнатой, в которой может находиться только один человек. Такая комната показана на рисунке 8.2. Если человек (процесс) желает войти в комнату (охраняемую процедуру монитора), то он становится во входную очередь к двери 1, в которой он ожидает (блокируется) до тех пор, пока комната (монитор) не освободится. Дверь 1 (вход) отпирается только в том случае, если комната пуста, пропускает

только одного человека и запирается за ним. Дверь 2 (выход) не заперта, когда она открывается, отпирается и дверь 1.

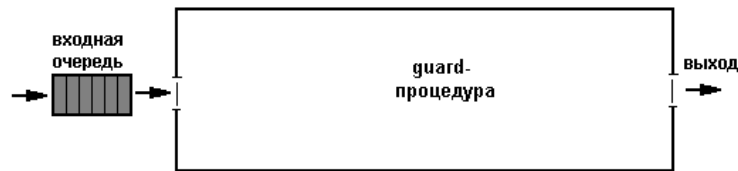


Рисунок 8.2 Простая модель монитора

Обратите внимание на то, что взаимное исключение обеспечивается для всех охраняемых процедур, а не только для одноименных. В сущности, такая процедура представляет собой ту же критическую секцию, и ее охрана реализуется любым из методов защиты критической секции, скорее всего, в роли "охранника" будет выступать скрытый семафор.

Другие наши нововведения связаны с блокировками внутри охраняемой процедуры. Мы ввели тип данных, названный нами `event`. Этот тип представляет некоторое событие. Примитив `wait` проверяет наступление этого события и переводит процесс в ожидание, если событие еще не произошло. Примитив `signal` сигнализирует о наступлении события. Событие является потребляемым ресурсом: если два процесса ожидают одного и того же события, то при наступлении события разблокирован будет только один из процессов, другой будет вынужден ждать повторного наступления такого же события.

Примитивы ожидания-сигнализации требуют принятия решений по ряду проблем их реализации. Если один процесс выдает сигнал о наступлении некоторого события, то в какой момент должен быть разблокирован процесс, ожидающий этого события? Если немедленно, то тогда в нашей "комнате" окажется два процесса одновременно: разблокированный процесс и процесс, выдавший сигнал, но еще не покинувший монитор. Если позже, то за это время в монитор может войти

какой-то третий процесс. Если несколько процессов ожидают одного и того же события, то какой (или какие) из них должен быть разблокирован? Если в момент выдачи сигнала нет процессов, ожидающих этого события, то должен ли сигнал сохраняться или его можно "потерять"?

Общий подход к решению этой проблемы иллюстрируется расширением модели "одноместной комнаты", показанным на рисунке 8.3. Для каждого события, которое может ожидаться в мониторе, мы вводим свою очередь с соответствующими входными и выходными дверями для нее. На рисунке эти очереди показаны внизу монитора. Мы вводим также очередь, которую мы называем приоритетной. Процессы, находящиеся в очередях, не считаются находящимися в мониторе. "Правила для посетителей" комнаты-монитора следующие.

1. Новый процесс поступает во входную очередь. Новый процесс может войти в монитор через дверь 1 только, если в мониторе нет других процессов.
2. Если процесс выходит из монитора через дверь 2 (выход), то в монитор входит процесс из двери 4 – из приоритетной очереди. Если в приоритетной очереди нет ожидающих, входит процесс из двери 1 (если есть ожидающие за ней).
3. Процесс, выполняющий операцию `wait`, выходит из монитора в дверь, ведущую в соответствующую очередь (5 или 7).
4. Если процесс выполняет операцию `signal`, то проверяется очередь, связанная с событием. Если эта очередь не пуста, то сигнализирующий процесс уходит в приоритетную очередь (дверь 3), а в монитор входит один процесс из очереди к событию (дверь 6 или 8). Если очередь пуста, сигнализирующий процесс остается в мониторе.
5. Все очереди обслуживаются по дисциплине FCFS.

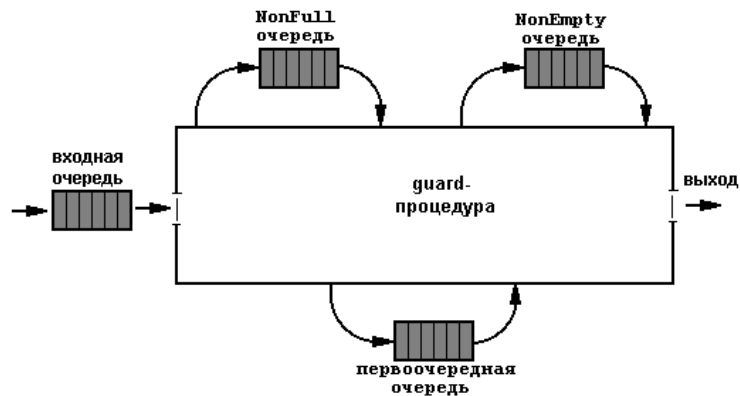


Рисунок 8.3 Расширенная модель монитора

Эти правила предполагают, что процесс будет разблокирован немедленно (речь идет не о немедленной активизации процесса, а о его разблокировании – перемещении в очередь готовых к выполнению). Разблокированный процесс имеет преимущество перед процессом, ожидающим во входной очереди.

В нашем примере производителей–потребителей мы употребляли операцию `signal` в конце процедуры. Такое употребление характерно для очень большого числа задач. Наши правила требуют перемещения сигнализирующего процесса в приоритетную очередь. Однако, если сигнализирующий процесс после выдачи сигнала больше не выполняет никаких действий с разделяемыми данными, необходимости в таком перемещении (и вообще в приоритетной очереди) нет. Жесткая привязка сигнала к окончанию охраняемой процедуры снижает гибкость монитора, но значительно упрощает диспетчеризацию процессов в мониторе.

Возможно решение, в котором операция `signal` разблокирует все процессы, находящиеся в соответствующей очереди. Поскольку все ожидавшие процессы не могут вместе войти в монитор, в нем остается только один из них, успевший "подхватить" событие, а остальные направляются в приоритетную очередь. Процесс, который разблокировался таким образом, уже не может, однако, быть уверенным в



том, что его разблокирование гарантирует наступление события (событие могло быть перехвачено другим процессом). Поэтому в проверке условия ожидания оператор `if` для такой реализации должен быть заменен оператором `while`, например, строки 14 - 16 последнего примера должны выглядеть так:

```
14      /* если буфер полон -  
15      ожидать события НЕ_ПОЛОН */  
16      while ( cnt == BSIZE ) wait (nonFull);
```

Поскольку охраняемые процедуры есть критические секции, для поддержания высокого уровня мультипрограммирования нахождение в них процессов должно быть кратковременным. Проблема времени может возникнуть в том случае, когда в процедуре монитора имеется обращение к другому монитору. По нашим правилам такое обращение не выводит процесс из первого монитора. Однако во втором (вложенном) мониторе процесс может быть заблокирован, тогда его пребывание в первом мониторе недопустимо затянется. Возможно несколько вариантов решения этой проблемы:

- переложить ответственность за возникновение такой ситуации на программиста;
- запретить вложенные вызовы вообще;
- при вхождении во вложенный монитор автоматически снимать исключение с внешнего монитора, когда процесс возвращается из вложенного монитора, он попадает в приоритетную очередь внешнего монитора;
- предоставить программисту выбор из перечисленных выше возможностей.

## 8.9. "Читатели–писатели" и групповые мониторы

Еще одна классическая задача синхронизации называется задачей "читателей–писателей" и формулируется следующим образом. Имеется произвольное число процессов-писателей и процессов-читателей, которые совместно используют какие-то данные (обычно имеется в виду файл). В любой момент процесс-читатель может потребовать прочитать данные. В любой момент процесс-писатель может потребовать прочитать или записать данные. Чтение и запись данных – операции длительные, но конечные. В то время, когда процесс записывает данные, никакие другие читатели или писатели не должны иметь доступа к данным. Любое число процессов может читать данные одновременно. Решение должно обеспечивать целостность данных и отсутствие бесконечного откладывания процессов.

Существенные отличия этой задачи от задачи производителей–потребителей состоят в следующем:

- процессы чтения и записи длительные;
- данные представляют собой повторно используемые ресурсы, а не потребляемые, как в предыдущей задаче;
- требуются различные режимы доступа для чтения и записи.

Нам не удастся решить эту задачу при помощи мониторов, описанных в предыдущем разделе, так как, если мы сделаем процедуры `read` и `write` охраняемыми, то, во-первых, у нас получатся слишком большие (длительные) критические секции, а во-вторых, мы исключим параллельное выполнение читателей.

Решение может быть получено при помощи так называемого группового монитора. В такой монитор входят как охраняемые, так и

неохраняемые процедуры. Для того, чтобы процесс получил возможность доступа к неохраняемой процедуре, он должен быть членом группы, за которой право такого доступа закреплено. Группы формируются динамически. Для прикрепления к группе процесс должен обратиться к охраняемой процедуре. Если в данный момент прикрепление к группе возможно, эта процедура прикрепит процесс к группе, если нет – заблокирует процесс до появления такой возможности. После окончания доступа процесс должен вызвать также охраняемую процедуру открепления от группы. Для задачи "читатели–писатели" предполагается такой порядок доступа процессов к данным:

- для читателей:  

```
startRead ( proc );  
read( proc, ... );  
endRead ( proc );
```
- для писателей:  

```
startWrite ( proc );  
write ( proc, ... );  
endWrite ( proc );
```

где `proc` – идентификатор процесса.

Структура самого монитора в общих чертах следующая:

```
1  /* счетчик читателей */  
2  int rdCnt = 0;  
3  /* признак активности записи */  
4  char wrFlag = 0;  
5  /* списки - писателей и читателей */  
6  process *wrCrowd=NULL, *rdrowd=NULL;  
7  /* события:
```

```

8      МОЖНО_ЧИТАТЬ, МОЖНО_ПИСАТЬ */
9      event mayRead, mayWrite;
10     /* процедура регистрации читателя */
11     void guard startRead ( process *p ) {
12         rdCnt++; /* подсчет читателей */
13         /* если идет запись -
14            ожидать МОЖНО_ЧИТАТЬ */
15         if ( wrFlag ) wait (mayRead);
16         /* дублирование сигнала
17            для другого читателя */
18         signal (mayRead);
19         /* включение в список читателей */
20         inCrowd ( rdCrowd, p );
21     }
22     /* процедура открепления читателя */
23     void guard endRead ( process *p ) {
24         /* исключение из списка читателей */
25         fromCrowd ( rdCrowd, p );
26         /* уменьшение числа читателей,
27            если читателей больше нет -
28            сигнализация МОЖНО_ПИСАТЬ */
29         if ( --rdCnt==0 ) signal(mayWrite);
30     }
31     /* процедура регистрации писателя */
32     void guard startWrite ( process *p ) {
33         /* если есть другие читатели
34            или писатели - ждать */
35         if ( wrFlag || rdCnt ) wait(mayWrite);
36         /* установка признака записи */

```

```

37     wrFlag = 1;
38     /* писатель включается в оба списка */
39     inCrowd ( rdCrowd, p );
40     inCrowd ( wrCrowd, p );
41 }
42 /* процедура открепления писателя */
43 void guard endWrite ( process *p ) {
44     wrFlag=0; /* сброс признака записи */
45     /* исключение из списков */
46     fromCrowd ( rdCrowd, p );
47     fromCrowd ( wdCrowd, p );
48     /* если есть претенденты-читатели -
49        разрешение им */
50     if ( rdCnt ) signal (mayRead);
51     /* иначе - разрешение на запись */
52     else signal (mayWrite);
53 }
54 /* процедура чтения */
55 void read ( process *p,
56            < другие параметры > ) {
57     /* если процесс не зарегистрирован
58        читателем - отказ */
59     if (!checkCrowd(rdCrowd, p)) <отказ>;
60     else < чтение данных >;
61 }
62 /* процедура записи */
63 void write ( process *p,
64            < другие параметры > ) {
65     /* если процесс не зарегистрирован

```

```

66      писателем - отказ */
67      if (!checkCrowd(wrCrowd,p)) <отказ>;
68      else < запись данных >;
69  }

```

Прежде чем процесс получит доступ к данным, он должен зарегистрироваться как читатель или как писатель. В нашем примере переменные `rdCrowd` и `wrCrowd` (строка 6) являются указателями на списки читателей и писателей соответственно, хотя можно интегрировать процессы в группы и любым другим способом. Используемые (но не определенные) нами функции `inCrowd` и `fromCrowd` обеспечивают включение процесса в группу и исключение из группы, а функция `checkCrowd` возвращает 1, если указанный процесс входит в группу (иначе – 0). Процедура `read` выполняется только для процессов, включенных в группу читателей (строки 59, 60), а `write` – только для включенных в группу писателей (строки 67, 68). Переменная `rdCnt` (строка 2) – счетчик текущего числа читателей, переменная `wrFlag` (строка 4) – счетчик писателей (или признак наличия писателя, так как писателей не может быть более одного). События `mayRead` и `mayWrite` (строка 9) являются разрешениями читать и писать соответственно.

Входная точка `startRead` (строка 11) выполняет регистрацию читателя. Это охраняемая процедура. Она наращивает счетчик читателей, но если в данный момент работает писатель, то потенциальный читатель блокируется до наступления события `mayRead` (строка 15). После того, как процесс будет разблокирован (или если он не блокировался вообще), он выдает сигнал `mayRead` (строка 18), который предназначается для другого читателя, возможно, также ждущего разрешения на чтение (можно сказать, что процесс этим

восстанавливает потребленный им ресурс), и включается в список читателей. После завершения доступа читатель обращается к входной точке `endRead` (строка 23). Эта процедура исключает процесс из списка читателей, уменьшает счетчик читателей и, если читателей больше не осталось, сигнализирует разрешение на запись.

Писатель регистрируется/разрегистрируется через входные точки `startWrite/endWrite` (строки 32/43). При регистрации потенциальный писатель может быть заблокирован, если в настоящий момент зарегистрирован другой писатель или хотя бы один читатель (строка 35). Сигнал `mayWrite` разблокирует писателя, и он включается в обе группы (строки 39, 40), так как имеет право и читать, и писать. При откреплении писатель исключается из групп. Если за время его работы попытался зарегистрироваться хотя бы один читатель, выдается разрешение на чтение (строка 50), в противном случае – разрешение на запись для другого писателя, возможно, ждущего своей очереди (строка 52).

## **8.10. Примитивы синхронизации в языках программирования**

До сих пор мы рассматривали методы, которые обеспечивают и взаимное исключение, и синхронизацию. Целый ряд прикладных задач, однако, требует только синхронизации без взаимного исключения. Отказ от последнего, если это не мешает решению задачи, всегда оправдан, так как взаимное исключение усложняет решение и может снижать уровень мультипрограммирования.

Один из возможных примитивов, обеспечивающих синхронизацию без взаимного исключения, называется счетчиком событий. Счетчик

событий – тип данных, представляемый неубывающим целым числом с начальным значением 0. Его значение в любой момент времени – число событий определенного типа, происшедших от некоторой точки начала отсчета. Над этим типом данных возможны следующие операции:

- `advance(E)` – увеличение значения счетчика событий `E` на 1, атомарная операция;
- `eread(E)` – возвращает текущее значение счетчика `E`, эта операция не взаимоисключающая с `advance`, так что к моменту, когда значение попадет в читающий его процесс, текущее значение счетчика может быть уже изменено;
- `await(E,value)` – ждать – ожидание (блокировка процесса), пока значение счетчика `E` не станет большим, чем `value`, или равным ему.

Существенно, что из перечисленных операций только `advance` является взаимоисключающей, остальные могут выполняться параллельно друг с другом и с `advance`.

Вот как решается с помощью счетчиков событий задача для одного производителя и одного потребителя:

```
1  /* тип данных - счетчик событий */
2  typedef unsigned int eventcounter
3  /* счетчики для чтения и записи */
4  static eventcounter inCnt = 0,
5      outCnt = 0;
6  /* буфер */
7  static portion buffer [BUFSIZE];
8  /* процесс-потребитель */
9  void consumer ( void ) {
```



```

10     int portNum; /* номер порции */
11     /* рабочая область порции */
12     portion work;
13     /* цикл потребления */
14     for ( portNum = 1; ; portNum++ ) {
15         /* ожидание доступности порции
16            по номеру */
17         await (inCnt, portNum);
18         /* выборка из буфера */
19         memcpy (&work,
20                buffer + portNum % BSIZE,
21                sizeof(portion) );
22         /* продвижение счетчика записи */
23         advance (outCnt);
24         < обработка порции в work>
25     }
26 }
27 /* процесс-производитель */
28 void producer ( void ) {
29     int portNum; /* номер порции */
30     /* рабочая область для порции */
31     portion work;
32     /* цикл производства */
33     for ( portNum = 1; ; portNum++ ) {
34         < производство порции в work >
35         /* ожидание доступности порции
36            по номеру */
37         await (outCnt, portNum - BSIZE);
38         /* запись в буфер */

```

```

39     memcpy (buffer + portNum % BSIZE,
40             &work, sizeof(portion) );
41     /* продвижение счетчика чтения */
42     advance (inCnt);
43 }
44 }

```

Как мы уже отмечали выше, производитель и потребитель работают с разными секциями буфера и взаимное исключение для них не требуется. Процессы – производитель и потребитель – могут перекрываться в любых своих фазах, кроме операций `advance` (строки 23 и 42). Переменные `inCnt` и `outCnt` являются счетчиками событий – производства порции и потребления порции соответственно. Кроме того, каждый процесс хранит в собственной локальной переменной `portNum` номер порции, с которой ему предстоит работать (счет начинается с 1). Потребитель ждет, пока счетчик производств не достигнет номера очередной его порции, затем выбирает порцию из буфера и увеличивает счетчик потреблений. Производитель работает симметрично. Обратите внимание на второй параметр операции `await` в производителе (строка 37). Он задается таким, чтобы обеспечить отсутствие ожидания при наличии хотя бы одной свободной секции в буфере.

Другой механизм синхронизации носит название секвенсоров (`sequencer`). Буквальный перевод этого слова – "упорядочиватель"; так называются средства, которые выстраивают неупорядоченные события в определенном порядке. Как и счетчик событий, секвенсор представляется целым числом, над которым выполняется единственная операция: `ticket`. Операция `ticket(S)` возвращает текущее значение секвенсора и увеличивает его на 1. Операция является атомарной. Начальное значение секвенсора – 0.

Имея в своем распоряжении секвенсоры, мы можем так записать решение задачи производителей–потребителей для произвольного числа процессов:

```
1    /* типы данных - счетчик событий
2      и секвенсор */
3    typedef unsigned int eventcounter;
4    typedef unsigned int sequencer;
5    /* счетчики для чтения и записи */
6    static eventcounter inCnt = 0,
7      outCnt = 0;
8    /* секвенсоры для чтения и записи */
9    static sequencer inSeq = 0, outSeq = 0;
10   /* буфер */
11   static portion buffer [BUFSIZE];
12   /* процесс-производитель */
13   void producer ( void ) {
14     int portNum; /* номер порции */
15     /* рабочая область для порции */
16     portion work;
17     /* цикл производства */
18     while (1) {
19       < производство порции в work >
20       /* получение "билета"
21         на запись порции */
22       portNum = ticket (inSeq);
23       /* ожидание номера порции */
24       await (inCnt, portNum);
25       /* ожидание свободного места
```

```

26     в буфере */
27     await (outCnt, portNum - BSIZE+1);
28     /* запись в буфер */
29     memcpy (buffer + portNum % BSIZE,
30             &work, sizeof(portion) );
31     /* продвижение счетчика чтения */
32     advance (inCnt);
33 }
34 }
35 /* процесс-потребитель */
36 void consumer ( void ) {
37     int portNum; /* номер порции */
38     /* рабочая область для порции */
39     portion work;
40     /* цикл потребления */
41     while (1) {
42         /* получение "билета"
43            на выборку порции */
44         portNum = ticket (outSeq);
45         /* ожидание номера порции */
46         await (outCnt, portNum);
47         /* ожидание появления в буфере */
48         await (inCnt, portNum+1);
49         /* выборка порции */
50         memcpy (&work,
51                buffer + portNum % BSIZE,
52                sizeof(portion) );
53         /* продвижение счетчика записи */
54         advance (outCnt);

```

```

55      < обработка порции в work>
56      }
57      }

```

Каждый производитель получает "билет" со своим номером в очереди на запись в буфер (строка 22). Затем он ожидает, когда до него дойдет очередь (строка 24), ожидает освобождения места в буфере (строка 27), записывает информацию (строки 29, 30) и наращивает счетчик производств (строка 32). Увеличение счетчика событий `inCnt` является сигналом к разблокированию как для потребителя, получившего "билет" на выборку этой порции и ожидающего в строке 46, так и для производителя, получившего "билет" на запись следующей порции и ожидающего в строке 27. Полученный процессом "билет" определяет и адрес в буфере той секции, с которой будет работать процесс. Хотя каждый процесс работает со своей секцией в буфере, одновременный доступ к буферу однотипных процессов исключается ожиданием в строке 24 или 46. Если разрешить одновременный доступ к буферу двух, например, производителей, то процесс, получивший "билет" на запись порции в  $n$ -ю секцию буфера может закончить запись раньше, чем процесс, пишущий порцию в  $n-1$ -ю секцию, даже если последний начал запись раньше. Процесс, закончивший запись, увеличит счетчик `inCnt` и выйдет из ожидания потребитель, имеющий билет на  $n-1$ -ю секцию, запись в которую еще не закончена.

## 8.11. Рандеву

Модель взаимодействия процессов, названная рандеву, рассматривает синхронизацию и передачу данных как единую

деятельность. Когда процесс А намерен передать данные процессу В, оба процесса должны объявить о своей готовности установить связь, выдав запросы на передачу и прием данных соответственно. Если процесс А выдает заявку на передачу прежде, чем процесс В выдал заявку на прием, то процесс А приостанавливается до выдачи заявки процессом В. И наоборот: если процесс В выдал заявку на прием раньше, чем процесс А на передачу, то приостанавливается процесс В. Таким образом, процессы взаимодействуют только при встрече (рандеву) их заявок на передачу и прием.

В абстрактной записи взаимодействие между процессами записывается так:

```
1   processA {
2       объявление локальной переменной x;
3       . . .
4       B!x;
5       . . .
6   }
7   processB {
8       объявление локальной переменной y;
9       . . .
10      A?y;
11      . . .
12  }
```

Нотация B!x в строке 4 означает, что процесс А передает процессу В значение своей переменной x. A?y в строке 10 означает, что процесс В принимает значение, переданное процессом А, и записывает его в свою переменную y.

Эта запись отражает так называемую синхронную модель рандеву. Запись асинхронной модели мы можем получить, заменив строку 10 на:

```
10      ?y;
```

В синхронной модели оба процесса должны указывать в операторах приема или передачи имя процесса-корреспондента. В асинхронной модели только процесс-передатчик указывает имя процесса-приемника. "Безадресный" оператор приема соответствует идеям структуризации данных и программирования "снизу вверх", развивавшимся автором моделей рандеву и мониторов – К.Хоаром [10]. Асинхронная модель делает возможным разработку библиотечных процессов, которые, во-первых, могут использоваться в разных разработках, а во-вторых, играть роль процессов-серверов, обрабатывающих запросы от разных, параллельно выполняющихся процессов-клиентов.

Асинхронная модель рандеву лежит в основе взаимодействия процессов в языке ADA [6]. Мы не имеем возможности привести здесь полное описание языка (его синтаксис во многом подобен синтаксису языка Pascal) и ограничимся только средствами, интересующими нас в первую очередь. Во всех последующих примерах ключевые слова языка ADA записаны строчными буквами.

Процесс в языке ADA называется задачей и описание задачи состоит из спецификаций задачи и ее тела. Спецификация имеет структуру:

```
task ИМЯ_ЗАДАЧИ is
    < описания входных точек >
end;
```

Тело имеет структуру:

```

task body ИМЯ_ЗАДАЧИ is
    < переменные и операторы >
end ИМЯ_ЗАДАЧИ;

```

В спецификации указываются точки входа задачи для рандеву. Их описания идентичны описаниям процедур: имя и параметры с указанием направления передачи параметров: `in`, `out` или `inout`. В задаче, обращающейся к входной точке, обращение выглядит точно так же, как обращение к процедуре. Однако, выполняется такое обращение иначе. В задаче-приемнике такое обращение обрабатывается оператором приема. В простейшем случае такой оператор имеет вид:

```

accept ИМЯ_ВХОДА ( < параметры > ) do
    < операторы >
end;

```

Оператор приема входит в структуру последовательно выполняемых операторов тела задачи. Если при обращении к данному входу выполнение задачи-приемника еще не дошло до оператора приема, то задача-передатчик блокируется. Если выполнение дошло до оператора приема, но обращения к данному входу не было, блокируется задача-приемник.

В ряде случаев неизвестно заранее, в какой последовательности будут поступать запросы. Для недетерминированного выбора из нескольких возможных запросов используется оператор отбора:

```

select
    < оператор accept >
    < другие операторы >

```



```

or
    < оператор accept >
    < другие операторы >
or
    . . .
else
    < другие операторы >
end;

```

Когда выполнение приемника доходит до оператора отбора, приемник готов выполнить любой из операторов приема, перечисленных среди альтернатив отбора. Если к этому моменту уже поступили обращения к нескольким входам, включенным в отбор, принимается одно из обращений (какое именно – правила языка не определяют). Если обращений нет, то либо выполняется альтернатива `else`, либо (если эта альтернатива не задана) процесс-приемник ожидает.

Операторы `accept`, составляющие альтернативы отбора, могут быть "защищены" условиями. Заголовок оператора в этом случае выглядит так:

```

when <логическое выражение > =>
accept
...

```

Защищенный оператор приема включается в число альтернатив отбора только в том случае, если логическое выражение имеет значение "истина".

Наше краткое описание средств языка само по себе, видимо, недостаточно для его понимания, поэтому проиллюстрируем его примером – все той же задачей производителей–потребителей:

```

1  PROD_CONS: declare;
2  /* пустая спецификация производителя */
3  task PRODUCER is
4      end;
5  /* тело производителя */
6  task body PRODUCER is
7      /* рабочая область для порции */
8      WORK : PORTION;
9      begin
10     loop /* цикл производства */
11         < производство порции в WORK >
12         /* запись порции */
13         PUTPORTION(WORK);
14         /* конец цикла производства */
15     end loop;
16     /* конец тела производителя */
17 end PRODUCER;
18 /* пустая спецификация потребителя */
19 task CONSUMER is
20     end;
21 /* тело потребителя */
22 task body CONSUMER is
23     /* рабочая область для порции */
24     WORK : PORTION;
25     begin
26     loop /* цикл потребления */
27         /* выборка порции */
28         GETPORTION ( WORK );

```

```

29     < обработка порции в WORK >
30     /* конец цикла потребления */
31     end loop;
32     /* конец тела потребителя */
33     end CONSUMER;
34     /* спецификация задачи-сервера */
35     task SERVER is
36         /* описание входных точек сервера */
37         entry GETPORTION(PORT : out PORTION);
38         entry PUTPORTION(PORT : in PORTION);
39         end;
40     /* тело сервера */
41     task body SERVER is
42         /* буфер */
43         BUFFER : array [1..BSIZE] of PORTION;
44         /* индексы для чтения и записи */
45         INCNT, OUTCNT :
46             INTEGER range 1..BSIZE := 1;
47         /* счетчик порций в буфере */
48         PORTCNT :
49             INTEGER range 0..BSIZE := 0;
50         begin
51         loop /* цикл обслуживания */
52             /* выбор из наступивших событий */
53             select when PORTCNT < BSIZE =>
54                 /* если буфер не полон,
55                 обслуживается запись */
56                 accept
57                 PUTPORTION(PORT:in PORTION) do

```

```

58      /* запись */
59      BUFFER[INCNT] := PORT;
60      end;
61      /* модификация счетчиков */
62      INCNT := INCNT mod BSIZE + 1;
63      PORTCNT := PORTCNT + 1;
64      or
65      /* или если буфер не пуст,
66         обслуживается выборка */
67      accept
68      GETPORTION(PORT:out PORTION) do
69      /* выборка */
70      PORT := BUFFER[OUTCNT];
71      end;
72      /* модификация счетчиков */
73      OUTCNT := OUTCNT mod BSIZE + 1;
74      PORTCNT := PORTCNT - 1;
75      end select; /* конец выбора */
76      /* конец цикла обслуживания */
77      end loop;
78      /* конец тела сервера */
79      end SERVER;
80      /* главная процедура */
81      begin
82      /* запуск всех задач */
83      initiate SERVER, PRODUCER, CONSUMER;
84      end.

```

В нашу программу входят:

- главная процедура (строки 80 - 84);
- задача-производитель (строки 2 - 17);
- задача-потребитель (строки 18 - 33);
- задача-сервер (строки 34 - 79), обеспечивающая обмен производителя и потребителя с буфером.

Главная процедура запускает три другие задачи оператором `initiate` (строка 83) и переходит в ожидание. Она завершится, когда завершатся все запущенные ею задачи. Задачи `PRODUCER` и `CONSUMER` не имеют операторов приема, поэтому их спецификации (строки 2 - 4 и 18 - 20) вырожденные – пустые. Тела этих задач содержат простые бесконечные циклы (`loop`), в которых выполняется подготовка или обработка порции и обращение к соответствующей входной точке сервера. Задача `SERVER` является аналогом монитора. В ее спецификации (строки 34 - 39) описаны две входные точки: `GETPORTION` и `PUTPORTION`. Сам буфер является локальным в теле сервера (строка 43), также локальны и индексы чтения и записи (строки 45, 46) и счетчик порций (строки 48 - 49). Выполнение сервера представляет собой бесконечный цикл (строки 51 - 77), в каждой итерации которого обрабатывается одно обращение. Оператор `select` (строки 52 - 75) обеспечивает выбор из обращений: `GETPORTION` или `PUTPORTION`. В зависимости от значения счетчика `PORTCNT` из числа альтернатив может исключаться `GETPORTION` – если буфер пуст или `PUTPORTION` – если он полон. Если к началу очередной итерации обращений нет или есть обращение, которое не позволяет принять защита `when`, сервер ожидает. Обратите внимание на операторные скобки `do ... end`, следующие за операторами `assert` (строки 57 - 60 и 68 - 71). Они ограничивают критическую секцию. Выполнение процесса-передатчика не возобновится до тех пор, пока процесс-приемник не выйдет из критической секции. Мы включили в

критические секции приемов только операторы, непосредственно работающие с параметрами вызовов, так как основное предназначение этой секции – защита параметров от преждевременного их изменения. Остальные операторы, выполняемые в ходе обработки вызовов (модификация индексов и счетчика), выполняются уже вне критической секции.

В нашем примере мы запустили по одному производителю и потребителю. В языке, однако, имеется возможность запускать любое число однотипных задач: как полностью идентичных, так и различающихся по значениям параметров.

Модель рандеву как достаточно универсальный метод взаимодействия позволяет легко реализовать и примитивы взаимного исключения и синхронизации. Двоичный семафор, например, выглядит так:

```
task SEMAPHORE is
  entry P;
  entry V;
end;
task body SEMAPHORE is
  begin
  loop
    accept P;
    accept V;
  end loop;
end SEMAPHOR;
```

В этой задаче операторы приема не являются альтернативными, а выполняются строго последовательно. Если какая-либо внешняя задача

выполнит Р-обращение, то любая задача, выдавшая еще одно Р-обращение, будет заблокирована до тех пор, пока не будет выполнено V-обращение и семафор не войдет в следующую итерацию своего цикла.

Асимметричные рандеву являются дальнейшим развитием идеи мониторов. В большинстве ADA-приложений задачи четко разделяются на задачи-клиенты, выдающие вызовы, и задачи-серверы, их принимающие. Однако, концептуально рандеву являются более универсальным и гибким средством взаимодействия процессов. Обратите внимание на то, что взаимодействующие задачи не используют общих переменных. Это делает язык ADA независимым от конкретной реализации параллельной работы в системе: это может быть однопроцессорная система с разделением времени, мультипроцессорная система с общей памятью или многомашинная система (сеть).

Существенным недостатком модели рандеву является то, что большинство решений, ее использующих, требует введения дополнительных процессов (в наших примерах – задача-сервер или семафор, как отдельная задача). Это увеличивает число переключений процессов и накладные расходы системы.

## **КОНТРОЛЬНЫЕ ВОПРОСЫ**

1. Покажите, что задача синхронизации является частным случаем задачи взаимного исключения.

2. Для каких задач использование единственной общей переменной исключения может быть оправданным?

3. Сопоставьте свойства алгоритмов взаимного исключения, использующих атомарность команд и использующих атомарность обращений к памяти.

4. В состав семафора входит переменная взаимного исключения и скобки критической секции. Почему же потери на занятое ожидание в семафоре не могут быть значительными?

5. Какие ограничения имеются в решении задачи "производители–потребители" методом семафоров?

6. В чем преимущества встраивания критической секции в язык программирования? Покажите, как используются скрытые семафоры для реализации встроенной критической секции.

7. В чем преимущество использования мониторов? Покажите, как используются скрытые семафоры для реализации защищенных процедур.

8. Проблема вложенных вызовов мониторов может быть решена при помощи иерархической дисциплины, описанной в разделе 5.3. Покажите пути такой реализации.

9. Почему при применении групповых мониторов процедуры `read` и `write` не должны быть защищенными?

10. Покажите реализацию задачи "читатели–писатели" с исключением возможности бесконечного откладывания процесса-писателя.

11. В чем преимущества решения задачи "производители–потребители" методами счетчиков событий или секвенсоров перед методом семафоров?

12. Как, используя семафоры, реализовать счетчики событий и секвенсоры?

13. В каких ситуациях процесс, участвующий во взаимодействии по модели рандеву, может быть заблокирован?

14. Объясните реализацию семафора методом рандеву.

## **Глава 9. Системные средства взаимодействия процессов**



В предыдущей главе мы уже фактически затронули эту тему, однако средства, которые мы рассматривали, в основном ограничивались теми, которые реализуются самим программистом или компилятором. Здесь мы сосредоточимся на тех средствах, использование которых поддерживается API ОС.

### **9.1. Скобки критических секций.**

Выделение критических секций как системное средство целесообразно применять для относительно сильно связанных процессов – таких, которые разделяют большой объем данных. Кроме того, поскольку, как мы показали в предыдущей главе, при применении программистом скобок критических секций возможны ошибки, приводящие к подавлению одних процессов другими, важно, чтобы конфликты между процессами не приводили к конфликтам между пользователями. Эти свойства характерны для нитей – параллельно выполняющихся частей одного и того же процесса: они все принадлежат одному процессу – одному пользователю и разделяют почти все ресурсы этого процесса. Следовательно, критические секции целесообразно применять только для взаимного исключения нитей. ОС может предоставлять для этих целей элементарные системные вызовы, функционально аналогичные рассмотренным нами в предыдущей главе `csBegin` и `csEnd`. Когда нить входит в критическую секцию, все остальные нити этого процесса блокируются. Блокировка не затрагивает другие процессы и их нити. Естественно, что такая политика весьма консервативна и снижает уровень мультипрограммирования, но это может повлиять на эффективность только в рамках одного процесса. Программист может самостоятельно организовать и более либеральную

политику доступа к разделяемым ресурсам, используя, например, семафоры, которые будут описаны ниже.

Кроме того, роль таких скобок могут играть системные вызовы типа `suspend` и `release`, первый из которых приостанавливает выполнение нити, а второй – отменяет приостановку.

## 9.2. Виртуальные прерывания или сигналы

Мы уже говорили о виртуальных прерываниях, как о средстве, при помощи которого ОС сигнализирует процессу об окончании асинхронно выполняемой операции ввода-вывода. Расширяя эту концепцию, можно применять виртуальные прерывания для сообщения процессу о любом внешнем по отношению к нему событии. В частности, виртуальное прерывание может использоваться для того, чтобы выдавать синхронизирующий сигнал из одного процесса в другой. ОС может предоставлять в распоряжение процессов системный вызов:

```
raiseInterrupt (pid, intType );
```

где `pid` – идентификатор процесса, которому посылается прерывание, `intType` – тип (возможно, номер) прерывания. Идентификатор процесса – это не внешнее его имя, а манипулятор, устанавливаемый для каждого запуска процесса ОС. Для того, чтобы процесс мог послать сигнал другому процессу, процесс-отправитель должен знать идентификатор процесса-получателя, то есть находиться с ним в достаточно "конфиденциальных" отношениях. Чтобы предотвратить возможность посылки непредусмотренных прерываний, могут быть введены дополнительные ограничения: разрешить посылку прерываний только от процессов-предков к потомкам или ограничить обмен прерываниями только процессами одного и того же пользователя.

Когда процессу посылается прерывание, управление передается на обработчик этого прерывания в составе процесса. Процесс должен установить адрес обработчика при помощи системного вызова типа:

```
setInterruptHandler  
  
    (intType, action, procedure );
```

где `action` – вид реакции на прерывание. Вид реакции может задаваться из перечня стандартных, в число которых могут входить: реакция по умолчанию, игнорировать прерывание, восстановить прежнюю установку или установить в качестве обработчика прерывания процедуру `procedure`, адрес которой является параметром системного вызова.

Разумеется, в системе должны быть определены допустимые типы виртуальных прерываний. Виртуальные прерывания могут генерироваться в следующих случаях:

- завершение или другое изменение статуса процесса-потомка;
- программные ошибки (прерывания-ловушки);
- ошибки в выполнении системных вызовов или неправильные обращения к системным вызовам;
- терминальные воздействия (например, нажатие клавиши "Внимание" или Ctrl+Break);
- при необходимости завершения процесса (системный вызов `kill`);
- сигнал от таймера;
- сигналы, которыми процессы обмениваются друг с другом;
- и т.д.

Если процесс получает прерывание, для которого он не установил обработчик, то процесс должен аварийно завершиться (это устанавливаемый по умолчанию вид реакции на прерывание). Такая установка может показаться чрезмерно жесткой, но вспомните, например,

какова будет реакция системы на реальное прерывание, для которого не определен его обработчик (вектор прерывания в Intel-Pentium).

Еще одно решение, которое должен принять конструктор ОС, – является ли установка обработчика постоянной (до ее явной отмены) или одноразовой (для обработки только одного прерывания). Второй вариант является более гибким, так как каждая процедура обработки прерывания может при необходимости заканчиваться новым системным вызовом `setInterruptHandler`, которым будет задана установка на обработку следующего прерывания этого типа. Это решение можно также переложить на программиста, включив соответствующий параметр в спецификации системного вызова.

Как должна реагировать ОС на посылку прерывания несуществующему процессу? По-видимому, аварийное завершение процесса, выдавшего такое прерывание, может быть нормальной реакцией системы. Возможно, впрочем, и более либеральное решение – завершить вызов `raiseInterrupt` с признаком ошибки. Аналогичный эффект может вызвать выполнение прерывания, для которого в процессе-приемнике установлен специальный режим обработки – недопустимое прерывание.

Как и для реальных прерываний, процесс должен иметь средства запрещения виртуальных прерываний (например, при вхождении в критическую секцию) – всех или выборочно по типам. Для этих целей должны использоваться специальные системные вызовы. Если прерывание запрещено, то его обработка откладывается до разрешения прерываний. Когда обработка разрешается, она выполняется по тому виду реакции, который установлен на момент выполнения (он может отличаться от установленного на момент выдачи прерывания). Среди зарезервированных за ОС типов прерываний обязательно должны быть такие, запретить

которые или переопределить обработку которых процесс не имеет возможности, – обязательно в этом списке должно быть прерывание `kill`.

В большинстве современных ОС (Unix, OS/2 и др.) виртуальные прерывания носят название сигналов и используются прежде всего для сигнализации о чрезвычайных событиях. Сигнальные системы конкретных ОС, как правило, не предоставляют в составе API универсального вызова типа `raiseInterrupt`, который позволял бы пользователю выдавать сигналы любого типа. Набор зарезервированных типов сигналов ограничен (в Unix, например, их 19, а в OS/2 – всего 7), не все из них доступны процессам и для каждого из доступных имеется собственный системный вызов. Недопустимы также незарезервированные типы сигналов. В набор включается несколько (по 3 – в упомянутых ОС) типов сигналов, зарезервированных за процессами, – эти типы и используют взаимодействующие процессы для посылки друг другу сигналов, которые они интерпретируют по предварительной договоренности.

В момент, когда для процесса генерируется виртуальное прерывание, процесс, возможно (в однопроцессорной системе – наверняка), пребывает в неактивном состоянии. Поэтому обработка прерывания откладывается до момента активизации процесса (в порядке очереди к планировщику), а прерывание запоминается в блоке контекста процесса. Как должно обрабатываться виртуальное прерывание, если во время его поступления процесс выполняет системный вызов? Выполнение системного вызова включает в себя как фрагменты кода, выполняемые в привилегированном режиме, так и фрагменты, выполняемые в режиме задачи. Очевидно, что привилегированные фрагменты прерываться не могут – их выполнение может быть связано с изменениями системных структур данных, которые должны выполняться транзакционно (т.е. не должны прерываться). В этом случае пришедшее виртуальное прерывание запоминается в блоке контекста процесса и обрабатывается при переходе процесса из состояния ядра в состояние задачи. Но системный вызов может содержать и непривилегированную часть, к тому же выполняющуюся весьма длительно

(например, ввод с клавиатуры с ожиданием). Разумным решением будет разрешение прерывать такой системный вызов, но в этом случае выполнение прерванного системного вызова может заканчиваться с ошибкой – и процесс должен быть готов к этому.

### **9.3. Модель виртуальных коммуникационных портов**

Большинство средств взаимодействия процессов соответствуют концепции коммуникационных портов – виртуальных устройств, через которые процессы обмениваются данными. Как устройства коммуникационные порты могут "вписываться" в файловую систему в качестве специальных файлов. Такое сведение средств взаимодействия к файловой модели в общем случае обеспечивает три преимущества:

- возможность единообразных операций со средствами взаимодействия и с файлами;
- возможность доступа к удаленным средствам как к удаленным файлам;
- возможность использования единых средств контроля доступа для файловой системы и для коммуникаций.

Концепция коммуникационных портов, однако, в реальных ОС выдерживается далеко не строго. Реально манипулирование далеко не всеми средствами взаимодействия между процессами возможно свести к однотипным операциям. Доступ к удаленным средствам решается методами сетевых модулей ОС. Разграничение доступа в полном объеме мы наблюдали только в AS/400, и то не в рамках файловой системы, а в контексте общей объектно-ориентированной структуры этой системы (см. разд. 13.7). Тем не менее, тенденция к модели портов в той или иной степени наблюдается в современных ОС, прежде всего, в части именования средств взаимодействия. В этом разделе мы рассмотрим

общие свойства средств взаимодействия, называя их виртуальными коммуникационными портами.

Виртуальный коммуникационный порт представляет собой ресурс ОС. Он, разумеется, имеет и физическое представление: структуры данных, области памяти, скрытые семафоры и т.п. Порты, как ресурсы, конструируемые ОС, не имеют жесткого ограничения по количеству – новые порты могут создаваться по мере надобности и уничтожаться, когда необходимость в них отпадает. При использовании порта несколькими процессами один процесс создает порт, а другие – получают доступ к уже существующему порту.

Как и при работе со всяким ресурсом, процесс должен получить доступ к этому порту – открыть его. Системный вызов открытия коммуникационного порта (или его создания) возвращает процессу манипулятор, который процесс использует как идентификатор порта во всех последующих операциях с ним. Порт используется одновременно как минимум двумя процессами, поэтому важно, чтобы манипуляторы у процессов-корреспондентов, взаимодействующих через этот порт, связывались с одним и тем же физическим представлением порта. Возможны два варианта: либо все использующие порт процессы имеют один и тот же манипулятор порта, либо для каждого процесса этот манипулятор индивидуальный. В любом случае ОС поддерживает в ядре таблицу открытых портов (точнее, несколько таких таблиц – по одной для каждого типа средств взаимодействия процессов). В качестве манипулятора может использоваться либо указатель на элемент этой таблицы – тогда манипулятор будет одинаковым для разных процессов, либо указатель на элемент индивидуальной таблицы, входящей в состав контекста процесса, а уже элемент таблицы процесса содержит ссылку на общесистемную таблицу. Очевидно, что второй вариант более надежен, так как исключает случайный доступ к порту. Даже в тех случаях, когда

ОС выдает один и тот же манипулятор нескольким процессам, она требует, чтобы процесс выполнил системный вызов получения доступа к ресурсу.

Для доступа двух процессов к одному и тому же физическому представлению порта в вызове открытия порта необходимо указать параметры, позволяющие системе это физическое представление найти. С точки зрения идентификации порты могут быть именованными или неименованными.

Именованный порт имеет внешнее имя. Системный вызов открытия именованного порта требует указания этого имени в качестве параметра вызова. Пользователи-разработчики взаимодействующих процессов заранее договариваются об используемых именах портов. Система именования портов и открытия именованных портов аналогична файловой системе. Имена средств взаимодействия формируются по соглашениям именования файлов и выглядят как имена файлов, расположенных в специальных каталогах, например: каталог `\shrmem` – для общих областей памяти, каталог `\sem` – для системных семафоров, `\pipe` – для каналов, `\queues` – для очередей.

Неименованный порт внешнего имени не имеет. При создании такого порта системный вызов возвращает его манипулятор – и это единственное, чем располагает процесс для идентификации порта. Манипулятор порта почти наверняка будет разным при разных выполнениях одной и той же программы. Для установления связи между процессами процесс-создатель порта должен передать процессу-корреспонденту манипулятор созданного им порта. Процесс-корреспондент в системном вызове открытия порта указывает идентификатор процесса-создателя и манипулятор порта у процесса-создателя, а в ответ получает манипулятор того же порта для себя. Передача манипулятора процессу-корреспонденту может производиться как передача ресурса от предка к потомку или (если процессы не связаны родством) через именованный порт. Как правило,



неименованные порты используются для связи между процессами – предком и потомком, в этом случае потомок наследует от предка уже открытый коммуникационный порт. Неименованные порты используются для связи между независимыми процессами.

В связи с использованием портов несколькими процессами возникают проблемы закрытия портов. Закончив работу с портом, процесс выполняет системный вызов закрытия. В ОС поддерживается общесистемная таблица портов и обязательным для системы требованием является закрытие при завершении процесса всех портов, которые процесс "забыл" закрыть явным образом. Если с портом работают два процесса, и один из них закрыл порт, а другой обращается к этому порту, то для последнего процесса этот системный вызов заканчивается с признаком ошибки – и это единственно возможное решение. Еще одна проблема – уничтожение физического представления портов. Она может решаться двумя путями: либо порт уничтожается, когда он закрывается последним из использующих его процессов, либо он уничтожается (специальным системным вызовом или простым закрытием) тем процессом, который его создал. Последний случай более привлекательный с точки зрения упорядочения доступа, но он может порождать больше ошибок, когда процессы-корреспонденты обращаются к уже несуществующему порту.

Средства взаимодействия, рассматриваемые нами ниже, являются частными случаями модели виртуальных коммуникационных портов.

## **9.4. Общие области памяти**

Два и более процессов могут использовать одну и ту же физическую область памяти. Наиболее просто это достигается в тех моделях памяти, которые обеспечивают динамическую трансляцию адресов. Напомним, что

каждый процесс имеет собственную таблицу сегментов или страниц, в которой содержится помимо прочего базовый адрес сегмента/страницы в физической памяти. Для разделяемой области памяти создается по элементу в таблице для каждого процесса, ее использующего. В чисто страничной модели, однако, возникают трудности, связанные с тем, что разделяемая область памяти может иметь объем, некратный размеру страницы, обычно в таких случаях разделяемая область выравнивается до границы страницы. В сегментной или сегментно-страничной модели таких проблем нет. Поскольку для каждого процесса разделяемый сегмент описывается своим дескриптором, права доступа к сегменту могут быть установлены различными для разных процессов.

В случае именованных областей памяти один процесс создает общую область памяти:

```
vAddr =  
    createNamedMemorySegment (segmentName ,  
                               segmentSize) ;
```

а второй ее "открывает":

```
vAddr =  
    openNamedMemorySegment (segmentName) ;
```

В этих вызовах `segmentName` – имя области, `segmentSize` – ее размер. Оба вызова возвращают виртуальный адрес общей области памяти в виртуальном адресном пространстве процесса – `vAddr`.

Для неименованной области памяти создание области осуществляется вызовом:

```
vAddr = createMemorySegment (segmentSize) ;
```

а "открытие":

```
vAddr =  
    openMemorySegment (hostAddr , hostPid) ;
```

где `hostAddr` – виртуальный адрес области памяти у процесса-создателя области, `hostPid` – идентификатор процесса-создателя области. Этот вызов возвращает виртуальный адрес области в адресном пространстве процесса, открывшего область.

Разумеется, в составе API имеются системные вызовы "закрытия"/уничтожения общей области памяти.

Разделяемые области памяти, однако, порождают ряд проблем, как для программистов, так и для ОС. Проблемы программистов – те, что рассматривались в предыдущем разделе: взаимное исключение процессов при доступе к общей памяти. Программисты могут дифференцировать права доступа для процессов или организовать взаимное исключение, используя семафоры (см. ниже). Проблемы ОС – организация свопинга. Очевидно, что вероятность вытеснения разделяемого сегмента или страницы должна быть тем меньше, чем больше процессов разделяют этот сегмент/страницу. Если каждый процесс имеет собственный дескриптор разделяемого сегмента или страницы, то учет использования сегмента или страницы (поля `used` и `dirty`) будут вестись по каждому процессу отдельно. ОС должна обрабатывать, например, такой случай: два процесса А и В разделяют сегмент; процесс А произвел запись в сегмент и в его дескрипторе сегмент помечен как "грязный". В то время, когда активен процесс В, принимается решение о вытеснении этого сегмента из памяти. Но в дескрипторе процесса В этот сегмент имеет признак "чистый", поэтому сегмент может быть освобожден в физической памяти без сохранения на внешней памяти и изменения в сегменте, сделанные процессом А, будут утеряны. ОС приходится вести отдельную таблицу разделяемых сегментов, в которой отражать истинное их состояние.

С точки зрения идентификации, разделяемые области памяти могут рассматриваться как виртуальные коммуникационные порты. Для общей области может быть по соглашению между разработчиками

взаимодействующих процессов установлено внешнее имя, которое будет использовано для получения доступа. (В Windows 95, например, такие области называются "файлами отображаемой памяти" – memory mapped file – и для установления доступа к ним используются системные вызовы типа create и open). Для неименованных областей памяти возможна передача селектора (номера в таблице дескрипторов) от одного процесса к другому.

В системах, которые ориентированы на процессор Intel-Pentium, может использоваться то обстоятельство, что адресация возможна через две таблицы дескрипторов: LDT и GDT. За счет этого общее адресное пространство процесса может достигать 4 Гбайт. Из них младшие 2 Гбайт адресуются через LDT, а старшие – через GDT. Глобальная таблица дескрипторов – общая для всех процессов, и именно она может использоваться для доступа к совместно используемой памяти. Размещение в общих виртуальных адресах удобно для системных программ и динамических библиотек, к которым происходят частые обращения из приложений: если эти компоненты ОС находятся в адресном пространстве процесса, то обращения к ним не требуют переключения контекста. Но с другой стороны, это снижает надежность: если системные компоненты доступны для приложения, то они могут быть им испорчены. Поэтому такая "роскошь" может быть допущена только в однопользовательских системах.

## **9.5. Семафоры**

В предыдущей главе мы достаточно подробно рассмотрели сущность и свойства семафоров. ОС может предоставлять семафоры в распоряжение пользователя, как средство для самостоятельного решения задач, требующих взаимного исключения и/или синхронизации. Помимо

основных для семафоров P- и V-операций конкретные семафорные API ОС могут включать в себя расширенные и сервисные функции.

При работе с именованным семафором один из процессов должен создать системный семафор при помощи вызова `createSemaphore`, другие процессы получают доступ к созданному системному семафору при помощи вызова `openSemaphore`. Среди входных параметров этих вызовов имеется внешнее имя семафора, вызовы возвращают манипулятор для семафора, используемый для его идентификации при последующей работе с ним. При окончании работы с системным семафором процесс должен выполнить вызов `closeSemaphore`. Семафор уничтожается, когда он закрыт во всех процессах, его использовавших.

Выполнение операций над семафором может обеспечиваться системным вызовом вида:

```
flag = semaphoreOp(semaphorId, opCode,  
                    waitOption);
```

где `semaphorId` – манипулятор семафора, `opCode` – код операции, `waitOption` – опция ожидания, `flag` – возвращаемое значение, признак успешного выполнения операции или код ошибки.

Помимо основных для семафоров P- и V-операций конкретные семафорные API ОС могут включать в себя расширенные и сервисные функции, такие как безусловная установка семафора, установка семафора с ожиданием его очистки, ожидание очистки семафора. При выполнении системных вызовов – аналогов P-операции, как правило, имеется возможность задать опцию ожидания – блокировать процесс, если выполнение P-операции невозможно, или завершить системный вызов с признаком ошибки.

Во многих современных ОС наряду с семафорами "в чистом виде" API представляет те же семафоры и в виде "прикладных" объектов – объектов взаимного исключения и событий. Хотя содержание этих

объектов одно и то же – семафор, ОС в отношении этих объектов представляет для прикладных процессов специфическую семантику API, соответствующую задачам взаимного исключения и синхронизации.

Все современные ОС предоставляют прикладному процессу возможность работать с "массивами семафоров", то есть задавать список семафоров и выполнять операцию над всем списком, например, ожидать очистки любого в заданном списке. Наиболее развито это средство в ОС Unix, где имеется возможность выполнять за один системный вызов `semop` (аналог нашего `semaphoreOp`) сразу нескольких различных операций над несколькими семафорами, причем весь список операций выполняется как одна транзакция.

Неименованные семафоры обычно используются как средство взаимного исключения и синхронизации работы нитей одного процесса.

## 9.6. Программные каналы

Программный канал по-английски называется `pipe` (труба), и это весьма удачное название. Канал действительно можно представить как трубопровод пневматической почты, проложенный между двумя процессами, как показано на рисунке 9.1. По этому трубопроводу данные передаются от одного процесса к другому. Как и трубопровод, программный канал однонаправленный (хотя, например, в Unix одним системным вызовом создаются сразу два разнонаправленных канала). Как и трубопровод, программный канал имеет собственную емкость: данные, записанные в канал, не обязательно должны немедленно выбираться на противоположном его конце, но могут накапливаться в канале, пока это позволяет его емкость. Как и трубопровод, канал работает по дисциплине FIFO: первый вошел – первый вышел.

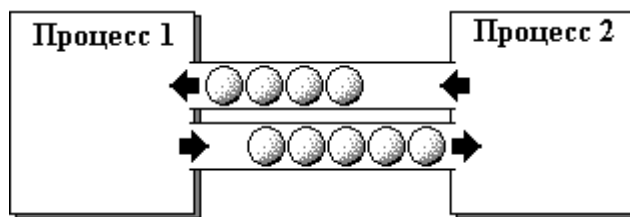


Рисунок 9.1 Программные каналы

Из всех средств взаимодействия между процессами программные каналы лучше всего вписываются в модель виртуальных коммуникационных портов. Канал для процесса практически аналогичен файлу. Специальные системные вызовы типа `createPipe`, `openPipe` используются для создания канала и получения доступа к каналу, а для работы с каналом используются те же вызовы `read` и `write`, что и для файлов, и даже закрытие канала выполняется файловым системным вызовом `close`. При создании канала для него создается дескриптор, как для открытого файла, что позволяет работать с ним далее, как с файлом. Канал, однако, представляет собой не внешние данные, а область памяти. Для канала выделяется память в системной области, что может ограничивать емкость канала.

Наиболее часто используются неименованные каналы как средство связи между родителем и потомком. Операция создания неименованного канала возвращает два файловых манипулятора: для чтения и для записи. Процесс-предок передает эти манипуляторы процессу-потомку. Если связь между предком и потомком однонаправленная, то каждый из них закрывает канал по одному из манипуляторов. Например, если данные передаются только от предка к потомку, то предок после передачи манипуляторов закрывает канал на чтение, а потомок – на запись. Пример установления такой связи в ОС Unix приведен в главе 11, во фрагменте программы командного интерпретатора.

С точки зрения реализации канал представляет собой классический вариант задачи "производитель–потребитель": один процесс пишет в канал данные, другой – читает их из канала. Если при попытке записи данных в канал обнаруживается, что канал полон, пишущий процесс блокируется до освобождения места в канале, если при попытке чтения данных обнаруживается, что канал пуст, читающий процесс блокируется до появления в канале данных. Внутренним механизмом ОС, обеспечивающим синхронизацию в таких ситуациях, является, конечно же, семафор.

Именованные каналы представляют собой удобное средство клиент/серверных коммуникаций. Именованные каналы в некоторых ОС (например, OS/2) существенно отличаются от неименованных. Именованные каналы ориентированы в этих системах прежде всего на взаимодействие процессов в сетевой среде (или, точнее, для них прозрачно, находятся ли оба процесса на одном компьютере или в разных узлах сети). Такой канал двунаправленный, то есть к нему возможны обращения и для чтения, и для записи. Данные в канале могут передаваться как потоком байт, так и сообщениями. В последнем случае каждое сообщение снабжается заголовком, в котором указывается его длина. К одному концу канала постоянно подключен процесс, его создавший, – владелец или сервер, к другому концу могут подключаться различные процессы-клиенты, таким образом, обмен данными по именованному каналу между процессами, ни один из которых не является владельцем канала, невозможен. При создании канала (системный вызов `createNamedPipe`) указывается максимально возможное число клиентов, которые могут быть одновременно подключены к каналу (это число, впрочем, может и не ограничиваться). Когда владелец создает канал, последний находится в так называемом отключенном состоянии. Системным вызовом `connectNamedPipe` владелец переводит канал в



ждущее состояние. Теперь процессы-клиенты могут подключаться к другому концу канала при помощи файловых системных вызовов `openNamedPipe`. Канал, открытый хотя бы одним клиентом, считается подключенным, сервер и подключенные клиенты могут работать с ним, используя вызовы файлового API. Клиенты отключаются от канала вызовом `close`, в распоряжении владельца есть вызов `disconnectNamedPipe` – разрыва канала. Помимо обычных вызовов файлового обмена для работы с именованным каналом в составе API могут быть специальные системные вызовы, обеспечивающие выполнение сложных транзакций на именованном канале, например: `transactNamedPipe` – взаимный обмен данными (вывод и ввод) за одну операцию, `callNamedPipe` – обеспечивает также открытие канала, взаимный обмен, закрытие канала. Кроме того, к именованному каналу или к нескольким именованным каналам может быть подключен семафор, который сбрасывается при изменении состояния канала (заполнен – не заполнен, пуст – не пуст).

## 9.7. Очереди сообщений

Очереди воплощают модель взаимодействия процессов "много отправителей – один получатель". Эту модель часто называют почтовым ящиком (mailbox) из-за сходства с почтовым ящиком, висящим на дверях каждой квартиры.

Процесс-получатель является владельцем очереди, он создает очередь, а остальные процессы получают к ней доступ, "открывая" ее. Очередь обычно имеет внешнее имя. Передача данных в очереди происходит всегда сообщениями, причем каждое сообщение имеет заголовок и тело. Заголовок всегда имеет фиксированный для данной

системы формат. В него обязательно входит длина сообщения, а другая информация зависит от спецификаций конкретной системы: это может быть приоритет сообщения, тип сообщения, идентификатор процесса, пославшего сообщение, и т.п. Тело сообщения интерпретируется по правилам, устанавливаемым самими процессами: отправителем и получателем. Заголовок и тело представляют собой существенно разные структуры данных и располагаются в разных местах в памяти. Собственно очередь (чаще всего – линейный список) ОС составляет из заголовков сообщений. В элементы очереди включаются указатели на тела сообщений, располагающиеся в памяти системы или процессов (об этом – ниже).

Как правило, возможности процесса-получателя сообщений не ограничиваются чтением по дисциплине FIFO, ему предоставляется более богатый выбор дисциплин: LIFO, по приоритету, по типам, по идентификаторам отправителя и т.п. В распоряжении владельца имеются также средства определения размера очереди, а возможно, и просмотра очереди – неразрушающего чтения из нее. В распоряжении процесса-отправителя имеется только вызов типа `sendMessage` – посылки сообщения в очередь. Если при попытке процесса послать сообщение обнаруживается, что очередь заполнена, процесс-отправитель блокируется. Это, впрочем, довольно редкий случай, так как системные ограничения на размер очередей никогда не бывают слишком жесткими. Процесс-получатель блокируется при попытке читать сообщение, когда очередь пуста.

Существенным вопросом при конструировании механизма очередей является вопрос о включении или невключении в ОС системной буферизации сообщений. При включении такого средства (рисунок 9.2.) тело посылаемого сообщения копируется в системную область памяти, а при чтении – копируется из нее в адресное пространство процесса-получателя.

При отсутствии системной буферизации тела сообщений хранятся в общей для отправителя и получателя памяти, а передается только указатель на тело сообщения (рисунок 9.3.). В первом случае выполняются дополнительные пересылки, затрачивается дополнительная память и вводятся более жесткие ограничения на объем сообщений, но достигается надежность передачи и значительно более простой интерфейс процессов. Во втором случае значительно экономится память, но сами процессы должны заботиться об управлении совместно используемой памятью и о сохранности сообщений в ней. При отсутствии системной буферизации сообщений применяются обычно два метода передачи тела сообщения: либо процесс-отправитель помещает тело сообщения в отдельный разделяемый сегмент, получает у ОС манипулятор этого сегмента для процесса-получателя и передает этот манипулятор в составе сообщения; либо для всех сообщений выделяется одна общая область памяти с общим манипулятором и для размещения сообщения в нем используются системные вызовы выделения памяти в куче.

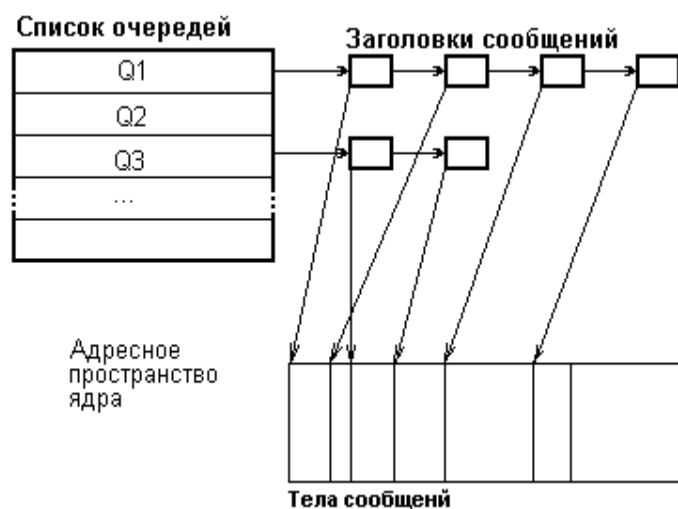


Рис.9.2. Размещение сообщений в адресном пространстве ядра

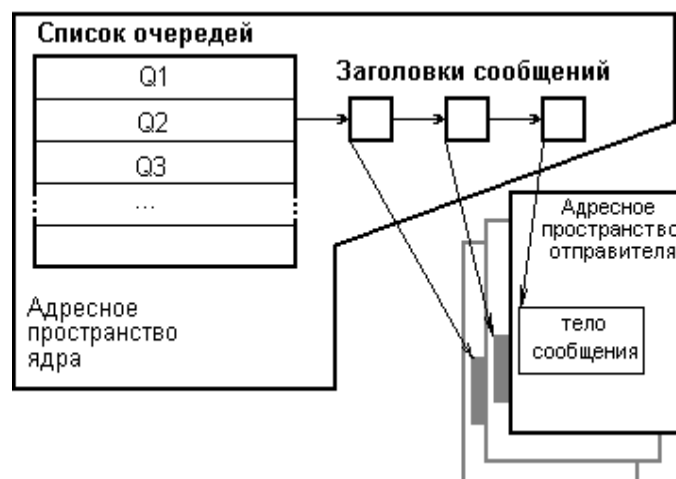


Рис.9.3. Размещение сообщений в адресном пространстве процесса-отправителя

### Контрольные вопросы

1. Почему системные вызовы – скобки критических секций применяются для нитей, но не для процессов?
2. В чем сходство и в чем различия между сигналами и реальными прерываниями?
3. Процесс, которому посылается сигнал, как правило, в момент отправки неактивен. Как поступает ОС с сигналом в таком случае?
4. Опишите различия между именованными и неименованными программными средствами взаимодействия процессов.
5. Общие области памяти могут располагаться либо в перекрывающейся части виртуальных адресных пространств процессов, либо в изолированных частях виртуальных адресных пространств. Каким образом реализуется тот и другой метод размещения? Сопоставьте их достоинства и недостатки.
6. Какими внутренними механизмами обеспечивается защита от записи в заполненный программный канал и защита от чтения из пустого программного канала?

7. Покажите, как представить семафор в виде "переменной взаимного исключения" и "события".

8. Каким образом используются скрытые семафоры во внутренней реализации механизма очередей?

9. Покажите, что задачи взаимного исключения и синхронизации могут быть решены при помощи очередей сообщений.

## **Глава 10. Защита ресурсов**

### **10.1. Общие требования безопасности**

В системе ценностей современного мира информация занимает одно из ведущих мест и рейтинг ее в этой системе возрастает с течением времени почти экспоненциально. Как всякая ценность, информация должна быть надежно защищена. Защита информации достигается комплексом мер безопасности (security), включающим в себя как обеспечение целостности (непротиворечивости) и сохранности информации, так и контроль доступа к ней.

Требования к безопасности – целостности, сохранности и конфиденциальности пользовательских и системных программ и данных, а также к потреблению ресурсов пользователями в пределах установленных им бюджетов всегда являлось одним из важнейших в вычислительных системах. В обеспечении этих требований был некоторый период "тяжелых времен" – в начале бума персональных компьютеров, что было связано с внедрением в сферу полупрофессионального использования вычислительной техники большого числа неподготовленных лиц с задачами, ценность которых не стоила расходов по обеспечению их безопасности. Но с неизбежным разделением сфер использования

компьютеров на персональную и производственную вопросы безопасности в производственной сфере не только восстанавливают свои позиции, но и приобретают все больший вес в связи с развитием компьютерных коммуникаций. В настоящее время все сколько-нибудь серьезные пользователи в производственной сфере готовы платить за гарантии безопасности в своей работе – как материальными затратами, так и некоторым снижением эффективности выполнения своих приложений.

Целостность информации – обеспечение непротиворечивости данных. Правила целостности определяются законами прикладной области, к которой относится информация, так называемыми, бизнес-правилами. Поскольку ОС является универсальным программным обеспечением, предназначенным для поддержки выполнения процессов обработки данных практически любых прикладных областей, поддержание целостности прикладных данных не входит в ее функции. Целостность, определяемая пользователем, поддерживается средствами промежуточного программного обеспечения, наиболее развиты эти средства в системах управления базами данных. Целостность же, за которую отвечает ОС – это целостность общей структуры хранения информации, прежде всего она касается файловой системы. Целостность файловой системы состоит в соответствии метаданных файловой (дисковых и файловых справочников) системы реальному ее состоянию. В задачи ОС входит также обеспечение сохранности данных при ошибках. Ошибок не должно быть при правильном функционировании всех звеньев вычислительной системы, но такое функционирование относится к области идеалов. Источниками ошибок являются ошибки в системном программном обеспечении, сбои оборудования и неправильные (иногда – злонамеренные) действия пользователей. "Первые – возможны, вторые – неизбежны, третьи – гарантированы" [4] Применительно к данным на внешней памяти мы уже рассматривали методы защиты целостности и сохранности в разделах

главы 7, такие методы (избыточность, резервное копирование, дополнительные указатели) применимы и к другим ресурсам вычислительной системы.

Контроль доступа – обеспечение доступа к информации только тому, кто имеет на это право. Этот аспект безопасности является основным предметом рассмотрения в данной главе. В современных системах обработки данных применяется обязательное или избирательное управление безопасностью.

Основными понятиями обязательного управления безопасностью являются уровень секретности и уровень доступа. Каждый объект, включенный в систему защиты, имеет некоторый уровень секретности (например: совершенно секретно, секретно, для служебного пользования и т.д.). Каждый пользователь, получающий доступ к защищенным ресурсам, имеет некоторый уровень допуска. Число уровней допуска равно числу уровней секретности. Если обозначить уровни секретности и уровни допуска числовыми кодами таким образом, чтобы больший числовой код соответствовал большей секретности или более высокому допуску, то правила предоставления разрешений на доступ к ресурсам можно сформулировать следующим образом:

- пользователь получает разрешение на чтение объекта только в том случае, если его уровень допуска равен уровню секретности объекта или больше него;
- пользователь получает разрешение на запись в объект или модификацию объекта только в том случае, если его уровень допуска равен уровню секретности объекта.

Другими словами второе правило можно сформулировать так: любая информация, записанная пользователем с уровнем допуска  $L$ , получает уровень секретности  $L$ . Таким образом, например, пользователь, имеющий допуск к совершенно секретной информации, не может записать

информацию в открытый документ для служебного пользования (документ с более низким уровнем секретности), так как это может нарушить безопасность.

Избирательное управление безопасностью базируется на правах доступа. В случае избирательного управления каждый пользователь обладает определенными правами доступа к определенным ресурсам. Права доступа разных пользователей к одному и тому же ресурсу могут различаться. Избирательное управление часто базируется на принципе владения. В этом случае у каждого ресурса имеется владелец, который обладает всей полнотой прав доступа к ресурсу. Владелец определяет права доступа к своему ресурсу других пользователей.

Министерством обороны США разработана общая классификация уровней безопасности, которая применяется и во всем мире. Эта классификация определяет четыре класса безопасности (в порядке возрастания):

- D – минимальная защита;
- C – избирательная защита (с подклассами C1 и C2;  $C1 < C2$ );
- B – обязательная защита (с подклассами  $B1 < B2 < B3$ );
- A – проверенная защита (с математическим доказательством адекватности).

На сегодняшний день некоторые компьютерные системы в бизнесе удовлетворяют требованиям класса B1, однако большинство коммерческих применений компьютерных систем ограничиваются требованиями класса C2.

Основные требования класса C2 сводятся к следующим:

- владелец ресурса должен управлять доступом к ресурсу;



- ОС должна защищать объекты от несанкционированного использования другими процессами (в том числе и после их удаления);
- перед получением доступа к системе каждый пользователь должен идентифицировать себя, введя уникальное имя входа в систему и пароль; система должна быть способной использовать эту уникальную информацию для контроля действий пользователя;
- администратор системы должен иметь возможность контроля (audit) связанных с безопасностью событий, доступ к этим контрольным данным должен ограничиваться администратором;
- система должна защищать себя от внешнего вмешательства типа модификации выполняющейся системы или хранимых файлов.

Сертификация на соответствие уровню безопасности – длительный процесс, и ему подвергается не только программное обеспечение, а весь комплекс средств системы обработки данных, включая аппаратные, системные и прикладные программные средства, каналы связи, организацию эксплуатации системы и т.д. Но операционная система может оцениваться на предмет того, может или не может она поддерживать функционирование системы обработки данных, соответствующей определенному уровню.

## **10.2. Объектно-ориентированная модель доступа и механизмы защиты**

В главе 7 мы рассмотрели модель управления доступом применительно к файловой системе. Управление доступом к файлам является частным случаем более общей модели управления доступом

субъектов к объектам. Другой частный случай мы рассмотрели в главе 3 применительно к памяти. Модель рассматривает объекты – элементы системы, которым требуется защита, и субъекты – активные элементы, стремящиеся получить доступ к объектам. Типичный пример объекта – файл, типичный пример субъекта – процесс. Элемент, выступающий в одной ситуации в роли субъекта, в другой ситуации может выступать в роли объекта. Так, например, необходимо обеспечивать защиту адресных пространств одних процессов (объектов) от других процессов (субъектов).

Механизмы защиты должны обеспечивать ограничение доступа субъектов к объектам: во-первых, доступ к объекту должен быть разрешен только для определенных субъектов, во-вторых, даже имеющему доступ субъекту должно быть разрешено выполнение только определенного набора операций.

Для обеспечения защиты могут применяться следующие механизмы:

- кодирование объектов;
- сокрытие местоположения объектов;
- инкапсуляция объектов.

Кодирование предполагает шифрование информации, составляющей объект. Любой субъект может получить доступ к информации, но воспользоваться ею может лишь привилегированный субъект, знающий ключ к коду. Другой вариант защиты через кодирование предполагает, что расшифровка информации производится системными средствами, но только для привилегированных субъектов. Кодирование не защищает объект от порчи, поэтому оно может использоваться только для защиты узкого класса специфических объектов или в качестве дополнительного средства в сочетании с другими механизмами защиты. Рассмотрение способов кодирования не входит в задачи нашего пособия, оно должно происходить при изучении курса "Защита информации".

Соккрытие местоположения объекта предполагает, что адрес объекта в памяти системы известен только тем субъектам, которые имеют право доступа к объекту. Такие привилегированные субъекты могут выполнять любые операции над объектом. Непривилегированные субъекты могут запрашивать доступ к объекту и получать некий внутренний идентификатор объекта. Используя этот идентификатор, они могут выполнять над объектом ограниченный набор операций, но не непосредственно, а обращаясь к привилегированным субъектам. Примером такого механизма является соккрытие дескрипторов ресурсов. Местоположение (адрес в памяти) дескрипторов известно ОС, прикладные же процессы получают манипулятор ресурса, который, как правило, адресует дескриптор только косвенно (через таблицы). Если соккрытие является единственным механизмом защиты, то ее нельзя назвать непроницаемой. Субъект может получить адрес объекта случайно или намеренно (получив доступ к внутренней документации).

Инкапсуляция предполагает полное закрытие для субъектов возможности оперирования с внутренним содержимым объектов. Субъект даже не должен знать структуры этого содержимого. Для каждого объекта, однако, определено множество допустимых операций над ним, которые реализуются монитором объекта. Объект, таким образом, предстает в виде "черного ящика" с четко специфицированными входами и выходами и с абсолютно непроницаемыми стенками. Механизм инкапсуляции может обеспечивать наиболее полную защиту, но его реализация требует решения двух важных вопросов: во-первых, как обеспечить "непрозрачность стенок"; во-вторых, как принимать решения о предоставлении доступа или об отказе в нем.

Непроницаемость ящика обеспечивается чаще всего средствами защиты памяти. Область памяти, в которой расположен объект, делается недоступной для любых субъектов, кроме монитора, реализующего

операции над объектом. Как мы знаем (см. главу 3), защита памяти поддерживается аппаратными средствами вычислительных систем и защита объектов, таким образом, представляется реализованной на аппаратном уровне. Надежность такой защиты, однако, в значительной степени иллюзорна. Во-первых, память также представляет собой объект, нуждающийся в защите. От того, насколько сама защита памяти защищена от перепрограммирования ее произвольным субъектом, зависит эффективность всей системы защиты в целом. Во-вторых, аппаратно поддерживается только конечное число уровней защиты памяти, на практике же используются только два уровня: доступ к ограниченному подмножеству адресов и полный доступ. Если в системе имеется большое количество объектов разного типа, то каждый тип обеспечивается своим монитором. Если все мониторы работают с полным доступом, то нет гарантии в том, что монитор в результате, например, ошибки в нем не воздействует на объект, не предусмотренный в данной операции.

Наиболее надежным образом защита памяти достигается при помощи изоляции адресных пространств процессов и мониторов. Если таблицы дескрипторов ресурсов и сами ресурсы располагаются в отдельных адресных пространствах, то процесс просто не может получить к ним доступа, а может воздействовать на них только косвенно, передавая в системном вызове индекс элемента в недоступной для него таблице. Адресные пространства различных мониторов тоже могут быть изолированы друг от друга, что (при надежной изоляции) исключит воздействие ошибки в мониторе на другие ресурсы.

Для каждого объекта определено множество допустимых для него операций. Применительно к файлам, например, возможны в общем случае следующие операции:

- Read – получение информации из объекта;
- Write – обновление информации в объекте;

- Append – добавление в объект новой информации (не изменяя старой);
- Execute – интерпретация объекта как исполняемого кода;
- Delete – уничтожение объекта;
- Getinfo – получение информации об объекте;
- Setinfo – установка информации об объекте;
- Privilege – установка прав доступа к объекту (частный случай Setinfo, по понятным причинам выделенный в отдельную операцию).

Неизбыточный набор операций повышает надежность объекта, лишая потенциального "взломщика" возможностей воздействовать на объект. Так, например, для объекта "программа" могут отсутствовать операции Read и Write. Действия всех компьютерных вирусов заключаются в том, что они читают программный файл как файл данных и дописывают в него (как в файл данных) коды, выполняющие несанкционированные действия. Если программу невозможно читать и писать, то у вируса просто нет возможности "заразить" ее своим кодом.

Перечень всех операций, допустимых для данной пары субъект–объект составляет привилегию доступа (access privilege) или право доступа (access right) данного субъекта к данному объекту.

Каждая пара субъект–объект при каждом акте доступа взаимодействует в определенном режиме доступа (access mode). Условием разрешения доступа является совпадение запрошенного субъектом режима доступа с его привилегиями по отношению к запрошенному объекту.

Проведение политики контроля доступа включает в себя процедуры аутентификации и авторизации, показанные на рисунке 10.1.



многопользовательскую, возможности защиты ресурсов в той, ОС у которой эти свойства заложены в ядро, принципиально большие.

Помимо имени и пароля, в профиль могут входить настройки интерфейса рабочего места, процедура, автоматически выполняемая при начале пользователем сеанса, установки библиотек и каталогов для поиска и т.д. В профиль может также входить "бюджетная информация" (account) – сведения о правах доступа и полномочиях пользователя. Даже в системах защиты, ориентированных на списки контроля доступа, некоторые данные бюджета все равно содержатся в профиле пользователя: его имя и пароль, его полномочия, список групп, к которым он принадлежит. (Некоторые из составляющих бюджета объясняются ниже.) Профили пользователей должны быть одними из наиболее строго защищаемых объектов в системе.

При входе пользователя в систему или при запуске приложения от его имени выполняется аутентификация (authentication). Эта процедура состоит в представлении пользователя системе при установлении связи с ней и подтверждении его подлинности. Подтверждение подлинности выполняется паролем. При выполнении аутентификации процесс входа в систему находит имя пользователя в базе профилей и сверяет введенный пользователем пароль с паролем, хранящимся в профиле. Как правило, пароль хранится в профиле пользователя в закодированном виде, причем используется кодирование на основе хеширования, не допускающее восстановление пароля по его хеш-коду. (Если пользователь забыл свой пароль, то даже системный администратор не может сообщить ему его пароль, а может только предоставить ему возможность назначить новый пароль.) Поэтому введенный пользователем пароль кодируется по тому же алгоритму и сравниваются уже хеш-коды паролей. При подтверждении системой легальности пользователя выбирается идентификатор безопасности для этого пользователя (косвенный указатель на бюджет пользователя) и этот идентификатор включается в контексты всех

процессов, создаваемых данным пользователем. Естественно, что пользователь (или приложение), не прошедший процедуру аутентификации, к работе не допускается.

В системах обычно предусматривается возможность входа в систему пользователя-гостя (guest). Такой пользователь имеет доступ только к полностью открытым ресурсам и не имеет никаких полномочий.

После входа в систему пользователь (или работающее от его имени приложение) может запрашивать доступ к ресурсам. Каждый случай получения пользователем тех ресурсов, которые включены в число защищаемых, сопровождается процедурой авторизации (authorization), которая заключается в проверке того, может ли данный пользователь выполнять запрошенное действие над данным объектом. В процессе выполнения авторизации привлекается информация безопасности, связанная с объектом, и информация безопасности, связанная с пользователем, и выносится решение о предоставлении или отклонении доступа. В объектно-ориентированных системах процедура авторизации может быть решена единообразно. Любая операция получения ресурса (getResource) должна сопровождаться единообразной проверкой полномочий и привилегий субъекта по отношению к данному объекту.

### **10.3. Представление прав доступа**

Полный набор прав доступа для всех субъектов и всех объектов может быть представлен в виде матрицы доступа, строки которой соответствуют субъектам, а столбцы – объектам (или наоборот). На пересечениях строк и столбцов указываются права доступа для данной пары. Права доступа могут задаваться, например, в виде битовых строк с позиционными кодами. На рисунке 10.2 представлен пример такой матрицы.



объекты субъекты	file1	file2	dir1	file3	docA	docB	docC	docD	dir2	file4	pic1
adm	rwx	rwx	rwx	rwx	rwx	rwx	rwx	rwx	rwx	rwx	rwx
pgm1	---	---	---	r-x	---	---	---	---	rwx	rwx	---
pgm2	rwx	rwx	rwx	r-x	---	---	---	---	rwx	rwx	---
pgm3	rwx	rwx	rwx	---	---	---	---	---	---	---	---
op1	---	---	rwx	r-x	---	---	---	---	---	---	---
op2	---	---	rwx	r-x	---	---	---	---	---	---	---
op3	---	---	rwx	r-x	---	---	---	---	---	---	---
us1	--x	---	r--	---	rwx	rwx	---	r--	---	---	---
us2	---	---	r--	---	r--	r--	---	r--	---	---	---
us3	--x	---	r--	---	r--	r--	rwx	rwx	---	---	rwx

Рисунок 10.2 Матрица доступа

Как видно даже из рисунка 10.2, в матрице присутствует избыточность – пустые клетки, так как некоторые объекты недоступны для некоторых субъектов. В многопользовательских системах же с большим количеством как объектов, так и субъектов, во-первых, размер матрицы может быть чрезвычайно большим, во-вторых, сама матрица наверняка будет очень сильно разрежена. Поэтому матричное представление прав доступа является неэффективным. Матрица может быть представлена либо в виде списков привилегий (privileges list), либо в виде списков управления доступом (rights list). Списковые структуры позволяют экономить память за счет исключения из матрицы позиций с пустыми значениями доступа.

В первом случае в системе существует таблица субъектов и с каждым элементом этой таблицы связывается список объектов, к которым субъект имеет доступ (рисунок 10.3).

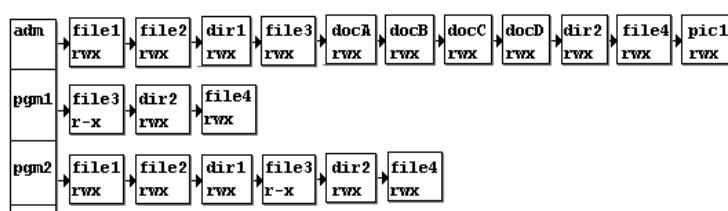


Рисунок 10.3 Списки привилегий

Во втором случае имеется таблица объектов, с элементами которой связаны списки субъектов (рисунок 10.4).

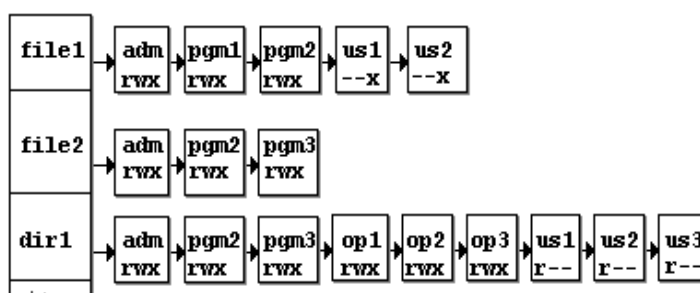


Рисунок 10.4 Списки управления доступом

## 10.4. Дополнительные возможности

Среди действий, которые может выполнять пользователь, есть и такие, которые не связаны с конкретным объектом, а относятся к большой группе объектов или даже ко всей системы в целом, например, выполнение тех или иных команд, создание объектов определенного типа и т.п. Возможность выполнять такие действия иногда называют полномочиями (authority), для присваивания и учета полномочий существуют специальные механизмы. Независимо от того, как представляется информация о правах доступа субъектов к объектам, список полномочий связывается обязательно с профилем пользователя, так как нет возможности связать его с объектом. Впрочем, в системах, последовательно реализующих объектно-ориентированный подход, полномочия, точнее – средства, через которые они реализуются (команды, системные вызовы) сами являются объектами, для доступа к ним назначаются права, как и для других объектов, поэтому здесь нет необходимости выделять их в отдельную категорию.

Для большинства задач управления доступом может оказаться удобной интеграция некоторого подмножества объектов и связанных с

ними подмножеств возможностей в единый элемент, называемый доменом (domain). Если несколько субъектов имеют доступ к одному и тому же набору объектов, причем, с одинаковыми правами, то говорят, что субъекты работают в одном домене. Домены представляют собой удобное средство для сокращения представления информации о защите: в списке возможностей, например, перечисление ряда объектов может быть заменено идентификатором домена, который эти объекты составляют. Домены могут быть пересекающимися. Субъект может работать в нескольких доменах одновременно. Концепция доменов широко используется в современных ОС.

Различаются домены системные и специфицируемые. Системные домены являются предустановленными и неизменяемыми. С системными доменами связываются классы пользователей: принадлежность пользователя к тому или иному классу разрешает пользователю работать в том или ином домене. Обычно в системные домены входят объекты и соответствующие полномочия позволяющие, например, выполнять управление системой. Простейший пример пользовательских классов – обычный пользователь и привилегированный (суперпользователь). Домен привилегированного пользователя охватывает все объекты системы со всеми правами доступа к ним. Предустановленный домен обычного пользователя разрешает ему доступ к такому подмножеству объектов и возможностей, оперируя которыми, он не может вывести из строя ОС или повредить другим пользователям. Дополнительные права обычного пользователя обеспечиваются специфицируемыми доменами и персональными разрешениями.

Специфицируемые домены могут создаваться и конфигурироваться динамически. Во всех ОС пользователи, разделяющие специфицированный домен, объединяются в группы. ОС поддерживает список групп и с каждой группой связывает отдельный домен. В примере

раздела 7.7 мы видели, что Unix различает "группу, в которую входит владелец файла". В более развитых системах защиты пользователь может включаться в несколько групп, и домены групп могут пересекаться. Состав пользователей в группе, состав и возможности домена группы могут изменяться администратором или теми пользователями, которые имеют на это право. Группам могут даваться также и полномочия. С точки зрения представления информации о безопасности группа выступает как один пользователь, и для каждой группы создается собственная запись в базе профилей. Вместо того, чтобы в список прав включать отдельные записи для всех членов какой-то группы, в список заносится единственная запись для всей группы. Имеются обычно также и системные, предустановленные группы – такие, как системные администраторы, операторы и т.п., но состав системных групп может меняться динамически.

Объединение объектов составляет группу объектов, права доступа в которой ко всем объектам одинаковы. Права могут быть разные для разных пользователей и групп, но каждый пользователь имеет одни и те же привилегии в отношении всех объектов группы. Так же, как и группы пользователей, группы доступа позволяют сократить объем информации по безопасности, представляя всю группу доступа как один объект.

Еще одним средством сокращения объема такой информации являются интегрированные права, сформированные из нескольких элементарных прав, например, право данных – права читать и изменять данные.

Среди стандартных прав может быть также право Exclude, явный запрет на доступ к объекту. Записи о правах Exclude размещаются первыми в списках информации о безопасности, таким образом, при поиске они находятся в первую очередь.

В разделе 10.2 мы упоминали о возможности работы в системе "гостя". "Гость" не может иметь никаких индивидуальных или групповых

прав, так как он не зарегистрирован в базе профилей, но некоторые объекты в системе могут быть для него доступны. Говоря шире, в системе могут быть объекты, доступные для всех. Права доступа "для всех" называются публичными (public), они реализуются либо в специальной группе пользователей PUBLIC (гости получают идентификатор безопасности этой группы), либо специальным разделом публичных прав в списках прав объектов и групп доступа.

С учетом перечисленных компонентов системы безопасности процедура авторизации выполняется в такой последовательности:

- поиск в индивидуальных правах пользователя;
- поиск в групповых правах пользователя;
- поиск в публичных правах.

Поиск прекращается, если на любом из этапов будут найдены права, соответствующие запрошенному доступу, доступ разрешается. Поиск прекращается, если на любом из этапов будет найдено право Exclude, доступ не разрешается. Если права, соответствующие запрошенному доступу, не найдены, доступ не разрешается. Индивидуальные права, таким образом, имеют приоритет над групповыми, а групповые – над публичными.

В ряде случаев пользователю может понадобиться выполнить какую-либо корректную задачу, включающую в себя доступ к тем системным объектам, к которым процесс права доступа не имеет. Для выполнения этой задачи ОС предоставляет в распоряжение пользователя процесса программы-утилиты, которые гарантируют корректность манипуляций с защищенными объектами. Но если такая утилита будет выполняться в контексте процесса пользователя, то есть с его идентификатором безопасности, то в доступе ей будет отказано. Для решения этой проблемы применяется так называемый адаптивный доступ (adopted access). В

дескрипторе исполняемого модуля – системной утилиты сохраняется признак, по которому ОС на время выполнения (только не время выполнения!) такого модуля предоставляет процессу, его вызвавшему, те же права, которые имеет создатель утилиты (привилегированный пользователь). Поскольку права предоставляются только на время выполнения утилиты, пользователь, вызвавший ее, этими правами воспользоваться не может.

Еще одно требование к системе безопасности – возможность контроля связанных с безопасностью событий, таких как вход в систему, попытки доступа к объектам (разрешенные или отклоненные), попытки использования полномочий и т.д. Контроль заключается в том, что информация о таких событиях сохраняется в системном журнале аудита и может быть впоследствии проанализирована системным администратором. Администратор имеет возможность задавать события и выбирать объекты, подлежащие контролю. Такой контроль обеспечивается списками контроля доступа, которые подобны спискам прав. Элемент такого списка для объекта содержит вид доступа, который подлежит протоколированию, и результат авторизации. После любого выполнения процедуры авторизации происходит поиск в списке контроля доступа, и если найден элемент, в котором вид доступа и результат авторизации совпадают с произошедшим, в журнал аудита заносится отметка о событии (включая время и идентификатор пользователя).

Любые действия ОС по обеспечению безопасности (внутренние действия) не дадут желаемого эффекта, если они не будут поддержаны внешними действиями: системой организационных мероприятий, выполняемых администраторами системы, пользователями и их руководителями. Рассмотрение внешних действий по обеспечению безопасности не входит в наши задачи, упомянем только, что введение в эту область можно найти в [12, 29].

## КОНТРОЛЬНЫЕ ВОПРОСЫ

1. В чем различие между избирательным и обязательным управлением доступом? Какой из этих подходов более надежен?
2. В чем преимущества объектно-ориентированных (объектно-базированных) ОС с точки зрения защиты?
3. Назовите те операции доступа, которые должны быть общими для любых типов объектов.
4. Назовите специфические операции доступа для объектов типа: файл, папка, программа.
5. В чем состоят процессы аутентификации и авторизации?
6. В каком виде может храниться в системе информация о правах доступа?
7. Что представляет собой "право" Exclude? Как оно применяется?
8. В чем состоит отличие полномочий от прав?
9. Опишите типовой сценарий процесса авторизации.
10. Чем объясняется необходимость введения классов пользователей?

## Глава 11. Интерфейс пользователя

До сих пор мы рассматривали взаимодействие с ОС, выполняемое из программы при помощи вызовов API – интерфейса прикладного программиста. Теперь рассмотрим взаимодействие вне программы – через команды, вводимые с клавиатуры терминала в интерактивных системах или поступающие во вводном потоке в пакетных системах. В первом случае, как правило, новая команда вводится после выполнения предыдущей и сама новая команда или ее параметры могут выбираться в зависимости от результатов этого выполнения. Во втором случае задается сразу целая последовательность команд и возможные отклонения от последовательного

их выполнения должны задаваться явным образом. Из таких различий в технологии взаимодействия пользователей с системой вытекают естественные различия в интерактивных и пакетных командных языках, но по мере расширения командных языков они имеют тенденцию к сближению: в интерактивные командные языки включаются возможности задания последовательностей команд, а в пакетные – более гибкие средства управления последовательностью выполнения.

## **11.1. Командный язык и командный процессор**

Команды представляют собой инструкции, сообщаемые ОС, что нужно делать. Команды могут восприниматься и выполняться либо модулями ядра ОС, либо отдельным процессом, в последнем случае такой процесс называется командным интерпретатором (оболочкой – shell). Набор допустимых команд ОС и правил их записи образует командный язык (CL – control language).

Большинство запросов пользователя к ОС состоят из двух компонент, показывающих: какую операцию следует выполнить и в каком окружении (environment) должно происходить выполнение операции. Могут различаться внутренние и внешние операции-команды. Выполнение внутренних операций производится самим командным интерпретатором, выполнение внешних требует вызова программ-утилит. Наличие в составе командного языка внутренних команд представляется совершенно необходимым для тех случаев, когда командный интерпретатор является отдельным процессом. Среди команд имеются такие, в которых выполняются системные вызовы, изменяющие состояние самого командного интерпретатора; если для них интерпретатор будет создавать новые процессы, то изменяться будет состояние нового процесса, а не интерпретатора. Примеры таких команд: `chdir` – изменение рабочего каталога для интерпретатора, `wait` – интерпретатор должен ждать



завершения порожденного им процесса и т.п. Программы-утилиты (их загрузочные модули) записаны в файлах на внешней памяти. При их вызове порождаются процессы, и утилиты выполняются в контексте этих процессов. Вызов и выполнение программ-утилит ничем не отличаются от вызова и выполнения приложений. Командный интерпретатор порождает процессы-потомки и выполняет в них заданные программы, используя для этого те же самые системные вызовы, которые мы рассмотрели в главе 4. Задание операции в командном языке, таким образом, может иметь вид имени программного файла, который должен быть выполнен.

Окружением или средой (далее эти слова используются как синонимы) называется то, что отличает одно выполнение программы от другого. Например, при выполнении программы-компилятора должны быть определены следующие параметры выполнения:

- какую программу следует выполнить;
- откуда программа должна взять исходные данные;
- куда программа должна поместить результат компиляции;
- где находятся библиотеки системы программирования;
- должен или не должен формироваться листинг;
- должны ли выдаваться предупреждения о возможных ошибках;
- и т.д.

Только первая из перечисленных составляющих задает саму программу, остальные составляют окружение. Окружение конкретного выполнения может формироваться одним из следующих способов или их комбинациями:

- командами установки локального окружения;
- параметрами программы;
- командами установки глобального окружения.

Окружение может быть локальным или глобальным. В первом случае параметры окружения устанавливаются только для данного конкретного

выполнения данной конкретной программы-процесса и теряются по окончании выполнения. Во втором случае параметры окружения сохраняются и действуют все время до их явной отмены или переустановки.

Команды для установки локальных параметров окружения применяются обычно в системах, работающих в пакетном режиме. В таких системах основным понятием является задание – единица работы с точки зрения пользователя. Выполнение задания состоит из одного или нескольких шагов. Каждый шаг задания включает в себя оператор JCL (job control language – язык управления заданиями) вызова программы и операторов установки локальной среды. Интерпретатор JCL сначала вводит все операторы, относящиеся к шагу задания, и лишь затем их выполняет. Тот же способ может применяться и в интерактивных системах – в этом случае команды установки параметров локальной среды должны предшествовать команде вызова.

Параметры программы также задают локальную среду выполнения. Они являются дополнениями к команде вызова, вводятся в той же командной строке, что и команда вызова, или являются параметрами оператора вызова в JCL. Формы задания параметров можно, по-видимому, свести к трем вариантам: позиционные, ключевые и флаговые. Возможны также и их комбинации. Позиционная форма передачи параметров программе похожа на общепринятую форму передачи параметров процедурам: параметры передаются в виде списка значений, интерпретация значения зависит от его места в списке. При передаче параметров в ключевой форме задается обозначение (имя) параметра и его значение; имя от значения отделяется специфицированным разделителем. Иногда другой специфицированный разделитель ставится перед именем как признак ключевой формы. Флаговая форма применяется для параметров, которые могут иметь только логические значения типа

"да"/"нет"; такой параметр обычно кодируется одним определенным для него символом, иногда перед ним ставится специфицированный разделитель. Для флагового параметра информативным является само его задание в команде вызова: если параметр не задан, применяется его значение по умолчанию, если задан – противоположное значение.

Ниже приводятся примеры передачи параметров:

- позиционная форма:

```
pgm1 data1.dat data2.dat list
pgm2 10,33,p12.txt
```

- ключевая форма:

```
pgm1
fil1=data1.dat,file2=data2.dat,list=yes
pgm2 /bsize:10 /ssize:33 /name:p12.txt
```

- флаговая форма:

```
pgm1 /1 /2 /p
pgm2 s,m,l
```

- комбинированная форма:

```
pgm1 data1.dat data2.dat #bsize=10 #ssize=33
#l #f
```

Наиболее универсальным типом, который позволяет передать программе любые параметры, является строка символов. Некоторые CL требуют сформировать такую строку при вызове явным образом в виде строковой константы с использованием соответствующих символов-ограничителей, в других эта задача выполняется командным интерпретатором. Программа сама интерпретирует свою строку параметров, выполняя ее лексический разбор. В реальных системах суммарный размер строки параметров обычно имеет некоторые разумные ограничения. Иногда командный интерпретатор выполняет предварительный лексический разбор строки параметров, выделяя из нее

"слова", разделенные пробелами, и передавая программе параметры в виде массива строк переменной размерности.

Каким образом параметры могут быть переданы программе? Можно назвать такие возможные механизмы передачи параметров:

- если командный интерпретатор выполняется как процесс, то он может послать процессу-программе параметры в виде сообщения;
- если командный интерпретатор является ядром ОС, то он копирует параметры в системную область памяти и программа может получить их при помощи специального системного вызова;
- если командный интерпретатор участвует в порождении процесса-программы (а это обычно так и бывает, независимо от того, является интерпретатор модулем ядра или процессом), то параметры могут быть записаны в адресное пространство нового процесса сразу при его создании.

В языках программирования, однако, механизм передачи параметров прозрачен для программиста, доступ к параметрам обеспечивает компилятор и в любом случае программа получает значения параметров уже в своем адресном пространстве. Так, в языке С для главной функции программы предопределен прототип::

```
int main(int argn, char *argv[]);
```

где `argn` – число строк-параметров, `argv` – указатель на массив строк-параметров.

Для установки глобального окружения применяются команды типа `set`. Операндами такой команды могут быть символьные строки, задающие в ключевой форме значения параметров окружения. Например:

```
set tempdir=d:\util\tmp  
set listmode=NO, BLKSIZE=1024
```

Переменные окружения могут быть системными или пользовательскими. Системные имеют зарезервированные символьные

имена и интерпретируются командным интерпретатором либо другими системными утилитами. Например, типичной является системная переменная окружения `path`, значение которой задает список каталогов для поиска программ командным интерпретатором. Пользовательские переменные создаются, изменяются и интерпретируются пользователями и приложениями. Чтобы окружение могло быть использовано, в системе должны быть средства доступа к нему. На уровне команд это должна быть команда типа `show`, выводящая на терминал имена и значения всех переменных глобального окружения, на уровне API – системный вызов `getEnvironment`, возвращающий адрес блока глобального окружения.

Для внутреннего представления глобального окружения в ОС возможны два варианта: либо хранить в системе единую таблицу со значениями всех переменных окружения, либо для каждого процесса создавать собственную копию такой таблицы. Чаще используется второй вариант, который обеспечивает, во-первых, лучшую защиту глобального окружения, а во-вторых, возможность варьировать окружения для разных процессов, используя глобальное окружение отчасти как локальное. Очень удобным является этот вариант для систем с иерархической структурой отношений между процессами: в этом случае глобальное окружение является частью наследства, передаваемого от предка к потомку. Системные вызовы, связанные с порождением новых процессов, должны обеспечивать возможность передавать потомку как точную копию глобального окружения родителя, так и оригинальное окружение, специально созданное родителем для потомка.

Внутреннее представление глобального окружения – всегда текстовое, представленное в ключевой форме, как в команде `set`. Это объясняется тем, что окружение интерпретируется прикладными процессами, а именно текстовый тип может быть интерпретирован наиболее гибким образом.

В ОС, применяющих "философию дешевых процессов" (см. главу 4), предполагается выполнение сложных действий как результата совместной (последовательной или параллельной) работы нескольких простых процессов. Поэтому командный язык должен включать в себя средства интеграции процессов. К числу таких средств относятся:

- командные списки;
- переадресация системного ввода-вывода;
- конвейеризация;
- параллельное выполнение.

Командные списки представляют собой простое перечисление в одной командной строке нескольких команд. Например, результат выполнения такой командной строки:

```
pgm1 param11 param12; pgm2; pgm3 param31
```

будет таким же, как при последовательным выполнением трех строк:

```
pgm1 param11 param12
```

```
pgm2
```

```
pgm3 param31
```

Командные списки представляют более удобную форму, но не открывают никаких новых возможностей.

Переадресация системного ввода-вывода использует возможности, описанные нами в разделе 7.6. Процессы вводят данные из файла системного ввода, с которым по умолчанию связана клавиатура, а выводят в файл системного вывода, по умолчанию – на экран терминала. Переадресация ввода дает возможность использовать в качестве входных данных программы данные, заранее записанные в файл, причем программа вводит и интерпретирует эти данные как введенные с клавиатуры. Переадресация вывода сохраняет данные, которые должны выводиться на экран, в файл. Примеры:

```
pgm1 < infile
```

```
pgm2 > outfile
```

Соединение командного списка с переадресацией ввода-вывода обеспечивает конвейеризацию. В примере:

```
pgm1 param11 param12 | pgm2 | pgm3 param31
```

выходные данные программы `pgm1` направляются не на экран, а сохраняются и затем используются, как входные для программы `pgm2`. Выходные данные последней в свою очередь используются как входные для `pgm3`.

В ОС с "философией дешевых процессов" допускается параллельное выполнение любого количества процессов. В обычном режиме командный интерпретатор готов к приему следующей команды только после окончания выполнения предыдущей. Специальная команда запуска программы или какой-либо признак в командной строке (например, символ-амперсанд в конце ее) может применяться в качестве указания командному процессору вводить и выполнять следующую команду, не дожидаясь окончания выполнения предыдущей. Так, последовательность командных строк:

```
pgm1 param11 param12 &  
pgm2 &  
pgm3 param31
```

может привести к параллельному выполнению трех процессов (если программа `pgm1` не закончится прежде, чем будет введена третья строка).

Ниже приведен программный текст, представляющий в значительно упрощенном виде макет командного интерпретатора `shell` ОС `Unix` (синтаксис и семантика системных вызовов в основном тоже соответствуют этой ОС). Из этого примера можно судить о том, как `shell` реализует некоторые из описанных выше возможностей.

```

. . .
1  int fd, fds [2];
2  int status, retid, numchars=256;
3  char buffer [numchars], outfile[80];
4  . . .
5  while( read(stdin,buffer,numchars) ) {
6      <синтаксический разбор командной строки>
7      if (<не внутренняя команда>) {
8          if(<командная строка содержит &>) amp=1;
9          else amp=0;
10         if( fork() == 0 ) {
11             if( < переадресация вывода > ) {
12                 fd = create(outfile,<режим>);
13                 close(stdout);
14                 dup(fd);
15                 close(fd);
16             }
17             if(<переадресация ввода>) {
18                 <подобным же образом>
19             }
20             if(<конвейер>) {
21                 pipe (fds);
22                 if ( fork() == 0 ) {
23                     close(stdin);
24                     dup(fds[0]);
25                     close(fds[0]);
26                     close(fds[1]);
27                     exec(pgm2, <параметры> );
28                 }

```



```

29         else {
30             close(stdout);
31             dup(fds[1]);
32             close(ds[1]);
33             close(fds[0]);
34         }
35     }
36     exec(pgm1, <параметры>);
37 }
38 if ( amp == 0 ) retid = wait(&status);
39 }
40 }
    . . .

```

Наш макет рассчитан на интерпретацию командной строки, содержащей либо вызов одной программы (pgm1), либо конвейер из двух программ (pgm1 | pgm2). Макет также обрабатывает переадресацию ввода-вывода и параллельное выполнение.

Тело shell представляет собой цикл (5 - 39), в каждой итерации которого из файла стандартного ввода вводится (5) строка символов – командная строка. Далее shell выполняет разбор командной строки, выделяя и распознавая собственно команду (или команды), параметры и т.д. Если (7) распознанная команда не является внутренней командой shell (обработку внутренних команд мы не рассматриваем), а требует выполнения какой-то программы – безразлично, системной утилиты или приложения, – то shell проверяет наличие в командной строке признака параллельности и соответственно устанавливает значение флага amp (8, 9). Затем shell порождает новый процесс (10) и весь следующий блок (11 - 37) выполняется только в процессе-потомке. Если shell распознал в команде

переадресацию системного вывода (11), то выполняются соответствующие действия (11 - 16). Они состоят в том, что `shell` создает файл, в который будет перенаправлен поток вывода и получает его манипулятор (12). Затем закрывается файл системного вывода (13). Системный вызов `dup` (14) дублирует манипулятор `fd` в первый свободный манипулятор таблицы файлов процесса. Поскольку только что освободился файл системного вывода, манипулятор которого – 1, дублирование произойдет именно в него. Теперь два элемента таблицы файлов процесса – элемент с номером 1 и элемент с номером `fd` – адресуют один и тот же файловый дескриптор – дескриптор только что созданного файла. Но элемент `fd` сразу же освобождается (15), и теперь только манипулятор 1, который интерпретируется в программе как манипулятор системного вывода, адресует новый файл. Мы предлагаем читателю самостоятельно запрограммировать действия `shell` при переадресации ввода (17 - 19). (Для справки: манипулятор системного ввода – 0.)

Если в командной строке задан конвейер (20), то процесс-потомок прежде всего создает канал (21). Параметром системного вызова `pipe` является массив из двух элементов, в который этот вызов помещает манипуляторы канала: `fds[0]` – для чтения и `fds[1]` – для записи. Затем процесс-потомок порождает еще один процесс (22). Следующий блок (23 - 27) выполняется только во втором процессе-потомке. Этот потомок переадресует свой стандартный ввод на манипулятор чтения канала (23 - 25). Манипулятор записи канала освобождается за ненадобностью (26). Затем второй потомок загружается программой, являющейся вторым компонентом конвейера (27). Следующий блок (28 - 34) выполняется только в первом потомке: он переадресует свой стандартный вывод на манипулятор записи канала и освобождает манипулятор чтения канала.

Системный вызов `exes` (36) выполняется в первом потомке (он является единственным, если конвейер не задан), процесс загружается программой, являющейся первым компонентом конвейера или единственным компонентом командной строки. Обратите внимание на то, что из функции `exes` нет возврата. При ее вызове выполняется новая программа, с завершением которой завершается и процесс. Процесс-родитель, который все время сохраняет контекст интерпретатора `shell`, после запуска потомка проверяет (38) флаг параллельного выполнения `amp`. Если этот флаг не установлен, то родитель выполняет вызов `wait`, блокирующий `shell` до завершения потомка. (Этот вызов возвращает идентификатор закончившегося процесса, а в параметре – код завершения, но в нашем макете эти результаты не обрабатываются). Если флаг параллельности установлен, то `shell` начинает следующую свою итерацию, не дожидаясь завершения потомка.

При разработке командного интерпретатора необходимо следовать "принципу пользователя", который может быть сформулирован так: те операции, которые часто выполняются, должны легко вызываться. Для более полного воплощения этого принципа в командный интерпретатор могут быть встроены сервисные возможности. Можно привести такие примеры этих возможностей:

- установки по умолчанию – относятся, прежде всего, к параметрам глобального окружения, обычно эти установки записываются в отдельный файл – командный файл (см. следующий раздел) начальной загрузки типа `AUTOEXES.BAT` или в пользовательский профиль;
- встроенные сокращенные формы команд; в некоторых интерпретаторах применяется метод автоматического завершения: пользователь, введя первые символы команды, нажимает

определенную клавишу, и интерпретатор выводит в командную строку остаток команды;

- интеграция команд – введение в состав CL сложных команд, эквивалентных цепочке простых команд, иногда пользователь имеет возможность создавать в командном интерпретаторе собственные интегрированные команды, но чаще такая возможность реализуется через командные файлы;
- сохранение истории – введенные командные строки запоминаются в стеке, и по определенной клавише содержимое стека выбирается в командную строку.

В тех ОС, где командный интерпретатор является процессом, имеется возможность запуска вторичного интерпретатора. Эта возможность является очень удобной, если при работе в среде какого-либо приложения возникает необходимость выполнить команды ОС, но завершать приложение из-за этого нежелательно. Приложение является процессом-потомком командного интерпретатора. Оно порождает новый процесс (потомок приложения), в котором запускается еще одна копия интерпретатора, как показано на рисунке 11.1. (Новый процесс-интерпретатор может разделять сегмент кодов с первичным интерпретатором). Теперь весь ввод в командную строку обрабатывается этим вторичным интерпретатором. Вторичный интерпретатор может запустить другое приложение и т.д. Вторичный интерпретатор запускается в синхронном режиме и при его завершении (команда `exit`) продолжает работать запускаявшее его приложение.

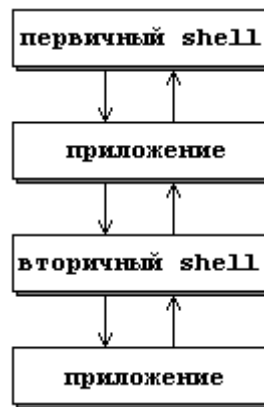


Рисунок 11.1 Запуск вторичного командного интерпретатора

## 11.2. Командные файлы и язык процедур

Принцип пользователя диктует необходимость короткого обращения к часто выполняемым последовательностям команд. Простым и эффективным решением этой задачи является запись такой последовательности в текстовый файл и обращение к ней в дальнейшем по имени файла. Мы будем называть такие файлы командными. В интерактивных системах их иногда также называют пакетными файлами (batch file), а в пакетных – файлами процедур JCL. Командный интерпретатор должен при вводе команды-обращения к такому файлу распознать тип файла (иногда признак обращения к командному файлу требуется указать в самой команде) и далее считывать и интерпретировать команды из файла.

В простейшем случае командный файл содержит неизменяемую последовательность команд и является просто аббревиатурой этой последовательности. Но сервис, обеспечиваемый инвариантными последовательностями, явно недостаточен. Например, для программиста типичным является "сценарий" работы, состоящий из многократного повторения таких шагов:

- редактирование исходного модуля;

- компиляция;
- компоновка;
- выполнение.

Каждый шаг связан с вызовом новой программы, следовательно, с новой командой. Очевидно, что при записи такого сценария в командный файл мы должны обеспечить для него хотя бы один параметр – имя исходного модуля. Параметры являются совершенно необходимым свойством для командных файлов. Параметры нетрудно обрабатывать простой текстовой подстановкой.

Очевидно также, что в том же сценарии не имеет смысла компоновать, а тем более выполнять программу, если при компиляции в ней обнаружены ошибки. Отсюда – необходимость управлять последовательностью выполнения команд в командном сценарии. Простейшим вариантом такого управления является включение в команду условия ее выполнения, более сложный и гибкий вариант – условный переход на ту или иную команду. В условии выполнения или перехода должен анализироваться код завершения одной или нескольких предыдущих команд. Общепринятым является успешное завершение программ и команд с кодом 0. Код завершения может формироваться программой, выполняющей команду, как параметр системного вызова `exit` и восприниматься процессом-родителем (командным интерпретатором) в системном вызове `wait`.

Параметры и условия являются тем набором средств, который обеспечивает минимальный необходимый сервис. Некоторые ОС на этом и останавливаются. Другие же идут дальше. Пионером в этой области (как и во многих других) является ОС Unix. Это неудивительно, так как Unix воплощает "философию дешевых процессов", а для того, чтобы скомпоновать из "дешевых" процессов сложные действия, нужны развитые средства интеграции. Просматривая публикации по ОС Unix в хронологическом порядке, можно наблюдать, как языковые средства shell наращивались новыми алгоритмическими возможностями, все более

приближая его к процедурному языку программирования. В итоге shell обладает полным набором средств процедурного программирования (операций и операторов), включая манипулирование с переменными shell-программы, условный оператор, оператор множественного выбора, операторы циклов, оператор обработки исключительных ситуаций, а также возможность создания и вызова функций.

По иному пути пошла фирма IBM, в середине 80-х годов представившая в составе ОС CMS (гостевой ОС в среде ОС виртуальных машин VM/370) реструктурированный расширенный язык процедур – REXX (Restructured EXtended eXecutor language) [46]. Разработчики этого языка пошли не по пути наращивания командного языка алгоритмическими возможностями, а по пути включения в мощный алгоритмический язык (за основу был взят язык PL/1) средств выполнения команд. Подход оказался настолько продуктивным, что за прошедшее с тех пор время REXX практически не претерпел изменений и сейчас входит в базовый комплект поставки не только CMS VM/ESA, но всех ОС фирмы. Наряду со средствами процедурного программирования, "унаследованными" от PL/1, в REXX включен в качестве базовых операций языка ряд операций расширенной обработки строк и большое количество встроенных функций, также прежде всего связанных с обработкой строк, которыми компенсируется отсутствие того богатого набора утилит, который имеется в Unix. Кроме того, в REXX имеются возможности (эти возможности системно-зависимые) работы с текстовыми файлами и обмена данными через очереди или перенаправление ввода-вывода не только в файлы, но и в буферы, подобные программным каналам, но со структурой дека (очереди с двумя концами). Вообще область применения REXX шире, чем только применение его в качестве командного интерпретатора ОС. Целый ряд продуктов системного и промежуточного программного обеспечения IBM использует REXX как

интерпретатор своих команд. Оператор ADDRESS задает имя программы-среды, в которую передается команда.

Оба типа развитых командных языков наряду с одинаковыми алгоритмическими возможностями обладают также еще одним принципиально важным общим свойством – они являются языками интерпретирующего типа. Командный файл REXX или shell не требует компиляции. Это означает, что полный анализ такого файла не производится (или производится только в первом приближении), и интерпретатор выполняет его команда за командой, "не заглядывая" вперед. Переменные командного файла имеют единственный тип – "строка символов", и основные манипуляции над ними представляют собой строковые операции. При выполнении арифметики строковые данные прозрачно преобразуются в числовые, а результат операции вновь преобразуется в строку. Результаты выполнения программ, вызываемых в командном файле. При выполнении каждого очередного оператора командного файла производится подстановка вместо переменных shell-или REXX-программы их значений. В обоих языках предусмотрены средства "экранирования", защищающие строковые литералы от интерпретации их как переменных. Строка, полученная после выполнения подстановки, интерпретируется как оператор командного языка или – если это невозможно – как команда ОС (или другой целевой среды). В REXX имеется возможность даже сформировать символьную строку в переменной REXX-программы, а затем выполнить ее как оператор языка.

Таким образом, командные языки ОС обладают всеми возможностями языков программирования, и, в принципе, пригодны для создания не только командных процедур, но и некоторых программ обработки данных. Выполнение командных файлов в режиме интерпретации, конечно, делает такие программы менее эффективными, чем программы, написанные на языках компилирующего типа, но создает



в них некоторые дополнительные возможности и вырабатывает некоторый особый стиль программирования.

### **11.3. Проблема идентификации адресата**

При параллельном выполнении нескольких процессов возникает проблема взаимодействия пользователя с ними: при появлении на экране сообщения, как определит пользователь, какой из процессов это сообщение выдал; при вводе сообщения пользователем, как ОС определит, какому процессу сообщение адресовано? Решение этой проблемы зависит, прежде всего, от степени интерактивности процессов (насколько часто они обмениваются сообщениями с пользователем). Если процессы не очень интерактивны, то им можно разрешить квазиодновременный вывод на терминал: их сообщения могут чередоваться, но каждое сообщение должно быть неделимым. В этом случае выводимое сообщение должно нести в себе и идентификацию процесса, его выдавшего. Дейкстра [11] подробно рассмотрел реализацию диалогового взаимодействия такого рода с помощью семафоров. Им показано, что для этого случая ответы пользователя должны либо также содержать идентификатор процесса-адресата, либо пара действий вопрос–ответ должна быть одной транзакцией. В последнем случае, однако, задержка ответа надолго блокирует общий канал обмена между процессами и пользователем, поэтому у пользователя также должна быть возможность откладывания ответа с последующим возвращением к диалогу с этим же процессом.

Подход, предполагающий явное адресование сообщений, становится излишне громоздким, если пользователь работает с существенно интерактивными процессами. Общая идея обеспечения взаимодействия для такого случая состоит в том, что в каждый момент времени только одному процессу (по выбору пользователя) разрешается быть

интерактивным (foreground – на переднем плане), т.е. иметь доступ к терминалу. Остальные процессы (background – фоновые) выполняются без доступа к терминалу. В распоряжении пользователя имеются средства изменения статуса процесса – перемещения его с заднего плана на передний (текущий процесс переднего плана при этом отодвигается на задний план).

Что делать с фоновым процессом, если у него возникла необходимость выдать сообщение на терминал? Одним из решений может быть блокировка такого процесса до его выдвижения. Более эффективным является "скрытый" вывод, при котором обеспечивается для выполнения процессов несколько параллельных сеансов. В каждом сеансе могут выполняться один или несколько процессов. С каждым сеансом связан собственный буфер логического терминала, включающий в себя логический видеобуфер, логический буфер клавиатуры и логический буфер мыши. Весь обмен процесс ведет с логическим видеобуфером терминала. В каждый момент логический буфер только одного (так называемого активного) сеанса связан с физическими буферами терминальных устройств. Таким образом, фоновый сеанс может вести обмен с терминалом (в основном – вывод) без отображения на терминал. Когда сеанс становится активным, его логический буфер связывается с физическими устройствами, и на экране видеотерминала отображается содержимое его логического видеобуфера. В составе API есть также средства аварийной активизации сеанса: если процессу, работающему в фоновом сеансе, необходимо выдать срочное сообщение, то он может воспользоваться так называемым "иллюминатором". На время существования иллюминатора ОС выделяет временный активный видеобуфер, и все средства обмена с консолью работают с этим буфером. Это очень сильное средство, используемое только в исключительных случаях, так как на время существования иллюминатора блокируются

запросы активного сеанса, а переключать сеансы в это время не может даже оператор.

## **11.4. WIMP-интерфейс**

Интерфейс командной строки (но не командные файлы!) на сегодняшний день уже можно считать отходящим в прошлое, хотя прогнозировать его окончательный уход мы не беремся. Программируемые видеотерминалы дают возможность выводить информацию в любую позицию экрана и, следовательно, использовать все пространство экрана для организации взаимодействия между ОС и пользователем. Современные интерфейсы как приложений, так и ОС, можно охарактеризовать как полноэкранные, графические, объектно-ориентированные.

Полноэкранный интерфейс строится на основе принципа согласованности, который состоит в том, что у пользователя формируется система ожидания одинаковых реакций на одинаковые действия. Основными компонентами интерфейса являются панели, диалог и окна.

Панель – это информация, определенным образом сгруппированная и расположенная на экране. Основные типы панелей:

- меню – содержит один или более списков объектов, представляющих группы действий, доступных пользователю;
- панель ввода – отображает поля, в которые пользователь вводит информацию;
- информационная панель – отображает защищенную информацию (данные, сообщения, справки);

- списковая панель – содержит один или более списков объектов, из которых пользователь выбирает один или несколько и запрашивает одно или несколько действий над ними;
- панель-канва – свободное пространство, на котором можно размещать другие объекты интерфейса, такие как:
  - кнопки различных видов и различной функциональности;
  - элементы выбора;
  - статические текстовые поля;
  - протяжки и т.д.

Диалог – это последовательность запросов между пользователем и компьютером: запрос пользователем действия, реакция и запрос компьютера, ответное действие пользователя и т.д. Диалог включает в себя запросы и навигацию – переходы из одной панели в другую. Информация, вводимая пользователем в ходе диалога, может удерживаться только на уровне данной панели или сохраняться.

Панели могут располагаться в отдельных ограниченных частях экрана, называемых окнами. Окна могут быть трех типов:

- первичное – окно, в котором начинается диалог;
- вторичные – вызываемые из первичного окна, в которых диалог ведется параллельно диалогу в первичном окне;
- всплывающие (pop-up) – расширяющие диалог в первичном или вторичном окне; перед тем, как продолжить диалог с окном пользователь должен завершить работу со связанным с ним всплывающим окном.

Общие принципы панельного интерфейса в основном не зависят от типа применяемых терминалов. Однако сочетание графических видеоадаптеров с высокой разрешающей способностью с общим увеличением вычислительной мощности (быстродействие и объем памяти)

персональных вычислительных систем позволяет существенно изменить общий облик экрана. Можно определить следующие основные направления этих изменений: многооконность, объемность, иконика. Такие интерфейсы получили название WIMP (Windows, Icons, Menus, Pointer). Первое воплощение идеи WIMP нашли в разработках фирмы Xerox, а первая их коммерчески успешная реализация состоялась в компьютерах Apple Macintosh в 1985 году. Позднее идеи WIMP были приняты в Microsoft Windows, а сейчас они воплощены практически во всех операционных системах.

Интерфейс WIMP обладает концептуальной целостностью, достигаемой принятием знакомой идеальной модели – метафоры рабочего стола – ровной поверхности, на которой расположены объекты и папки, и ее тщательного последовательного развития для использования воплощения в компьютерной графике. Главное изменение в облике интерфейса – иконика – представление объектов в виде миниатюрных графических изображений – пиктограмм. Помимо чисто внешних изменений иконика породила возможность манипулировать объектами через манипулирование их изображениями. Значки и вложенные папки и мусорная корзина являются точными аналогами документов на столе. Вырезание, копирование и вставка точно имитируют операции, которые обычно осуществляются с документами на столе. Транспортировка непосредственно вытекает из метафоры рабочего стола; выбор значков или окон с помощью курсора является прямой аналогией захвата предметов рукой. Из метафоры рабочего стола непосредственно следует решение о перекрытии окон вместо расположения их одно рядом с другим. Представление активного окна как документа, "лежащего сверху", интуитивно понятным образом решает проблему идентификации адресата. Возможность менять размер и форму окон не имеет прямой аналогии с бумажными документами, но является последовательным расширением,

дающим пользователю новые возможности, обеспечиваемые компьютерной графикой

В некоторых случаях интерфейс WIMP отходит от метафоры рабочего стола. Основные отличия: меню и работа одной рукой. Меню представляет собой не совершение действия, а выдачу кому-то (системе/приложению) команды на осуществление действия, причем, команда эта не формулируется языковыми средствами, а выбирается из списка.

Даже на чисто текстовых видеотерминалах имелась возможность вывода на экран нескольких окон одновременно, но для графического режима эта возможность значительно расширилась. Поскольку появление графического интерфейса в Mac OS и Windows совпало с введением многозадачности (сначала – без вытеснения), естественным образом возникло решение о выделении каждому из работающих приложений собственного окна (первичной панели). При одновременной работе нескольких приложений их окна могут перекрывать друг друга – частично или полностью, но на переднем плане всегда находится окно активного в данный момент приложения. Поскольку обилие окон может затруднить ориентацию пользователя, вводится возможность минимизации или сокрытия окон: окна неактивных приложений могут уменьшаться в размерах или вообще не выводиться на экран. Для предотвращения "потерь" скрытых окон у пользователя должна быть возможность в любой момент просмотреть список работающих приложений и восстановить нормальную визуализацию выбранных окон.

Высокая разрешающая способность графических дисплеев позволяет также имитировать объемные панели, создавая на плоском экране иллюзию светотеней. На "объемной" панели применяются графические элементы – органы управления, такие как: кнопки, линейка протяжки и т.д. Общепринятым является представление полей ввода в

"утопленном" виде, а органов управления – в "приподнятом". К настоящему времени облик объемного интерфейса в современных ОС сформировался почти окончательно и включает в себя единый "источник света" и однотипное расположение органов управления на всех панелях.

Объектно-ориентированные свойства интерфейса совершенно необязательно связаны с объектно-ориентированной структурой ОС. Так, например, OS/400 является объектно-ориентированной системой с объектно-ориентированным интерфейсом, Windows NT v.3.51 была объектно-ориентированной ОС без объектно-ориентированного интерфейса, OS/2 и Windows 9x – не объектно-ориентированные ОС с объектно-ориентированными интерфейсами. Объектно-ориентированный интерфейс обычно связывают с графическим интерфейсом, но это необязательно. Так, в той же OS/400 предусмотрены две модели интерфейса: текстовая и графическая, обе в полной мере объектно-ориентированные.

В противовес обычному интерфейсу, который представляет пользователю практически единственный тип объекта – файл, единицу хранения информации в ОС, объектно-ориентированный интерфейс представляет объекты различных типов. Файлы могут быть разными типами объектов – в зависимости от типа информации, в них хранящейся, и способов ее обработки. Кроме того, объектами могут быть устройства, сетевые ресурсы и т.д. В объектно-ориентированном программировании под объектом понимается абстрактный тип данных, включающий в себя как сами данные, так и процедуры их обработки. Аналогично объекты понимаются и в объектно-ориентированном интерфейсе. Объект обязательно обладает некоторым набором свойств, и значения этих свойств доступно пользователю. Среди свойств, присущих объекту, имеется и указание на способ его обработки – в том числе и на приложение, обрабатывающее данные этого типа. Выполнение некоторых

действий над объектом включает в себя автоматический запуск приложений, которые эти действия выполняют.

Концептуально важным объектом интерфейса является папка (folder). Папка – это контейнерный объект, содержащий в себе другие объекты и папки. Уместность папки в метафоре рабочего стола очевидна. Существенно то, что папка дает возможность пользователю создавать собственную структуру хранения объектов, альтернативную структуре хранения объектов в ОС (в файловой системе). Важным свойством, обеспечивающим эту возможность, является создание указателей на объекты. Если папка является физическим аналогом каталога файловой системы, то в нее может быть помещен указатель на объект, физически расположенный в другой папке-каталоге файловой системы (аналог косвенных файлов). Ссылка на объект с точки зрения пользователя выглядит так же, как оригинал объекта (хотя может иметь какие-то отличительные признаки ссылки), выполнение операции открытия над ссылкой приводит к открытию объекта-оригинала, но операции перемещения, удаления, переименования и т.п. выполняются не над объектом, а только над ссылкой. Возникает, однако, проблема согласования интерфейсной структуры хранения объектов с логической структурой файловой системы. Например, требуется, чтобы при перемещении объекта-оригинала в файловой системе все ссылки на него перенаправлялись на новое его место. Не все интерфейсы ОС успешно справляются с этой задачей.

Важным аспектом объектной ориентации является настройка интерфейса для конкретного пользователя. Обычно, если интерфейс рассматривается с точки зрения приложений, отмечается полезность создания нескольких форм интерфейса, ориентированных на пользователя разной квалификации – новичка, опытного, профессионала. Хотя та же задача может ставиться и перед интерфейсом ОС, более важной, на наш



взгляд является интеграция интерфейса с системой безопасности ОС. Интерфейс должен показывать пользователю только те объекты и предоставлять ему только те команды, к которым данный пользователь имеет доступ. Такое возможно в тех ОС, где система безопасности тесно связана с объектно-ориентированными свойствами ОС. Настройки интерфейса могут являться частью профиля пользователя.

Каково место интерфейса WIMP в ОС? Можно назвать три подхода к выбору такого места.

Графический интерфейс может встраиваться в саму ОС и быть ее неотъемлемой частью. Такой подход применяется во всех продуктах семейства Windows и в ОС компьютеров Apple (в последних WIMP даже встроен в ПЗУ компьютера). Это дает возможность тесно интегрировать интерфейс с ОС и повысить производительность интерфейсных модулей, выполняя часть из них в режиме ядра. Однако такой подход в то же время является неэкономным, так как интерфейс WIMP расходует много ресурсов и до некоторой степени опасным, так как модули WIMP могут явиться дополнительным источником ошибок в системе.

Графический интерфейс может представлять собой отдельное приложение, поставляемое в составе операционной системы и, возможно, достаточно тесно интегрированное с ней. Пример такого приложения – Workplace Shell OS/2. Такое приложение не допускается в режим ядра, но может использовать API более низкого уровня, чем обычно используемый в приложениях. Такое приложение WIMP не является обязательным компонентом ОС, система может работать и без него, в режиме командной строки или загрузить другое приложение WIMP.

Наконец, графический интерфейс может представлять собой приложение, никак не связанное с ОС, выполняющееся в тех же условиях, что и другие приложения, и выполняющее действия, задаваемые пользователем, используя обычный API ОС. В этом случае ОС не связана

жестко с одним модулем WIMP, и графический интерфейс может выбираться по желанию пользователя. Примером такой ОС с большим выбором интерфейсов является Linux.

Нам представляется, что второй и третий подходы, дающие пользователю возможность выбора, являются предпочтительными.

Принцип согласованности интерфейса диктует необходимость для всех разработчиков приложений обеспечивать однотипный интерфейс в разных приложениях. Естественным решением является возможность для разработчиков приложений использовать те же модули и объекты, которые используются для построения WIMP-интерфейса ОС. В случае встроенного в ОС графического интерфейса системные объекты, обеспечивающие интерфейсные функции, делаются доступными для пользователей через соответствующий API (Windows). В случае интерфейса, представляющего собой интегрированное с ОС приложение, библиотека интерфейсных функций и объектов поставляется в составе ОС (Object Class Library в OS/2). Основой независимых графических интерфейсов являются независимые инструментальные средства, на основе которых может быть построен тот или иной WIMP-интерфейс.

Одной из наиболее успешных систем для построения таких интерфейсов является X Window, созданная в Массачусетском Технологическом Институте. Архитектура X Window построена по принципу клиент/сервер. Взаимодействие X-клиента и X-сервера происходит в рамках прикладного уровня – X-протокола. Для X Window безразличен транспортный уровень передачи, таким образом, X-клиент и X-сервер могут располагаться на разных компьютерах, в разных аппаратных и операционных средах, то есть программа может осуществлять ввод-вывод графической информации на экран другого компьютера. Все различия в аппаратных и программных архитектурах X-клиента и X-сервера сглаживаются стандартом X-протокола. На базе

инструментальных средств X Window было создано несколько сред WIMP-интерфейсов, наиболее популярный из которых, по-видимому, Motif, являющийся стандартом Open Software Foundation.

По-видимому, WIMP-интерфейс не является окончательным решением, по мере развития аппаратных и программных средств обработки данных он, вполне вероятно, будет вытеснен новыми подходами и новыми средствами. Скорее всего на смену метафоре рабочего стола придет какая-то новая модель, обладающая собственной концептуальной целостностью. Сегодня представляется вероятным, что новая модель будет найдена где-то в области средств мультимедиа, которые сейчас развиваются весьма стремительно. Возможно, это будет голосовой ввод, возможно, что-нибудь другое. Однако, это будет достоянием уже следующего поколения. Пока, во всяком случае, мы не можем назвать какой-либо новой модели, обладающей такой же понятийной целостностью, как концепция рабочего стола.

## **КОНТРОЛЬНЫЕ ВОПРОСЫ**

1. В каких случаях целесообразно задавать условия выполнения программы в виде параметров глобального окружения, в виде параметров локального окружения, в виде параметров вызова?
2. Почему параметры вызова программы всегда имеют тип строки символов?
3. Что такое конвейеризация и чем она обеспечивается? Приведите примеры задач, легко решаемых при помощи конвейеризации.
4. Какие необязательные свойства командного интерпретатора могут сделать его более удобным в использовании?

5. Как при параллельном выполнении нескольких процессов в режиме командной строки определить процесс, выдавший сообщение на экран? Как адресовать ответ оператора определенному процессу?
6. Синтаксис языка C-shell создавался на основе языка программирования C, синтаксис языка REXX – на основе PL/1. В чем принципиальное отличие языков C-shell и REXX от их прототипов?
7. Назовите основные элементы интерфейса WIMP и их функции.
8. Какие свойства интерфейса WIMP полностью следуют метафоре рабочего стола? Какие ее расширяют?
9. Приведите соображения "за" и "против" встраивания графического интерфейса непосредственно в ОС.
10. На примере X Window объясните концепцию аппаратно-независимого интерфейса.

## **Заключение**

Принципы управления вычислительными ресурсами, как они изложены в данном пособии, сложились в конце 60-х - начале 70-х годов. Лишь очень ограниченный ряд общих решений (нити, файлы, отображаемые в память, объектно-ориентированные системы, WIMP-интерфейс и немногие другие) датируется более поздним периодом, хотя некоторые из этих решений локально (в отдельных системах) существовали и ранее. Прогресс в развитии ОС и информационных технологий вообще идет не столько по пути изобретения новых подходов, сколько по пути адаптации уже известных подходов к новым условиям. На разных этапах этого развития те или иные решения (концепции, дисциплины, алгоритмы) становятся более популярными, возможно, господствующими, но при изменении условий оказываются

востребованными и другие решения, которые казались отжившими. У авторов складывается впечатление, что в настоящее время информационные технологии завершают виток своего развития, и сегодняшние решения гораздо более базируются на концепциях начального этапа, чем решения, например, 10-летней давности.

Это позволяет сделать вывод о том, что вопросы, рассмотренные нами в данной книге относятся к багажу фундаментальных знаний специалиста в области информационных технологий, и значение этих знаний не зависит от рыночной конъюнктуры.

Во второй части учебного пособия будут рассмотрены реализации общих принципов управления ресурсами в конкретных современных ОС.

## **Список литературы**

1. Авен О.И., Коган Я.А. Управление вычислительным процессом в ЭВМ. – М.: Энергия, 1978.
2. Артамонов Г.Т., Брехов О.М. Аналитические вероятностные модели функционирования ЭВМ. – М.: Энергия, 1978.
3. Бек Л. Введение в системное программирование. – М.: Мир, 1986.
4. Брукс П.Ф. Мифический человеко-месяц или как создаются программные системы. – СПб.: Символ-Плюс, 2001.
5. Вебер Д. Технология Java™ в подлиннике. – СПб.: БХВ-Петербург, 2000.
6. Вегнер П. Программирование на языке АДА. – М.: Мир, 1983.
7. Гордеев А.В., Молчанов А.Ю. Системное программное обеспечение. – СПб: Питер, 2001.
8. ГОСТ 19781-90. Обеспечение систем обработки информации программное. Термины и определения. – М.: Изд-во стандартов, 1990.

9. Готье Р. Руководство по операционной системе Unix. – М.:Финансы и статистика, 1985.
10. Дал У., Дейкстра Э., Хоор К. Структурное программирование. – М.:Мир, 1972.
11. Дейкстра Э. Взаимодействие последовательных процессов. // "Языки программирования"; под ред. Ф.Женюи. – М.:Мир, 1972.
12. Дейтел Г. Введение в операционные системы – М.:Мир, 1986 – Т.1, 2.
13. Донован Дж. Системное программирование. М.:Мир, 1975.
14. Зелковиц М., Шоу А., Гэннон Дж. Принципы разработки программного обеспечения. – М.:Мир, 1982.
15. Кейслер С. Проектирование операционных систем для малых ЭВМ. – М.:Мир, 1986.
16. Клейнрок Л. Вычислительные системы с очередями. – М.:Мир, 1979.
17. Колин А. Введение в операционные системы. – М.,Мир, 1975.
18. Коэн Л.Дж. Анализ и разработка операционных систем. – М.:Мир, 1975.
19. Краковяк С. Основы организации и функционирования ОС ЭВМ. – М.:Мир, 1988.
20. Кузьминский М. Z-архитектура. // "Открытые системы", 2001, №10 (<http://www.osp.ru/os/2001/10/010.htm>).
21. Операционная система IBM/360. Супервизор и управление данными. – М.:Сов.радио, 1973.
22. Олифиер В.Г, Олифиер Н.А. Сетевые операционные системы. – СПб.: Питер, 2000.
23. Петцольд Ч. Windows 95 – вызов программистам // "PC Magazine. Russian Edition", 1995, N8.

24. Пратт Т. Языки программирования. Разработка и реализация. – М.:Мир, 1979.
25. Ресурсы Windows NT. – С-Пб.: BVH–Санкт-Петербург, 1995.
26. Соловьев Г.Н., Никитин В.Д. Операционные системы ЭВМ. – М.:Высшая школа, 1989.
27. Солтис Ф. Основы AS/400. – М.: Изд.отд. "Русская редакция" ТОО "Channel Trading Ltd." – 1998.
28. Тимонин В.М. СВМ ЕС. Основы функционирования и средства обеспечения пользователя. – М.:Изд-во МАИ, 1990.
29. Цикритзис Д., Бернстайн Ф. Операционные системы – М.:Мир, 1977.
30. Шоу А. Логическое проектирование операционных систем – М.:Мир, 1981.
31. Якушевский И. Мэйнфреймы – мифы и реальность. – "Hard'n'Soft", 1995, N11.
32. i486 процессор. Кн.1, 2. – М.:И.В.К.–СОФТ, 1993.
33. Bach M.J. Design of the Unix Operating System. – Prentice-Hall Inc, Englewood, 1986. (Русский перевод есть, например, в <http://lib.ru/BACH/>).
34. Bolthouse D. Exploring IBM Client/Server Computing. – Maximum Press, NJ, 1997.
35. Enterprise System/9000. 9221 Processor. Processors Characteristics. – IBM, Publ.No SA33-1609-03, 1994.
36. Finkel R.A. An Operating Systems Vade Mecum. – Prentice-Hall Inc, Englewood, 1988.
37. Gosling J., Joy B., Steele G., Bracha G. The Java Language Specification. Second Edition. – Sun Microsystems Inc., 2000.
38. Hoskins J. IBM System/390. A Business Perspective. – Maximum Press, NJ, 1997.

39. iSeries Library – <http://www-1.ibm.com/servers/eserver/series/library/>
40. Joung J. Exploring IBM's New Age Mainframes. – Maximum Press, NJ, 1995.
41. Madnick S., Donovan J. Operating Systems. – McGraw-Hill, 1986.  
(Есть русский перевод более раннего издания: Мэдник С., Донован Дж. Операционные системы. – М.:Мир, 1975)
42. QNX System Architecture. – [http://www.qnx.com/literature/qnx\\_sysarch/](http://www.qnx.com/literature/qnx_sysarch/)
44. Solomon D.A., Russinovich M. Inside Microsoft Windows 2000. – Microsoft Press, 2000
44. Vajapeyam S. Early 21<sup>st</sup> Centiry Processors. – "Computer", April 2001. – pp. 47 – 50.
45. z/Architecture. Principles of Operation, IBM SA22-7832-00
46. z/VM Internet Library – <http://www.vm.ibm.com/library/>