технологии программирования к оглавлению к высокоуровн. языкам - 3GL к визуальным средам - 4GL

Системный подход в технологии программирования

Операционная платформа

Изучение механизма и структуры операционных систем необходимо по следующим причинам:

- основные идеи, концепции и алгоритмы, лежащие в основе операционных систем, применимы ко многим другим областям программирования, и особенно к системному программированию;
- операционная система - большая и очень сложная программа, на примере которой можно изучать вопросы создания сложных программных продуктов;
- такие популярные программные продукты, как системы управления базами данных, могут рассматриваться как надстройки над операционными системами.

7.1. Введение в операционные системы

7.1.1. Основные понятия и определения

Операционная система (ОС) - часть программного обеспечения, выступающая в качестве интерфейса между приложениями (и пользователями) и аппаратурой компьютера.

Операционная система выполняет две основные функции [Олифер, Олифер 2001].

- Предоставление пользователю-программисту вместо реальной аппаратуры компьютера расширенной виртуальной машины, с которой удобнее работать. Виртуальная машина - это вычислительная система заданной конфигурации, моделируемая для пользователя программными и аппаратными средствами конкретного реально существующего компьютера. Операционная система является тем слоем программного обеспечения, которое преобразует аппаратную машину в виртуальную. Конфигурация виртуальной машины может существенно отличаться от реальной.
- Повышение эффективности использования компьютера за счет рационального управления его ресурсами. Ресурсы операционной системы можно разделить на две группы (рис. 7.1):
  - программные ресурсы (процессы, виртуальное адресное пространство, подсистема ввода-вывода);
  - аппаратные ресурсы (процессоры, память, устройства).

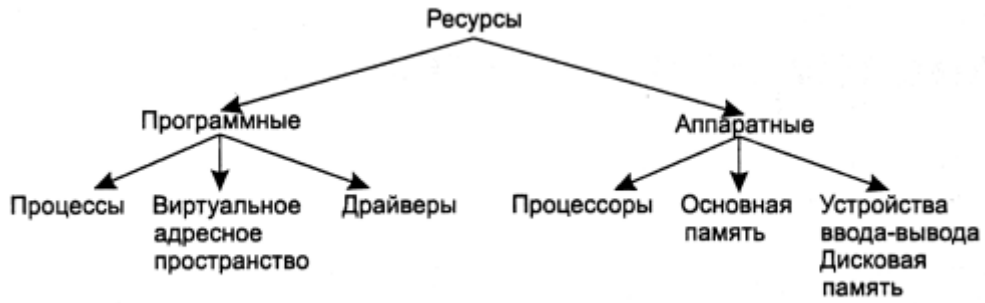

Рис. 7.1. Классификация ресурсов

Обратим внимание на то, что практически каждому аппаратному ресурсу соответствует некоторый программный ресурс, тесно с ним связанный (например, процессор и процесс).

Об основной функции операционной системы

Образно говоря, основной функцией операционной системы можно считать чародейство - превращение системы в нечто большее, чем есть на самом деле. Например, операционная система может создать иллюзию одновременного исполнения нескольких программ на одном процессоре. В итоге пользователь воспринимает виртуальную машину как компьютер, имеющий архитектуру, отличную от реально существующей.

Ядро операционной системы - модули, выполняющие основные функции операционной системы. Эти модули обычно поддерживают управление процессами, памятью, устройствами ввода-вывода. Код ядра операционной системы исполняется в привилегированном режиме работы процессора.

Некоторые компоненты операционной системы представляют собой обычные приложения в стандартном для данной операционной системы формате. Их называют вспомогательными модулями операционной системы. Поэтому часто бывает сложно провести границу между операционной системой и приложениями. Обычно решение о принадлежности некоторой программы операционной системе принимает производитель. Многие компоненты систем программирования, рассмотренные в гл. 5, часто вводятся производителями ОС в качестве ее вспомогательных модулей.

7.1.2. История и эволюция операционных систем

7.1.2.1. Поколения операционных систем

Принято выделять исторические поколения операционных систем, приведенные ниже.

- Нулевое поколение. В первых компьютерах операционные системы отсутствовали. Это период с момента появления первых компьютеров по середины 50-х годов XX века.
- Первое поколение. Пакетная обработка, мультипрограммные операционные системы. Появились в середине 50-х годов XX века.
- Второе поколение. Многорежимные операционные системы, операционные системы реального времени. Появились в середине 60-х годов XX века.
- Третье поколение. Операционные системы для персональных компьютеров, сетевые операционные системы. Появились, в начале 80-х годов XX века.
- Четвертое поколение. Распределенные операционные системы. Появились в начале 90-х годов XX века.

7.1.2.2. Краткий обзор истории создания операционных систем

Первая операционная система появилась в середине 50-х годов XX века. Она была создана в исследовательской лаборатории компании General Motors для компьютера IBM-701. Практически до середины 60-х годов XX века операционные системы распространялись бесплатно. На рис. 7.2 приведены наиболее известные операционные системы и указаны годы их создания.

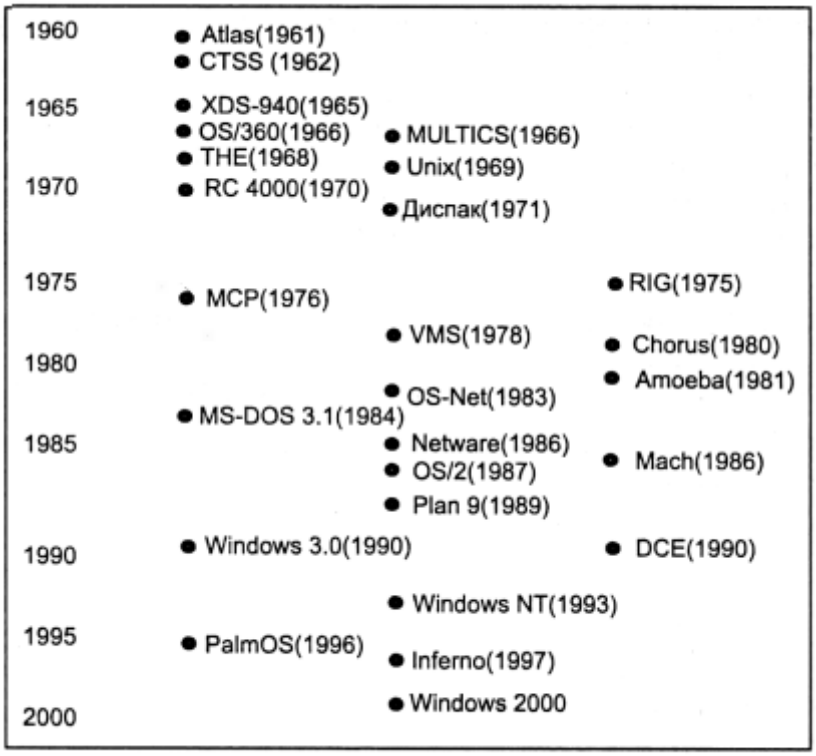

Рис. 7.2. Наиболее известные операционные системы

Дадим краткую характеристику некоторых из них [Silberschatz-Galvin 1995].

- Atlas. Эта операционная система была разработана в Манчестерском университете (Англия). Многие особенности, впервые появившиеся в ней (например, страничное управление памятью), сейчас являются стандартными частями современных операционных систем.
- XDS-940. Система разработана в университете Беркли (США). Это была операционная система с разделением времени. Она давала возможность пользовательской программе определять процессы и работать с ними посредством системных вызовов.
- THE. Система создана в Технической школе Эйндховена (Голландия). Система имела слоистую архитектуру и могла работать с параллельными процессами, выполняя синхронизацию с помощью семафоров.
- CTSS. Система была разработана в Массачусетсом технологическом институте (США), как экспериментальная система с механизмом разделения времени.
- MULTICS. Система также была разработана в Массачусетском технологическом институте и являлась развитием системы CTSS. В свою очередь эта система является предшественницей операционной системы Unix.
- OS/360. Операционная система для большого семейства компьютеров IBM/360. Она была написана на ассемблере тысячами программистов и представляла собой миллионы строк кода.
- Unix. Официальной датой рождения операционной системы Unix считают 1 января 1970 года. С этого момента любая система Unix отсчитывает свое системное время. Реально первая версия этой операционной системы была создана в 1969 году. Истории и эволюции Unix в Интернете посвящено много страниц, например, (http://perso.wanadoo.fr/levenez/unix/).

  О том, как правильно писать - UNIX или Unix
  Питер Салюс (Peter H. Salus) [Salus 1994] приводит слова Дагласа Макилроя (Douglas McIlroy) о том, что написание UNIX заглавными буквами является серьезной ошибкой. Мы далее будем придерживаться этой рекомендации и писать Unix.

Существуют даже "учебные" операционные системы, например ОС NACHOS (Not Another Completely Heuristic Operating System), первая версия которой была создана

Уэйном Кристофером (Wayne Christopher), Стивом Проктером (Steve Procter) и Томасом Андерсоном (Thomas Anderson) в январе 1992 года. Эта операционная система используется практически во всех университетах мира при выполнении студентами самостоятельных заданий по соответствующему учебному курсу (http://www.stanford.edu/class/csl40/projects/).

7.1.3. Классификация операционных систем

7.1.3.1. Классификация по типу централизации

В основу первой и основной классификации положим степень централизации (связности) операционной системы (рис. 7.3).

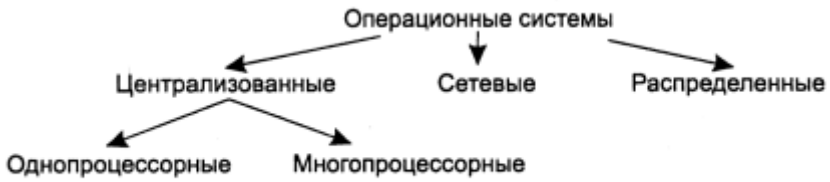

Рис. 7.3. Классификация по типу централизации

Эта классификация принимает во внимание особенности аппаратных платформ, для которых операционные системы создаются.

- Централизованные (локальные) операционные системы. Эти системы управляют ресурсами единственного локального компьютера. Они включают два различных с точки зрения алгоритмов подкласса:
  - однопроцессорные системы;
  - многопроцессорные системы.
- Сетевые операционные системы. Такие системы предоставляют пользователю сети некоторую виртуальную машину, работать с которой проще, чем с реальной сетевой аппаратурой. Однако пользователь всегда выполняет специальные операции для доступа к сетевым ресурсам. Сетевые системы включают дополнительные сетевые средства, состоящие из трех основных компонентов [Олифер, Олифер 2001]:
  - серверная часть операционной системы - средства предоставления локальных ресурсов и услуг в общее пользование;
  - клиентская часть операционной системы - средства запроса доступа к удаленным ресурсам и услугам;
  - транспортные средства операционной системы - средства обеспечения передачи сообщений между компьютерами сети.
- Распределенные операционные системы. Они предоставляют пользователю сети единую централизованную виртуальную машину, которая дает максимальную степень прозрачности сетевых ресурсов. Распределенные системы объединяют все компьютеры сети, для работы в тесной кооперации. При работе в таких системах пользователь, запускающий приложение, не знает, на каком компьютере оно реально выполняется.

Существует интересное обоснование данной классификации [Tanenbaum 1995]. В гл. 6 было отмечено, что основной характеристикой классификации параллельных и распределенных архитектур считают наличие общей или распределенной (локальной для каждого из узлов) памяти. Исходя из этого, вычислительные системы можно разделить на два класса.

- Системы с сильными связями. Сюда относятся системы, состоящие из нескольких однородных процессоров и массива общей памяти.
- Системы со слабыми связями. Это системы, состоящие из однородных вычислительных узлов, каждый из которых имеет свою память.

Программное обеспечение также можно разделить на два класса.

- Программное обеспечение с сильными связями. Сюда относятся программы, которые при исполнении на нескольких вычислительных модулях в большой степени являются связанными между собой.
- Программное обеспечение со слабыми связями. Оно позволяет вычислительным модулям быть независимыми друг от друга, но при необходимости взаимодействовать ограниченным количеством способов.

В результате можно получить четыре различных комбинации между этими парами, три из которых являются осмысленными и определяют следующие типы операционных систем (рис. 7.4).

Рис. 7.4. Обоснование классификации по типу централизации

7.1.3.2. Классификация по особенностям алгоритмов управления ресурсами

Классификация ОС по особенностям алгоритмов управления ресурсами имеет аспекты [Олифер, Олифер 2001], приведенные ниже.

- Поддержка многозадачности.
  - Однозадачные операционные системы выполняют функцию предоставления пользователю виртуальной машины, делая простым и удобным процесс его взаимодействия с компьютером.
  - Многозадачные операционные системы дополнительно управляют разделением совместно используемых ресурсов. В первую очередь они дают возможность одновременно исполнять несколько задач на одном процессоре.
- Поддержка многопользовательского режима.
  - Однопользовательские операционные системы не предоставляют средств зашиты информации одного пользователя от несанкционированного доступа другого пользователя. Такие системы не предоставляют возможностей разделения ресурсов.
  - Многопользовательские операционные системы такие средства защиты информации имеют.
- Поддержка многопоточности. Многопоточные операционные системы дают возможность разделять процессорное время не только между процессами, но и между отдельными ветвями процессов - потоками (см. разд. 7.2.1.2).
- Поддержка многопроцессорной обработки. Многопроцессорные операционные системы реализуют более сложные алгоритмы управления ресурсами, предоставляющие возможность работать с несколькими процессорами.

7.1.3.3. Классификация по особенностям аппаратных платформ

Специфика аппаратных средств, как правило, отражается на специфике операционной системы. В разд. 6.1.3.1 приведена функциональная классификация компьютеров.

Каждый из типов компьютеров в этой классификации имеет определенные свойства, оказывающие непосредственное влияние на свойства операционных систем.

Обратим внимание на то, что наибольший интерес в настоящее время вызывают следующие группы операционных систем:

- операционные системы для мощных серверов;
- операционные системы для рабочих станций и персональных компьютеров;
- операционные системы для карманных компьютеров.

7.1.3.4. Классификация по особенностям областей использования

По особенностям областей использования многозадачные операционные системы могут быть разделены на три типа.

- Операционные системы пакетной обработки. Они работают с пакетами задач, причем переключение процессора с одной задачи на другую происходит лишь в том случае, если активная задача сама отказывается от процессора.
- Операционные системы разделения времени. Такие системы предоставляют каждой из задач некоторый квант процессорного времени. При этом время ответа программы обычно оказывается достаточно приемлемым, что позволяет использовать эти ОС в качестве диалоговых.
- Операционные системы реального времени. Они применяются для управления некоторыми технологическими объектами и процессами. В них существует предельно допустимое время, в течение которого программа должна ответить.

Многие современные операционные системы совмещают в себе свойства систем разных типов. Например, часть задач исполняется в режиме разделения времени, а часть - в режиме реального времени.

7.1.3.5. Классификация по типу архитектуры ядра системы

Существуют основные разновидности архитектуры ядра, приведенные ниже.

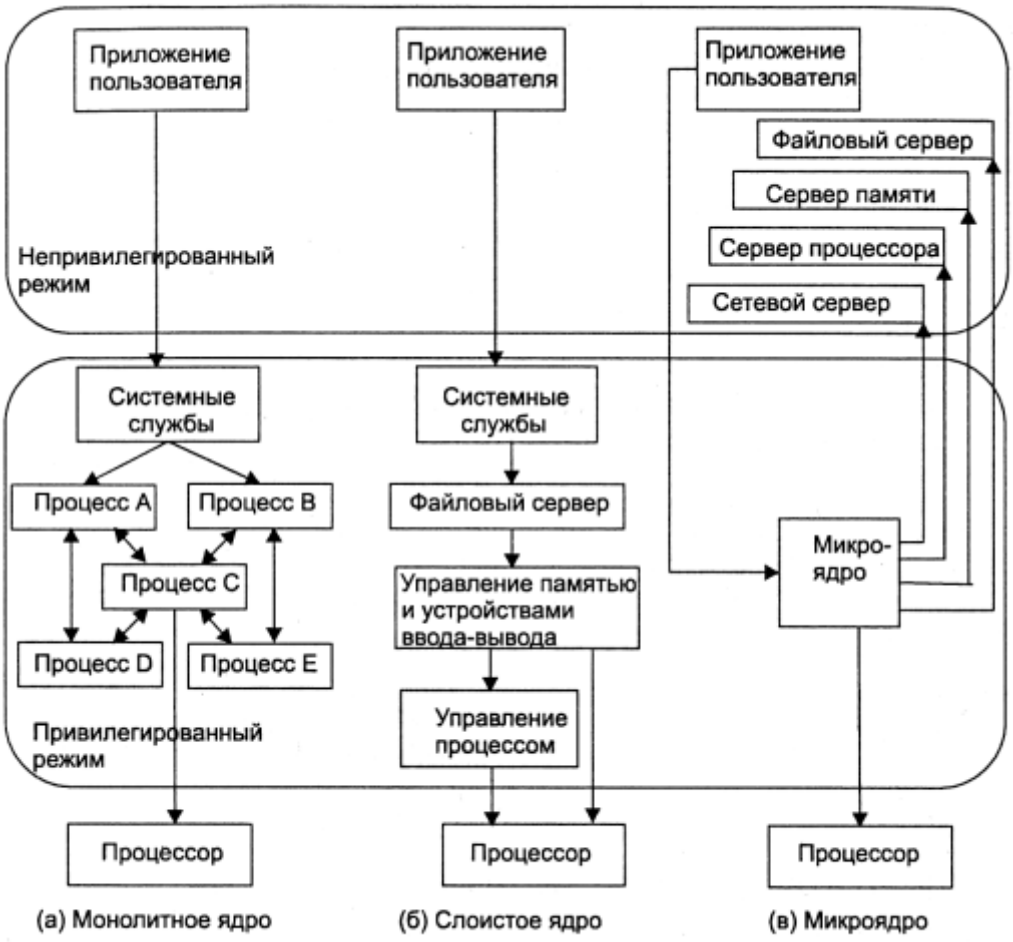

Рис. 7.5. Классификация по типу архитектуры ядра

- Монолитное ядро. Такое ядро компонуется как одна программа, работающая в привилегированном режиме и использующая быстрые переходы с одной процедуры на другую (рис. 7.5, а).
- Слоистое ядро. В этом случае компоненты операционной системы образуют уровни с хорошо продуманной функциональностью и интерфейсом. Как и в предыдущем случае, компоненты работают в привилегированном режиме (рис. 7.5, б).
- Микроядро. Микроядро выполняет минимум функций по управлению аппаратурой. Обычно в него включаются машинно-зависимые программы, некоторые функции управления процессами и обработка прерываний. Функции более высокого уровня выполняют специализированные компоненты операционной системы - сервер процессора, файловый сервер и т. п. Эти компоненты работают в пользовательском, непривилегированном режиме. Данная архитектура основана на подходе "клиент-сервер" и характеризуется переносимостью, расширяемостью и надежностью (рис. 7.5, в).
- Экзоядро. Основная идея данной архитектуры заключается в том, что операционную систему можно собрать, подобно сборке программы, с использованием большой библиотеки функций. В результате она будет включать лишь минимально необходимый набор для некоторой совокупности приложений. Этот тип архитектуры ядра сейчас набирает популярность.

Отметим, что архитектура ядра в значительной степени влияет на всю архитектуру операционной системы.

7.1.4. Проблемы и перспективы развития

Сообщение перед выключением компьютера: Вашу жену зовут Ольга, детей - Катя и Саша. Вы все еще хотите выйти из Windows?
Программистский фольклор

Настоящее время может быть охарактеризовано как время доминирования двух операционных систем: Windows и Unix. Каждая из них прочно заняла свою "нишу" и аппаратную платформу и стала там достаточно популярной. Вот очень краткая сравнительная характеристика двух этих систем.

- Операционная система Windows:
  - распространена на персональных компьютерах;
  - в сетях используется в качестве клиентской операционной системы;
  - имеет удобный графический интерфейс;
  - одна из основных целей - создание окружения, удобного для неопытных пользователей;
  - используется пользователями в широком диапазоне - от профессионалов до "чайников";
  - проектировалась большим количеством людей. Первые версии базировались на предшественнице - операционной системе MS-DOS;
  - не всегда надежна.
- Операционная система Unix:
  - распространена на мощных серверах и рабочих станциях;
  - в сетях используется в качестве серверной операционной системы;
  - исторически долгое время использовалась только с интерфейсом командной строки;
  - одна из основных целей - создание окружения, удобного для развития программ;
  - используется в основном профессионалами;
  - была спроектирована и построена небольшим количеством (двумя - Кен Томпсон (Ken Thompson) и Деннис Ритчи (Dennis Ritchie)) исключительно талантливых людей;
  - достаточно надежна.

О надежности Solaris

О надежности конкретных версий конкретных диалектов Unix известно много фактов. Например, Solaris интенсивно использовалась в одном из университетов без единой перезагрузки в течение четырех лет. Когда на рабочую станцию пришла пора ставить обновленную версию операционной системы, ее физическое местоположение в серверной комнате долго не могли найти, поскольку в этом четыре года не было необходимости.

Поддержка операционными системами различных микропроцессоров отражена в табл. 7.1, а в табл. 7.2 приводится информация о количестве инсталляций для различных операционных систем.

Таблица 7.1. Поддержка микропроцессоров операционными системами

Операционная система	Pentium (Intel)	Alpha (DEC)	SuperSparc (Sun)	PA7100 (HP)
Windows	Windows	-	-	-
OS/2	OS/2	-	-	-
Unix	Solaris, SCO	OSF1	Solaris, SunOS	HP-UX

Таблица 7.2. Инсталляции различных операционных систем

Операционная система	Основной микро-процессор	Другие микро-процессоры	Инсталляции (1996 год)	Инсталляции (2000 год)
MS-DOS	x86	Pentium	180 000 000	5 000 000

Windows 3.*	x86	Pentium	130 000 000	5 000 000
Windows 9x, 2K	Pentium	x86	60 000 000	300 000 000
Windows NT	Pentium	Power PC, Alpha	3 000 000	40 000 000
IBM OS/2	Pentium	x86	12 000 000	25 000 000
Macintosh	Power PC	680	25 000 000	35 000 000
Sun Solaris	SPARC	Pentium	1 500 000	10 000 000
Unix	Много	Много	6 000 000	20 000 000

7.1.5. Рекомендации по литературе

Искусство чтения состоит в том, чтобы знать, что пропустить.
Филип Хамертон

Всю литературу по операционным системам можно разделить на две группы. Первая содержит классическую теорию операционных систем:

- "Операционные системы" [Цикритизис-Бернстайн 1977]. Эта книга является классическим (к сожалению, устаревшим) учебником по теории операционных систем;
- "Сетевые операционные системы" [Олифер-Олифер 2001]. В учебнике рассматриваются современные взгляды на принципы построения операционных систем;
- "Operating system concepts" [Silberschatz-Galvin 1995]. Эта книга является базовым учебником по операционным системам во многих университетах мира;
- "Distributed operating system" [Tanenbaum 1995]. Книга является одним из лучших учебников по распределенным операционным системам.

Вторая группа содержит книги по конкретным операционным системам. Мы рекомендуем следующие книги:

- по операционной системе Windows:
  - "Внутреннее устройство Microsoft Windows 2000" [Соломон, Руссино-вич 2001];
  - "Персональные компьютеры в сетях TCP/IP" [Хант 1997];
- по операционной системе Unix:
  - "Операционная система UNIX" [Робачевский 1997];
  - "UNIX: руководство системного администратора" [Немет, Снайдер, Сибасс, Хейн 1999];
  - "A Quarter Century of UNIX" [Salus 1994].

7.2. Процессы

Понятие процесса является одним из основных в современных операционных системах. В этом разделе мы подробно рассмотрим сущность процессов и способы их коммуникации и синхронизации в централизованных, сетевых и распределенных операционных системах.

7.2.1. Процессы и потоки (нити) управления

7.2.1.1. Понятие процесса

Мы дадим несколько определений процесса (process), отражающих различные стороны этого важного понятия.

- Процесс - это абстракция, описывающая выполняющуюся программу.
- Процесс - исполнение последовательности действий в среде, включающей собственно выполняющуюся программу, а также связанных с ней данных и состояний (открытых файлов, текущего каталога и т. п.).
- С точки зрения операционной системы, процесс - единица работы, заявка на потребление системных ресурсов.
- Процесс - объект, которому выделяется процессор.
- Процесс - живая душа программы.

Первое упоминание о процессе появилось в 60-е годы XX века в операционной системе MULTICS.

Если говорить о соотношении между процессом и программой, то справедливы следующие два утверждения:

- программа - это часть состояния процесса. С этой точки зрения процесс - нечто большее, чем просто программа;
- программа может вызывать более чем один процесс для выполнения работы. С этой точки зрения программа - нечто большее, чем процесс.

Процессы образуют иерархию в операционной системе. Соответственно, будем называть порожденные процессы - потомками данного процесса, а родителя порожденного процесса - предком.

Отметим основные состояния процесса (рис. 7.6):

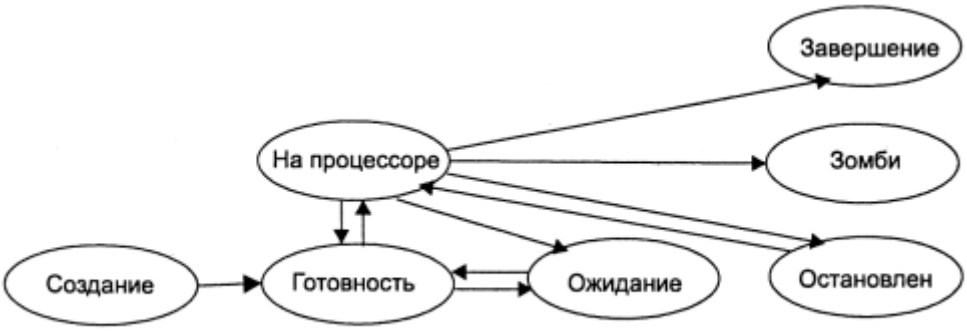

Рис. 7.6. Основные состояния процесса

- на процессоре - активное состояние, в котором процесс обладает всеми необходимыми ресурсами, в том числе самим процессором;
- готовность - процесс находится в очереди на выполнение;
- ожидание - процесс ожидает завершение события (например, освобождения ресурса);
- остановлен - процесс остановлен, как правило, в отладочном режиме;
- создание - выполнение действий, необходимых для создания процесса;
- завершение - выполнение действий, связанных с успешным завершением процесса;
- зомби - процесс закончен, но предок не принял его завершения.

При создании процесса должны быть выполнены следующие действия:

- присвоение процессу уникального номера (ID);
- добавление процесса в список процессов, известных системе;
- определение начального приоритета;
- формирование блока управления процессом;
- выделение необходимых ресурсов.

В листинге 7.1 приведена простейшая программа на языке C, порождающая новый процесс в ОС Unix.

Листинг 7.1. Порождение процесса в ОС Unix

```
void main() {
    int status;
    int child_pid, child_status;
    child_pid = fork();
    if (child_pid == 0) { /* находимся в потомке */
      execlpC'ls", "ls", (char*)0);
      exit(1); /* оказываемся здесь только в случае неудачного
              исполнения вызова execlp */
    }
    else if (child_pid != -1) { /* находимся в предке */
       status = wait(&child_status);
       printf(" %d %d \n", status, child_status);
    }
}
```

В листинге 7.2 приведена простейшая программа на языке С, порождающая новый процесс в ОС Windows.

Листинг 7.2. Порождение процесса в ОС Windows

```
void main() {
    STARTUPINFO si;
    PROCESS_INFORMATION pi;
    ZeroMemory(&si, sizeof(si));
    si.cb = sizeof(si);
    ZeroMemory(&pi, sizeof(pi));
    if( !CreateProcess(NULL, "c:\\windows\\notepad.exe",
            NULL, NULL, FALSE, 0, NULL, NULL,
            &si, &pi)) {
        ErrorExit("Не удалось создать процесс.");
    }
    WaitForSingleObject(pi.hProcess, INFINITE); // ждем завершения
    CloseHandle(pi.hProcess); // освобождаем ресурсы
    CloseHandle(pi.hThread);
}
```

7.2.1.2. Процессы с поддержкой многопоточности

Поток (нить) управления (thread) - исполнение команд программы в естественном порядке. Процессы делятся на традиционные (имеющие один поток управления) и многопоточные (многонитевые).

Многопоточность в рамках одного процесса имеет существенные преимущества. Переключение контекста между двумя потоками в одном процессе значительно проще, чем переключение контекста между двумя процессами. Все данные, за исключением стека исполнения и содержимого регистров процессора, разделяются между потоками. Таким образом, Многопоточность предоставляет эффективный параллелизм.

Можно указать целые классы программ, где необходима Многопоточность:

- операционные системы;
- сетевые серверы;
- встроенные системы;
- вычислительные программы.

Одна из наиболее элегантных реализаций потоков выполнена в ОС Solaris. Все потоки (нити) можно разделить на три класса (рис. 7.7).

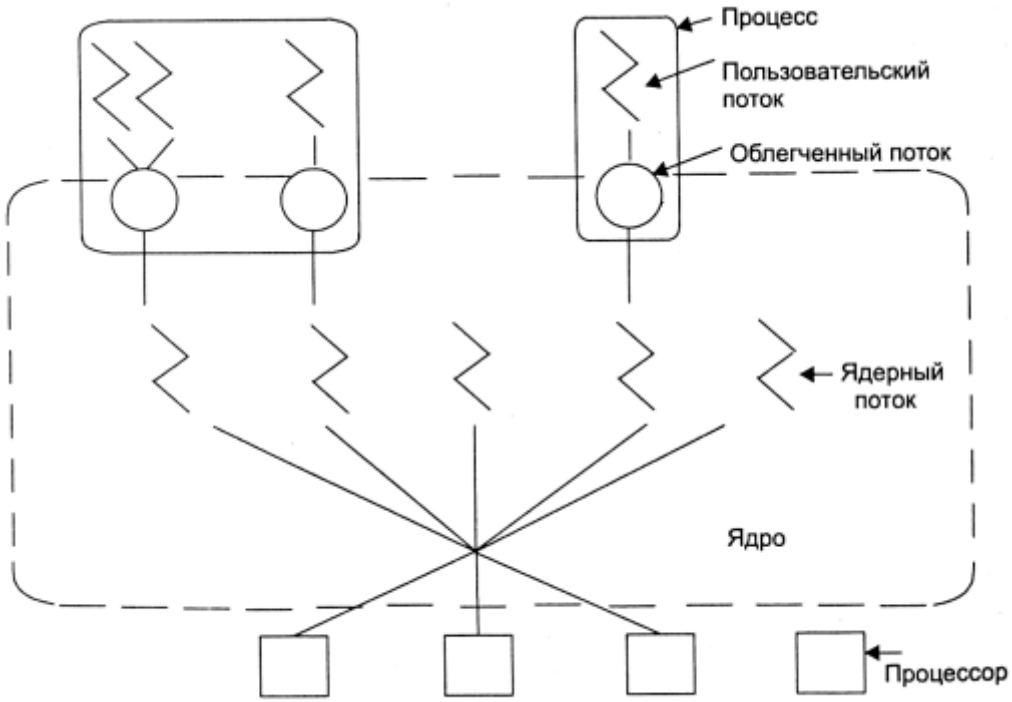

Рис. 7.7. Классы потоков

- Пользовательские потоки, для создания которых пользователь работает со стандартной библиотекой. Пользовательские потоки связываются с облегченными потоками.
- Потоки ядра являются базовыми потоками. Они располагаются в адресном пространстве ядра и непосредственно связаны с процессами.
- Облегченные (легковесные) потоки служат для организации пользователем нескольких потоков управления в адресном пространстве. Каждому облегченному потоку соответствует свой отдельный ядерный поток. Каждый облегченный поток может отдельно планироваться (на каждый процессор - по потоку). Облегченные потоки следует рассматривать как диспетчеризуемые сущности.

Об использовании потоков
Как правило, технику явного распараллеливания не следует использовать при прикладном программировании. Нужно переложить заботу о распараллеливании на компиляторы, возможно помогая ему директивами (например, OpenMP). Однако системному программисту знание и умение применять нити в своих программах необходимо.

Многие операционные системы, ориентированные на рабочие станции и персональные компьютеры, начиная с середины 80-х годов XX века, включают поддержку многопоточности. Впервые стандарт на потоки появился в 1995 году. Это был стандарт IEEE POSIX 1003.1c-1995. Однако стандарт появился достаточно поздно, и некоторые компании успели выпустить свои версии многопоточных библиотек, существенно отличающихся от стандарта. Можно выделить следующие основные семейства потоков:

- потоки, поддерживающие стиль POSIX-стандарта. Это семейство состоит из трех подгрупп:
  - "истинные" POSIX-потоки. Это потоки, базирующиеся на стандарте IEEE POSIX ЮОЗЛс-1995 (также известного как ISO/IEC 9945-1:1996), являющегося частью стандарта ANSI/IEEE 1003.1;
  - DCE-потоки, базирующиеся на ранней версии стандарта POSIX - 1003. la;
  - Unix International потоки, также известные как Solaris-потоки [Lewis, Berg 1995]. Они достаточно близки к стандарту и поддерживаются в операционных системах Solaris компании Sun Microsystems и UnixWare 2 компании SCO;
- потоки Microsoft. Это семейство состоит из двух подгрупп, причем обе разработаны в компании Microsoft:

  - потоки WIN32, являющиеся стандартными для семейства операционных систем Windows, включая Windows 2000, Windows 95, Windows 98, Windows ME и Windows CE;
  - потоки OS/2, являющиеся стандартными для операционной системы OS/2 компании IBM;
- другие варианты потоков. Их не так много. Отдельного упоминания заслуживает лишь пакет C threads, имеющийся в операционной системе Mach.

Различные семейства предлагают различный синтаксис основных функций потоковых библиотек. Основными группами функций являются: функции создания потоков и функции синхронизации потоков (как правило, набор таких функций достаточно богат и разнообразен).

В листинге 7.3 приведена простейшая программа, демонстрирующая различие в синтаксисе функций потоков POSIX и Solaris. Компилировать ее надо с указанием нитевых библиотек, например:

```
% ee thread_example.c -Ithread -Ipthread
```

Листинг 7.3. Пример создания и использование потоков POSIX и Solaris

```
#define _REENTRANT
#include <pthread.h>
#include <thread.h> #define NUM_THREADS 5

#define SLEEP_TIME 10 void *sleeping(void *); /* функция, которая будет исполняться
как поток */
int i;
thread_t tid[NUM_THREADS]; /* массив для хранения номеров потоков */
void main(int argc, char *argv[]){
    switch ( *argv[1]) (
    case '0' : /* POSIX */
        for ( i = 0; i < NUM_THREADS; i++)
            pthread_create(&tid[i], NULL, sleeping, SLEEP_TIME);
        for ( i = 0; i < NUM_THREADS; i++)
            pthread_join(tid[i], NULL);
        break;
    case '1': /* Solaris */
        for ( i = 0; i < NUM_THREADS; i++)
            thr_create(NULL,0,sleeping,NULL,0,&tid[i]);
        while (thr_join(NULL, NULL, NULL) == 0)
            ;
        break;
    } /* switch */
    printf("Все %d потоков управления завершены \n", i);
}

void *sleeping (int sleep_tirue *) {
    printf("Поток управления %d приостановлен на %d секунд\n",
            thr_self(), SLEEP_TIME);
    sleep(sleep_time);
    printf("Исполнение потока управления %d возобновлено\n", thr_self());
}
```

О безопасности использования фрагментов кода в многопоточных программах

В документации к некоторым фрагментам кода (как правило, к системным функциям) стоит пометка о том, что они безопасны с точки зрения использования в многопоточных программах (MT-safe). Отсутствие подобной пометки может означать наличие в таких функциях глобальных или статических переменных, приводящих к побочным эффектам.

7.2.1.3. Сигналы как простейшие средства коммуникации

Сигналы (signals) - программное средство, с помощью которого может быть прервано функционирование процесса в операционной системе Unix. Механизм сигналов позволяет процессам реагировать на различные события, которые могут произойти в ходе функционирования процесса внутри него самого или во внешнем мире [Lewine 1995].

Каждому сигналу ставится в соответствие:

- номер сигнала;
- символическая константа, используемая для осмысленной идентификации сигнала;
- принадлежность к одному из пяти классов сигналов;
- реакция процесса на сигнал:
  - вызов обработчика сигнала;
  - завершение процесса (возможно, с порождением дампа памяти);
  - игнорирование сигнала;
  - приостановка исполнения.

Следующие три ситуации приводят к генерации сигналов операционной системой:

- ядро отправляет процессу сигнал при нажатии пользователем определенной комбинации клавиш на клавиатуре;
- возникновение аппаратных особых ситуаций приводит к посылке уведомления ядру операционной системы, которое отправляет сигнал процессу, находящемуся в стадии выполнения;
- определенное программное состояние системы может вызвать отправку сигнала.

Наиболее интересными являются следующие типы сигналов:

- сигналы от оборудования, порождаемые при выполнении процесса в результате возникновения некоторых исключительных условий. Некоторые примеры сигналов:
  - SIGFPE (8) - исключительная ситуация при выполнении операции с плавающей или фиксированной точкой;
  - SIGILL (4) - попытка выполнить неверную машинную команду;
  - SIGSEGV (11) - нарушение защиты памяти;
  - SIGBUS (10) - ошибка шины (адресации);
- программные сигналы, либо генерируемые пользователем с клавиатуры, либо посылаемые одним процессом другому. Примеры сигналов:
  - SIGINT (2) - сигнал прерывания от терминала (пользователь нажал комбинацию клавиш <Ctrl>+<C>);
  - SIGQUIT (3) - сигнал аварийного завершения работы от терминала (пользователь нажал комбинацию клавиш <Ctrl>+<\>);
  - SIGTERM (15) - сигнал программного прерывания, посылаемого по умолчанию командой kill;
  - SIGKILL (9) - сигнал программного прерывания, который не может быть ни перехвачен, ни проигнорирован никаким процессом;
  - SIGHUP (1) - разрыв связи с терминалом;
- сигналы контроля процессов, либо генерируемые пользователем с клавиатуры, либо посылаемые одним процессом другому. Ряд этих сигналов используется отладчиками для посылки сигналов процессу с отлаживаемой программой. Примеры сигналов:
  - SIGSTOP (23) - остановка процесса;
  - SIGTSTP (24) - остановка процесса пользователем (пользователь нажал комбинацию клавиш <Ctrl>+<z>);
  - SIGCONT (25) - продолжение приостановленного процесса;
  - SIGCHLD (18) - изменение статуса порожденного процесса.

В листинге 7.4 приведена программа на языке Pascal, исполнение которой приведет к генерации сигнала SIGSEGV.

Листинг 7.4. Пример порождения сигнала SIGSEGV

```
program SegmentationViolation;
type
    Pinteger = "integer;
var
    IntVar: integer;
    NullPtr: Pinteger;
begin
    NullPtr := nil;
  { Попытка выполнить следующее присваивание приведет к сигналу SIGSEGV }
    IntVar := NullPtr";
end.
```

Листинг 7.5 содержит пример программы, устанавливающей обработчик сигнала прерывания от терминала.

Листинг 7.5. Пример перехвата и обработки сигнала прерывания от терминала

```
#include <signal.h>
static void reaction (int i){
    printf("Вызван обработчик сигнала %d \n", i);
    exit(i);
}

void main(){
    if (signal(SIGINT, SIG_IGN) != SIG_IGN)
        signal(SIGINT, reaction);
    printf("Перед подачей сигнала с клавиатуры \n") ;
    while (1);
    printf("После подачи сигнала с клавиатуры \n");
}
```

Запуск на исполнение данной программы и последующее нажатие пользователем комбинации клавиш <Ctrl>+<C> приведет к выводу следующего результата работы программы:

```
% a.out

Перед подачей сигнала с клавиатуры

Вызван обработчик сигнала 2
```

Листинг 7.6 содержит более сложный пример программы, в котором после обработки сигнала прерывания от терминала происходит возвращение к работе программы. Для этого используется нелокальный переход.

Листинг 7.6. Пример использования стека восстановления

```
#include <signal.h>
#include <setjmp.h>

static jmp_buf jb;
static void reaction (int i){
    printf("Вызван обработчик сигнала %d \n", i);
    longjmp(jb, i);
}

void main(){
    if (signal(SIGINT, SIG_IGN) != SIG_IGN)
        signal(SIGINT, reaction);
    if (!setjmp(jb))
        printf("Ветка исполнения после setjmp \n");
```

```
    else
        printf("Ветка исполнения после longjmp \n");
    printf("Перед подачей сигнала с клавиатуры \n");
    while (1);
}
```

Запуск на исполнение данной программы и последующее нажатие пользователем комбинации клавиш <Ctrl>+<C> приведет к выводу следующего результата работы программы:

```
% a.out
Ветка исполнения после setjmp
Перед подачей сигнала с клавиатуры
Вызван обработчик сигнала 2
Ветка исполнения после longjmp
Перед подачей сигнала с клавиатуры
%
```

7.2.2. Коммуникация и синхронизация процессов в централизованных архитектурах

7.2.2.1. Основные понятия и определения

Процессы называются параллельными, если они существуют одновременно. Параллельные процессы можно разделить на следующие две группы:

- независимые (не нуждающиеся во взаимодействии друг с другом) процессы;
- асинхронные (взаимодействующие и нуждающиеся в периодической синхронизации) процессы.

Синхронизация процессов - использование специальных атомических операций для осуществления взаимодействия между процессами.

Критический ресурс - ресурс, допускающий обслуживание только одного процесса за один раз. Если несколько процессов хотят использовать критический ресурс в режиме разделения, то им следует синхронизировать свои действия, чтобы ресурс всегда находился в распоряжении не более чем одного из них.

Критические участки -- участки процесса, где происходит обращение к критическому ресурсу. Критические участки должны быть взаимоисключаемыми, т.е. в каждый момент времени не более чем один процесс может быть занят выполнением своего критического относительно некоторого ресурса участка. Обеспечение (поддержка механизма) взаимоисключения - ключевая задача параллельного программирования.

Блокировка - предотвращение выполнения кем-либо чего-либо. Процесс должен устанавливать блокировку перед входом в критический участок и снимать ее после выхода. Естественно, если участок заблокирован, то другой процесс должен ждать снятия блокировки.

Вход взаимоисключения и выход взаимоисключения - участки процесса, обрамляющие критический участок и служащие для обеспечения взаимоисключения. Примером входа взаимоисключения может служить блокирование, а выхода - разблокирование.

Типичным примером критического ресурса является разделяемая переменная, суммирующая некоторую величину (назовем ее счетчик). Критические участки процессов тогда могут содержать,следующий код:

```
счетчик := счетчик + 1.
```

Гонки - ситуация, когда два или более процессов обрабатывают разделяемые данные и конечный результат зависит от соотношения скоростей их исполнения.

Об использовании термина "процесс"
Мы будем использовать в определениях и алгоритмах этого раздела термин "процесс", следуя традиции. Более корректно было бы использовать понятие "поток управления".

7.2.2.2. Алгоритм Деккера

В листинге 7.7 приведена реализация алгоритма Деккера на языке С. Это программный вариант решения проблемы взаимного исключения. Алгоритм учитывает следующие требования:

- относительные скорости параллельных процессов могут быть любыми;
- процессы вне критического участка не могут препятствовать другим процессам входить в критический участок;
- не должно быть бесконечного откладывания входа в критический участок.

Листинг 7.7. Алгоритм Деккера

```
enum state { UNLOCKED, LOCKED };
typedef struct {
char status [2]; /* байт статуса для каждого из двух процессов */
char turn;      /* какой из процессов будет следующим */
} lock_t;
void init_lock(lock_t* lock) {
     lock->status[0] = UNLOCKED;
     lock->status[1] = UNLOCKED;
     lock->turn = 0;
}
void lock(volatile lock_t* lock) {
    /* устанавливаем блокировку для текущего процесса */
    lock->status[cur_proc_id()] = LOCKED;
    /* проверяем, не установлена ли блокировка другим процессом */
    while ( lock->status[other_proc_id()] == LOCKED) (
      /* если другой процесс уже установил блокировку,
         проверяем - чья очередь войти в критический участок */
      if ( lock->turn != cur_proc__id () ) {
         lock->status[cur_proc_id()) = UNLOCKED;
         while ( lock->turn == other_proc_id())
            ;

         lock->status[cur_proc_id()] = LOCKED;
      }
    }
}
void unlock(lock_t* lock) {
    lock->status[cur_proc_id()] = UNLOCKED;
    lock->turn = other_proc_id();
}
```

7.2.2.3. Аппаратная поддержка взаимоисключений

Возможное аппаратное решение поддержки проблемы взаимного исключения - наличие неделимой команды test_and_set(a, b) (проверить и установить). Команда имеет две логические переменные в качестве параметра. Переменная а является локальной для каждого процесса, а переменная ь - глобальная и разделяется процессами. Выполнение команды заключается в следующих двух действиях:

- значение переменной ь копируется в а;
- значение переменной ь устанавливается в истину.

Практически все основные архитектуры имеют подобную команду в своем составе. Например, в S PARC-архитектуре существует команда idstub (load store unsigned byte) (см. разд. 6.4.3.6):

Idstub [addr] , reg

В результате выполнения этой команды содержимое памяти по адресу addr копируется в регистр reg, а все биты памяти addr устанавливаются в единицу. В листинге 7.8 представлен программный интерфейс команды test_and_set с помощью реальной ассемблерной команды idstub.

Листинг 7.8. Программный интерфейс команды test_and_set

```
int test_and_set (volatile int* addr) {
    asm (Idstub [addr], reg);
    if ( reg = 0)
     return 0;
    return 1;
}
```

7.2.2.4. Крутящаяся блокировка

Крутящаяся блокировка - механизм, реализующий взаимное исключение и являющийся достаточно эффективным для коротких критических участков. Название его вытекает из того факта, что процесс будет находиться в цикле, ожидая завершение блокировки, установленной другим процессом. В листинге 7.9 программная реализация крутящейся блокировки использует неделимую команду test_and_set.

Листинг 7.9. Крутящаяся блокировка

```
typedef int lock_t;
void init_lock(lock_t* lock_status) {
    *lock_status = 0;
}
void lock(volatile lock_t* lock_status) {
    while ( test_and_set(lock_status) == 1)
      ;
}
void unlock(lock_t* lock_status) {
    *lock_status = 0;
}
```

7.2.2.5. Блокировка с запретом прерываний

Блокировка с запретом прерываний применима только на однопроцессорной архитектуре. В листинге 7.10 приведен пример реализации такой блокировки.

Листинг 7.10. Блокировка с запретом прерываний

```
enum state { UNLOCKED, LOCKED };
typedef int lock_t;
void init_lock(lock_t* lock) {
    *lock = UNLOCKED;
}
void lock (volatile lock_t* lock) {
    disable_interrupts; /* запрет прерываний */
    while ( lock != UNLOCKED) {
      enable_interrupts; /* разрешение прерываний */
      disable_interrupts;
    }
    lock = LOCKED;
    enable_interrupts;
}
void unlock(lock_t* lock) {
    disable_interrupts;
    lock = UNLOCKED;
    enable_interrupts;
}
```

7.2.2.6. Семафоры

Семафор - защищенная переменная, значение которой можно запрашивать и менять только при помощи специальных операций p и v и при инициализации. Концепция семафоров была предложена Дейкстрой в начале 60-х годов XX века. Применяют три основных типа семафоров:

- двоичные (бинарные) семафоры, принимающие только два значения {0,1};
- считающие семафоры: их значения - целые неотрицательные значения;
- общие семафоры: принимают все множество целых значений. Реализация операций p и v для считающих семафоров выглядит так:

```
P(S): if ( S>0)
      then S:=S-1
      else ожидать_в_очереди(S)
V(S): if ( есть_процессы_в_очереди(3))
      then одному_продолжить (S)
      else S:=S+1.
```

Конечно, операции p и v должны быть неделимыми (например, защищенные крутящейся блокировкой). Как правило, семафоры реализуются в ядре операционной системы. Собственно название операций происходит от голландских слов Proberen - проверить и Verhogen - увеличить.

Три классические задачи, в которых применяются семафоры, таковы:

- решение проблемы взаимного исключения при помощи семафоров. Входом взаимоисключения будет операция p, а выходом - v;
- синхронизация при помощи семафоров. Если одному процессу необходимо, чтобы он получал уведомление от другого процесса о наступлении некоторого события и только после этого продолжал свою работу, то процессы должны иметь операции p и v соответственно, причем инициализироваться семафор должен нулем;
- выделение нескольких однотипных ресурсов из пула ресурсов при помощи семафоров. В этом случае надо применять общие или считающие семафоры.

Одна из возможных реализаций семафоров представлена в листинге 7.11.

Листинг 7.11. Реализация семафоров

```
typedef struct {
   lock_t  lock; /* байт статуса для каждого из двух процессов */
   int     count; /* какой из процессов будет следующим */
   proc_t* head; /* */
   proc_t* tail
} sema_t;
void init_sema(sema_t* seraa, int initial_count) {
   init_lock(&sema->lock);
   sema->head = NULL;
   sema->tail = NULL;
   sema-> count = initial_count;
}
void P(sema_t* sema) {
   lock(&sema->lock);
   sema->count-- ;
   if ( sema->count < 0) {
     if ( sema->head == NULL)
        sema->head = u.u_procp;
     else
        sema->tail~>p_next = u.u_procp;
     u.u_procp->p_next = NULL;
     sema->tail = u.u_procp;
     unlock(&sema->lock);
     switch();
     return;
```

```
    }
    unlock(&sema->lock);
}
void V(sema_t* sema) {
    proc_t* p;
    lock(&sema->lock);
    sema->count++;
    if ( sema->count <= 0) {
      p = sema->head;
      sema->head = p->p_next;
      if ( sema->head == NULL)
        sema->tail = NULL;
      unlock(ssema->lock);
      enqueue(Srunqueue, p);
      return;
    }
    unlock(&sema->lock);
}
```

7.2.2.7. Мониторы

Монитор - механизм организации параллелизма, который содержит как данные, так и процедуры, необходимые для динамического распределения конкретного общего ресурса или группы общих ресурсов.

Для выделения нужного ресурса процесс должен обратиться к конкретной процедуре монитора. В каждый конкретный момент времени монитором может пользоваться только один процесс. Процессам, желающим войти в монитор в тот момент, когда он занят, приходится ждать, причем процессом ожидания управляет монитор. Монитор обладает свойством инкапсуляции данных.

Принцип работы монитора может быть описан следующим образом:

- процесс, обращающийся к монитору за получением некоторого ресурса, обнаруживает, что ресурс занят. При этом процедура монитора выдает команду "ждать" (wait), по которой процесс будет ждать вне монитора, пока ресурс освободится;
- когда ресурс освобождается, то монитор выдаст команду "сигнал" (signal). Если очередь ждущих процессов не пуста, то по этой команде один из процессов может воспользоваться ресурсом монитора. Обычно очередь организуется по принципу "первый пришел - первый получил доступ к ресурсу".

Существует традиционный подход к реализации мониторов и очереди процессов. Очередь поддерживается переменной условия (condition variable). При этом в команды включаются имена условий, например, wait (имя_условия) ИЛИ signal(имя_условия).

В листинге 7.12 приведена реализация двоичного семафора при помощи монитора. Пример написан на языке, близком к языку Pascal, но расширенному Конструкцией monitor.

Листинг 7.12. Реализация двоичного семафора при помощи монитора

```
monitor sema;
var
    s: integer = 1; /* переменная, представляющая семафор */
    res_is_free: cond_t;
procedure P; /* захват ресурса */
begin
    if s = 1
    then s := 0
    else wait(res_is_free)
end;
procedure V; /* освобождение ресурса */
begin
```

```
   s := 1;
   signal(res_is_free)
end;
end monitor;
```

7.2.2.8. Рандеву в языке программирования Ada

В языке программирования Ada существует специальный встроенный механизм параллельного программирования - task (задача). Одни из задач могут обращаться к другим с помощью механизма вызовов и обслуживания входов. Рандеву происходит в момент, когда вызов входов принимается оператором accept и представляет собой выполнение операторов. Точка входа может обслужить только одну связанную с ней точку вызова. Остальные будут помещены в очередь ожидания.

7.2.2.9. Решение задачи передачи данных между процессами "читатель-писатель"

Существует классическая задача передачи данных между процессами "читатель-писатель". Процесс "писатель" генерирует информацию, а "читатель" ее потребляет. Обмен блоками информации осуществляется через специальные буферы, число которых ограничено. Одним из возможных вариантов буфера может быть циклический список. В листингах 7.13-7.16 эта задача решается с помощью блокировок, семафоров, мониторов и средств языка программирования Ada, соответственно.

Листинг 7.13. Решение задачи с помощью блокировок

```
char buf[N];
int head = 0, tail =0, n = 0;
lock_t lock_status;
void put(char c) {
   while ( n == N)
     wait ();
   lock(lock_status);
   buf[head] = c;
   head = (head + 1)%N;
   n++;
   unlock(lock_status) ;
}
char get() (
   while ( n == 0)
      wait ();
   lock(lock_status);
   c = buf[tail];
   tail = (tail + 1)%N;
   n--;
   unlock(lock_status);
   return c;
}
```

Листинг 7.14. Решение задачи с помощью семафоров

```
char buf[N];
int head = 0, tail = 0;
sema_t holes, chars;
init_sema(holes, N);
init_sema(chars, 0);
void put(char c) {
   P(holes);
   buf[head] = c;
   head = (head + 1)%N;
   V(chars);
}
char get() {
   P(chars);
   c = buf[tail];
   tail = (tail + 1)%N;
   V(holes);
```

```
   return c;
}
```

Листинг 7.15. Решение задачи с помощью мониторов

```
char buf[N];
int head = 0, tail = 0, n = 0;
cond_t not_empty, not_full;
void put(char c) {
   while (n = N)
      wait(not_full);
   buf[head] = c;
   head = (head + 1)%N;
   n++;
   signal(not_erapty);
}
char get() {
   while (n == 0)
     wait(not_empty);
   c = buf[tail];
   tail = (tail + 1)%N;
   n--;
   signal(not_full);
   return c;
}
```

Листинг 7.16. Решение задачи на языке программирования Ada

```
task body Buffer is
   buf: array(0..N-1) of CHAR;
   head, tail: INTEGER range 0..N-1 := 0;
   n: INTEGER range 0..N := 0;
begin
   loop
      select
         when n < N => accept put(in c: CHAR) do
            buf[head] = c;
         end;
         n +:= 1;
         head = (head + 1) mod N;
      or
         when n > 0 => accept get(out c: CHAR) do
            c = buf[tail];
         end;
         n -:= 1;
         tail = (tail + 1) mod N;
     end select;
   end loop;
end Buffer;
```

Существует частный случай задачи передачи данных между процессами "читатель-писатель". Это случай информационной базы данных, который характеризуется следующим образом. Процессов-"читатель" гораздо больше, чем процессов-"писатель". "Читатели" не изменяют содержимое базы данных, следовательно, их может быть несколько, при этом они одновременно имеют доступ к базе данных. "Писатели" должны иметь монопольный доступ к базе данных.

7.2.2.10. Тупики

Процессы и потоки управления - активные объекты. Ресурсы - неактивные объекты (процессор - вытесняемый ресурс, дисковое пространство - невытесняемый ресурс). Во время своей работы процесс (поток управления) может попасть в два неприятных состояния: зависание и тупик.

Зависание - состояние неопределенного ожидания, из которого рано или поздно процессы выходят. Связано с ожиданием каких-либо ресурсов.

Тупик - состояние ожидания некоторого события, которое никогда не произойдет (как правило, это круговое ожидание ресурсов).

Система находится в тупиковой ситуации, если один или более процессов находятся в состоянии тупика.

Существуют четыре необходимых условия для возникновения тупика [Цикритизис, Бернстайн 1977].

- Условие взаимоисключения (процессы требуют монопольного владения ресурсами, им предоставляемыми).
- Условие ожидания (процессы удерживают уже выделенные им ресурсы, ожидая выделения дополнительных).
- Условие нераспределяемости (ресурсы нельзя отобрать у удерживающих их процессов, пока они не будут использованы).
- Условие кругового ожидания (существует кольцевая цепь процессов, в которой каждый процесс удерживает за собой один или более ресурсов, требующихся следующему процессу).

Существует четыре основных стратегии работы с тупиками.

- Полное игнорирование проблемы ("алгоритм страуса"). В большинстве своем реальные операционные системы не осуществляют борьбу с тупиками, поскольку ресурсов и так достаточно много.
- Предотвращение тупиков (prevention) - подход, цель которого обеспечение условий, исключающих возможность возникновения тупиковой ситуации. Чтобы предотвратить тупик, достаточно нарушить хотя бы одно необходимое условие.
    - Первое условие (взаимоисключение) вполне естественно (например, для такого устройства, как накопитель на магнитной ленте).
    - Нарушая второе условие (ожидание), процесс должен сразу запросить все ресурсы. Эффективность системы при этом может значительно ухудшиться. В качестве компромиссного решения можно предложить разделять процесс на шаги.
    - Нарушить третье условие (нераспределяемость) можно следующим образом. Процесс, удерживающий ресурсы и получивший отказ на другие ресурсы, должен освободить все взятые ресурсы, и через некоторое время запросить их заново. При этом процесс может потерять большую часть проделанной работы.
    - Нарушая четвертое условие (кругового ожидания), следует ввести линейную упорядоченность ресурсов, пронумеровав их, и выдавать их в порядке, не допускающим возникновение кругового ожидания. Данный подход требует значительных накладных расходов на хранение информации, связанной с типами ресурсов и упорядоченностью экземпляров ресурсов.
- Обход тупиков (avoidance) - подход, который обеспечивает рациональное распределение ресурсов по рациональным правилам. Он вводит менее жесткие ограничения, чем предыдущий подход. Наиболее известным методом обхода тупиков является алгоритм банкира.
    - Алгоритм банкира имитирует действия банкира, располагающего определенным источником капитала, выдающего ссуды и принимающего платежи. Алгоритм был предложен Дейкстрой.
    - Состояние системы будем называть надежным, если операционная система может обеспечить всем пользователям завершение их задач в течение конечного промежутка времени. Суть алгоритма в том, что система удовлетворяет только те запросы, при которых ее состояние остается надежным. Остальные запросы откладываются.
    - Существуют два основных ограничения этого подхода: каждый процесс заранее должен указать максимальное количество ресурсов, которое ему понадобится и в каждый момент процесс должен захватывать только один

ресурс. Данный алгоритм на практике неприменим из-за его неэффективности, т. к. необходимость в пересчете возникает непрерывно, при каждом поступлении процесса в систему и его удалении для каждой разновидности ресурсов.

- Обнаружение тупиков (detection) - подход, который допускает возникновение тупиков, определяет процессы и ресурсы, которые вовлечены в тупиковую ситуацию, и пытается вывести систему из нее.

7.2.2.11. Модели для анализа свойств асинхронных процессов

Для анализа свойств асинхронных процессов (в частности, для обнаружения тупиков) используются графические модели, которые мы сейчас рассмотрим.

Граф распределения ресурсов

Задачу установления факта возникновения тупиковой ситуации удобно решать с помощью графа распределения ресурсов. На рис. 7.8 представлены два основных типа отношений (рис. 7.8, а) на графе (запрос ресурса и владение им) и простейший пример системы в состоянии кругового ожидания (тупика) (рис. 7.8, б).

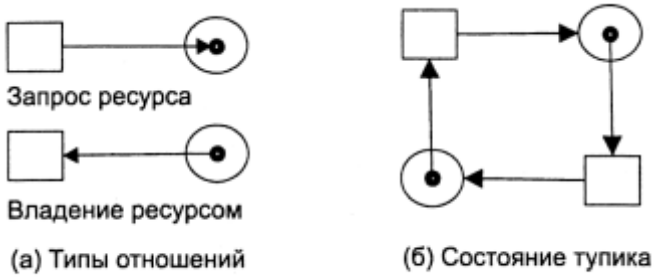

Рис. 7.8. Примеры графа распределения ресурсов

Цля обнаружения тупиков выполняется редукция (приведение) графа. Существует правило редукции - для каждого незаблокированного процесса (т.е. процесса, все запросы которого могут быть удовлетворены) нужно убрать все входящие и исходящие дуги. Граф полностью приводим, если после редукции он не содержит ни одной дуги. В системе отсутствуют тупиковые :итуации, если соответствующий граф полностью приводим.

Сеть Петри

Сеть Петри - помеченный ориентированный граф с двумя типами вершин: позициями и переходами. Позиции изображаются кругами, переходы - квадратами, а пометки - жирными точками в позициях.

Разметка - функция, которая ставит в соответствие пометкам позиции це-тое неотрицательное число. Разметка может быть изменена с помощью сра-эатывания (запуска) перехода. Переход называется запускаемым, если в каж-цой позиции, из которой ведет стрелка в данный переход, есть хотя бы одна четка. Запуск перехода заключается в том, что из каждой позиции, из которой ведет стрелка, число пометок уменьшается на единицу. А в каждую позицию, в которую ведет стрелка, число пометок увеличивается на единицу.

Семантически позиции удобно рассматривать как некоторые условия, а переходы как некоторые события, происходящие в системе. Разметка называется живучей, если каждый из переходов в системе может быть запущен бесконечное число раз. Когда достигается разметка, при которой ни один из переходов не может быть запущен, говорят, что сеть Петри "зависла". Если разметка живучая, то система не остановится.

На рис. 7.9 выполнено моделирование взаимного исключения между двумя процессами с помощью сети Петри.

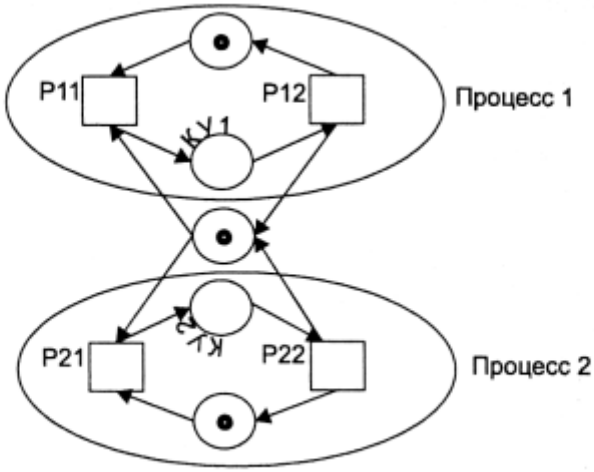

Рис. 7.9. Моделирование взаимного исключения с помощью сети Петри

Позиции, помеченные КУ1 и КУ2, представляют соответственно критические /частки первого и второго процесса. При текущей разметке могут быть запущены переходы P11 или P21. Если запускается переход рц, то позиция КУ1 получает пометку (система входит в критический участок). При этом второй процесс не может войти в КУ2, поскольку разделяемая позиция является пустой. Теперь может быть запущен только переход P12. После его запуска система возвращается в исходное состояние.

Вычислительные схемы

Сеть Петри не обеспечивает полной общности с вычислительной точки зрения. Например, сетью Петри нельзя смоделировать оператор отрицания ("не"). Существует еще одна структура под названием "вычислительная схема", которая эту проблему решает.

Вычислительная схема - представление в графической форме асинхронной системы, состоящей из набора операторов (процессов), воздействующих на множество регистров (ячеек памяти). Вычислительная схема поддерживает две структуры - граф потока данных и граф потока управления.

7.2.2.12. Планирование и диспетчеризация процессов

С точки зрения распределения процессорного времени операционные системы можно поделить на три группы.

- Системы с однопользовательским режимом (система запускает одну задачу, и ждет ее полного завершения).
- Системы с пакетным режимом (переключение процессора с одной задачи на другую происходит лишь в том случае, если активная задача сама отказывается от процессора).
- Системы с многозадачным режимом (переключение процессора происходит по истечении некоторого кванта времени).

Выделяют три уровня планирования.

- Планирование верхнего уровня - это планирование при поступлении в систему, планирование стадии "оформления" процесса и его допуска в систему.
- Планирование промежуточного уровня - это планирование при переводе процесса из очереди ожидающих ресурсы в очередь готовых к помещению на процессор;
- Планирование нижнего уровня (диспетчеризация) - это планирование очереди готовых к помещению на процессор процессов.

Можно сформулировать пять основных целей планирования.

- Справедливость планирования, заключающаяся в том, что надо относиться к процессам одинаково и не откладывать бесконечно их поступление на процессор.
- Завершение максимального количества процессов в единицу времени.
- Обеспечение приемлемого времени ответа максимальному числу пользователей.
- Предсказуемость планирования, определяемая тем, что одна и та же задача должна выполняться в системе за одно и то же время, независимо от условий.
- Постепенное снижение работоспособности системы.

Приоритеты

Приоритет - некоторое число, сопоставленное каждому процессу из очереди готовых процессов и обозначающее важность процесса. Приоритеты бывают:

- статистические (не меняющиеся с момента поступления процесса в систему) и динамические;
- присваиваемые автоматически (system defined) и назначаемые извне (user defined);
- купленные (например, в 70-х годах XX века в американских вычислительных центрах за назначение высокого приоритета платили деньги) и заслуженные;
- рациональные (назначенные из разумных соображений) и случайные (random).

Алгоритмы планирования

Перечислим основные алгоритмы планирования.

- Первый, пришедший в очередь процесс, обслуживается первым (First Comes First Served - FCFS). Процессы получают процессор в порядке их поступления в очередь готовых процессов. Получив процессор, они выполняются на нем до своего полного завершения.
- Циклическое (круговое) обслуживание (Round Robbin - RR). Каждый процесс находится на процессоре ограниченный квант времени, по истечении которого становится в конец очереди. Разновидность кругового обслуживания - круговорот со смещением, при котором квант времени зависит от внешнего приоритета процесса.
- Кратчайший процесс обслуживается первым (Shortest Job First - SJF). На процессор первым поступает процесс с минимальным оценочным временем исполнения. Процессы исполняются до полного завершения. Заметим, что в большинстве случаев невозможно предсказать оценочное время завершения. Возможным вариантом решения является сохранение истории и анализ косвенных признаков (число обращений к внешним устройствам).
- Первым обслуживается процесс с наименьшим остаточным временем (Shortest Rest Time - SRT). Это случай дополнения предыдущего алгоритма квантованием времени.
- Многоуровневая очередь с обратными связями. Можно сказать, что данный алгоритм использует прошлое, чтобы предсказать будущее. Вначале каждый процесс попадает в очередь с одинаковым приоритетом. Если процесс провел весь положенный ему квант времени на процессоре, то он переходит в очередь с меньшим приоритетом. Если процесс не отработал весь квант времени, то он переходит в очередь с большим приоритетом. Высший приоритет получают те задачи, которым он, как правило, нужен (например, интерактивные). Если процесс занимает много времени, то он попадает в очередь с небольшим приоритетом.

7.2.3. Коммуникация процессов в сетях

7.2.3.1. Уровневые протоколы

Протокол - совокупность синтаксических и семантических правил, которые определяют поведение функциональных блоков при передаче данных.

Уровень - компонент иерархической структуры. Назначение уровневых протоколов таково:

- обеспечение логической декомпозиции сложной сети на меньшие (и более понятные) части и уровни;
- обеспечение симметрии в отношении функций, которые реализуются в каждом узле сети;
- обеспечение стандартного интерфейса между сетевыми функциями.

Уровень является поставщиком сервиса и может состоять из нескольких сервисных функций. Каждый уровень преобразует полученную информацию. Верхний уровень обеспечен полным набором услуг, предлагаемых всеми нижними уровнями.

Семиуровневая модель протоколов Взаимосвязи Открытых Систем

Сначала рассмотрим семиуровневую модель протоколов Взаимосвязи Открытых Систем - ВОС (Open Systems Interconnections - OSI), разработанную институтом ISO. Семь уровней этой модели и их функции таковы:

1. Физический уровень. Отвечает за передачу неструктурированного потока данных по физической среде (например, коаксиальному кабелю, витой паре или оптоволокну).
2. Канальный уровень. Обеспечивает прием и передачу пакетов, определение аппаратных адресов.
3. Сетевой уровень. Выполняет маршрутизацию и ведение учета.
4. Транспортный уровень. Обеспечивает корректную сквозную передачу данных.
5. Сеансовый уровень. Осуществляет проверку полномочий (аутентификацию).
6. Уровень представления. Выполняет интерпретацию данных (например, определение кодировки).
7. Пользовательский уровень (уровень приложения). Предоставляет услуги конечному пользователю.

Существует реализация набора протоколов - GOSIP (Government OSI Profile), которой пользовались только правительственные организации США. Уже после того, как эта модель была разработана, выяснилось, что [Немет, Снайдер, Сибасс, Хейн 1999]:

- эти протоколы разрабатывались на основе концепций, практически не имеющих смысла в современных сетях;
- спецификации их были в ряде случаев неполны;
- по своим функциональным возможностям они уступали реально существующим протоколам;
- из-за наличия большого количества уровней они были медлительны и трудны для реализации. Некоторые иронически считают, что к уже существующим семи уровням можно добавить еще два: финансовый и политический.

Считается, что данная семиуровневая модель стала одной из самых неудачных разработок комитетов по стандартизации.

Протоколы TCP/IP

В 1969 году появился стек протоколов TCP/IP (Transfer Control Protocol/Internet Protocol). Он содержит всего четыре уровня. Можно провести приблизительные параллели между OSI и TCP/IP.

- Уровень сетевого протокола (эквивалентен 1 и 2 уровням в модели ВОС) Ethernet, Token ring.
- Уровень Интернета (эквивалентен 3-му уровню в модели ВОС) IP.
- Транспортный уровень (эквивалентен 4-му уровню в модели ВОС) TCP, UDP.

- Уровень приложений (эквивалентен 5, 6 и 7-му уровням в модели ВОС) FTP, SMTP, telnet, nfs - перечень файлов, электронная почта, терминал.

Приведем названия основных протоколов стека TCP/IP (рис. 7.10). Подробно все эти протоколы замечательно описаны в книге Стивенса [Stevens 1994].

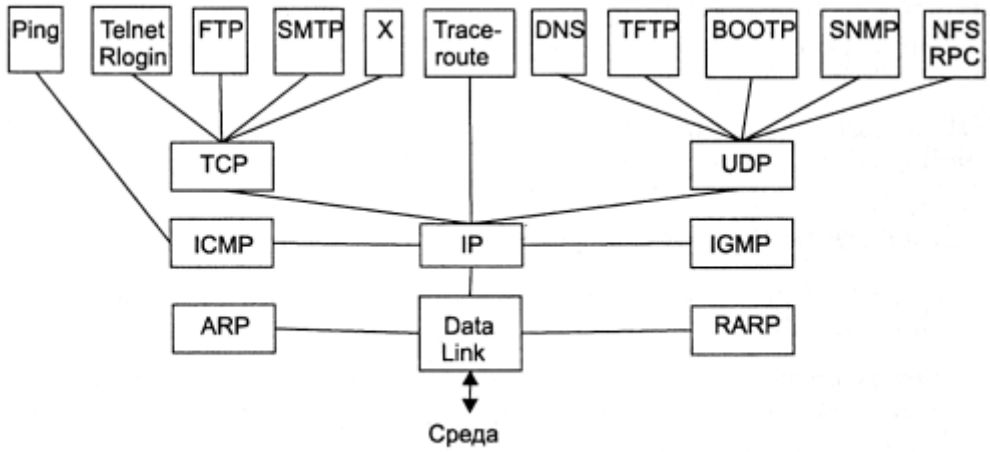

Рис. 7.10. Основные протоколы стека TCP/IP

Обмен данными между коммуникационными узлами

Она оседлала его и приготовилась выполнить необходимое соединение портов: штыревая и розеточная части подготовлены, ввод-вывод разрешен, режимы клиент-сервер, ведущий-ведомый.
Р. Дулинг. "Мозговой штурм"

Для того чтобы однозначно передать сообщение, отправитель должен знать:

- способ соединения. Для стека протоколов TCP/IP - это протокол, определяемый транспортным уровнем;
- адрес узла-адресата. Для стека протоколов TCP/IP - это IP-адрес;
- номер порта. Прикладной процесс, предоставляющий некоторые услуги другим прикладным процессам, ожидает поступления сообщений в порт, специально выделенный для этих услуг.

Данные пользователя, проходя по стеку протоколов при отправлении, дополняются заголовками на каждом уровне (рис. 7.11).

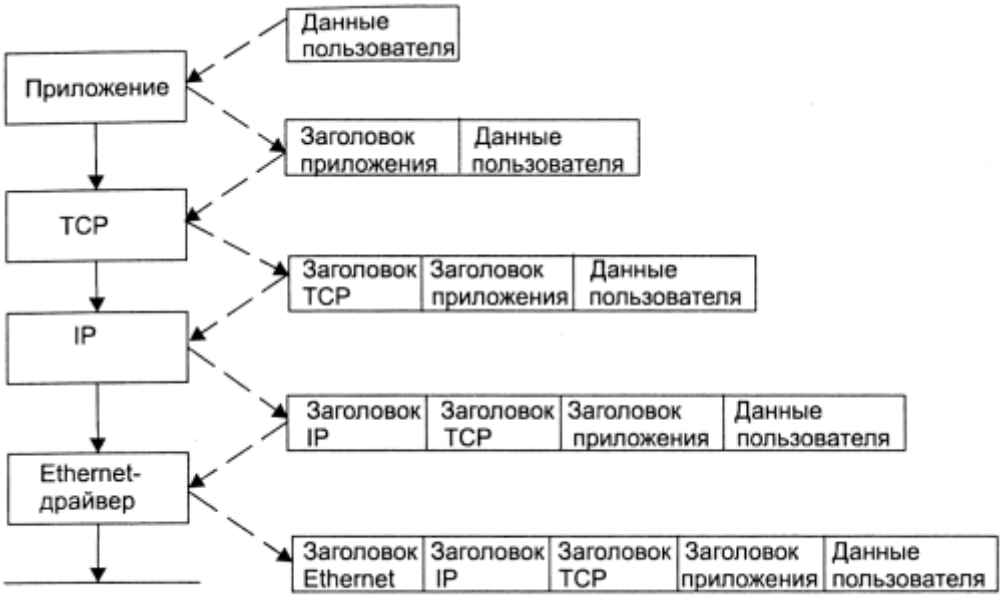

Рис. 7.11. Обмен данными в соответствии с протоколами стека TCP/IP

7.2.3.2. Адресация в сетях TCP/IP

Протокол TCP/IP использует следующую схему адресации [Немет, Снайдер, Сибасс, Хейн 1999]:

- нижний уровень адресации задается сетевыми аппаратными средствами. Сетевым картам Ethernet при их изготовлении задаются уникальные 6-байтные аппаратные адреса;
- следующий уровень использует IP-адресацию. Каждому включенному в сеть узлу присваивается уникальный четырехбайтный IP-адрес. Этот адрес глобален, уникален и не зависит от аппаратных средств. Основное назначение этих адресов в содействии маршрутизации пакетов из одной физической сети в другую. Можно задать соответствие между IP-адресами и аппаратными адресами, реализуемое на канальном уровне модели TCP/IP. Преобразованием адресов занимаются протоколы ARP (Address Resolution Protocol) и RARP (Reverse Address Resolution Protocol);
- верхний уровень адресации использует номера портов, служащие для определения процесса, которому адресованы данные и выполняющегося на данном узле.

IP-адресация

IP-адрес - уникальный адрес, идентифицирующий узлы или компьютеры в сети, управляемой протоколами TCP/IP.

Он состоит из следующих частей:

- сетевой части, обозначающей логическую сеть, к которой относится адрес. На основе этой части принимается решение о маршрутизации;
- машинной части, характеризующей конкретную машину в сети.

IP-адреса могут быть записаны как десятичные числа, разделенные точками. Существует несколько классов IP-адресов. Их отличие - в распределении байтов между сетевой и машинной частью. Класс адреса можно определить по его первому байту:

- 1 - 126 крупная сеть (класс A);
- 128 - 191 большие сети с подсетями (класс B);
- 192 - 223 сети не более чем из 254 компьютеров (класс C).

Значения первого байта 0, 127 и 255 являются специальными и не используются для обычных IP-адресов. Если первый байт находится в диапазоне 224-239, то это групповая адресация, а если в диапазоне 240-254, то это экспериментальные адреса.

> ***О сетевой и машинной частях***
> **Необходимость различать класс адреса (выделяя сетевую и машинную часть) требуется для того, чтобы была возможность осуществить маршрутизацию. Подробно вопросы маршрутизации, в том числе бесклассовой, будут рассмотрены в разд. 7.2.3.4.**

Общее число сетевых адресов классов A, B и C едва превышает два миллиона. Существующий Интернет-протокол IPv4 с его 32-разрядной схемой адресации не может предоставить достаточное число адресов для всех возможных участников сетевого обмена. Беспрецедентный рост Интернета приводит к необходимости перехода на новый протокол IPv6 [Алексеев, Захарова, Русаков 2000]. Новый протокол имеет 128-разрядные адреса, что теоретически позволяет адресовать 240 триллионов узлов.

Номера портов

Номера портов служат для определения процесса, который выполняется на данном узле и которому адресованы данные. В табл. 7.3 приведены некоторые зарезервированные номера портов для широко известных приложений.

Таблица 7.3. Некоторые зарезервированные номера портов

Номер	Имя и описание сервиса, использующего данный порт
7	ECHO - пересылка пакетов обратно отправителю
20	FTP - File Transfer Protocol (Default Data) - передача данных по протоколу передачи файлов
21	FTP - File Transfer Protocol (Control) - управляющие команды протокола передачи файлов
23	TELNET - удаленный доступ
25	SMTP - Simple Mail Transfer Protocol - электронная почта
53	DNS - Domain Name Server - сервер доменных имен
79	FINGER - возвращение информации об одном или нескольких пользователях на указанном компьютере
111	RPC - Remote Procedure Call - удаленный вызов процедур
115	SFTP - Simple File Transfer Protocol - простой протокол передачи файлов
123	NTP - Network Time Protocol - протокол синхронизации времени
161	SNMP- Simple Network Management Protocol- простой протокол управления сетью

7.2.3.3. Транспортные протоколы

В стеке протоколов TCP/IP существуют два основных транспортных протокола - TCP и UDP (User Datagram Protocol).

Протокол UDP

Протокол UDP обеспечивает пакетную передачу данных между источником и получателем без предварительного установления связи (т. е. сообщения, обрабатываемые протоколом, не имеют друг к другу никакого отношения с точки зрения UDP). Для доставки сообщений используется протокол IP. Надежность в протоколе UDP отсутствует и должна быть ее поддержка на уровне приложений. Данный протокол требует меньше накладных расходов, чем TCP. Обычно UDP используют такие протоколы верхнего уровня, как DNS и NTP.

Протокол TCP

Протокол TCP поддерживает надежную передачу потока данных с предварительной установкой связи между источником информации и ее получателем. Он используется такими протоколами верхнего уровня, как TELNET и FTP.

Отличительные черты TCP таковы:

- перед фактической передачей данных выполняется установка связи, т. е. запрос на начало передачи данных источником и подтверждение о готовности их принять получателем;
- доставка информации является надежной и не допускающей дублирования или изменения очередности получения данных;
- возможна доставка экстренных данных.

За счет усложнения транспортного уровня можно упростить уровень приложения. TCP поддерживает двунаправленный поток данных. Управление потоком данных осуществляется с помощью метода "скользящего окна". Начальная фаза сеанса носит название "тройного рукопожатия". Одна из сторон посылает пакет, вторая - подтверждает прием, тогда первая сторона посылает еще один пакет.

7.2.3.4. Маршрутизация в сетях TCP/IP

Маршрутизация - выбор маршрута следования информации от источника к получателю через объединенную коммуникационную сеть.

С точки зрения графов маршрутизацию можно рассматривать как ориентированный граф, нагруженный по дугам. Маршрут - путь, по которому движется сообщение (набор дуг). Оптимальный маршрут - маршрут, для которого сумма весов всех дуг, его составляющих, минимальна. Для решения задач нахождения оптимального маршрута созданы алгоритмы маршрутизации.

Данные маршрутизации хранятся в одной из таблиц ядра. Каждая строка этой таблицы содержит несколько параметров, например:

- IP-адрес сети назначения;
- IP-адрес следующего узла (обычно называемого шлюзом или маршрутизатором), через который нужно посылать пакеты, чтобы достигнуть сети назначения;
- стоимость пути до назначения.

Протокол маршрутизации используется демонами маршрутизации для обмена информации о сети, создается на базе алгоритма маршрутизации. Задача протокола - следить за правильной работой алгоритма.

Метрика - применяемая система измерения весов связей, которая в реальной работе может выступать как длина маршрута, время задержки, пропускная способность.

Маршрутизатор - устройство, единственной функцией которого является распределение сетевого трафика или потока сообщений. Получая пакет, маршрутизатор направляет его либо по назначению, либо обратно, если узел недостижим.

Классификация алгоритмов маршрутизации

Существуют различные варианты разделения алгоритмов маршрутизации по категориям.

Например, алгоритмы могут быть:

- внутреннего или внешнего шлюза:
  - алгоритмы внутреннего шлюза (внутридоменные) управляют данными маршрутизации в пределах логических групп узлов, называемых доменами, или автономными системами, или областями;
  - алгоритмы внешнего шлюза (междоменные) отвечают за маршрутизацию между автономными системами. Природа этих двух типов алгоритмов различная. Поэтому понятно, что оптимальный алгоритм для внутри-доменной маршрутизации не обязательно будет оптимальным для междоменной;
- статическими или динамическими:
  - при применении статических (неадаптивных) алгоритмов выбор маршрутов осуществляется заранее и заносится вручную в таблицу маршрутизации, где хранится информация о том, в каком направлении, какому соседу отправить пакет, следующий к определенному узлу назначения. Протоколы, разработанные на базе статических алгоритмов, называют также немаршрутизируемыми протоколами;

    - в случае динамических алгоритмов таблица маршрутизации меняется автоматически при изменении топологии сети или трафика в ней. Динамические алгоритмы отличаются по способу получения информации о состоянии сети, времени изменения маршрутов и по используемым показателям оценки маршрута;
- одномаршрутными или многомаршрутными:
    - одномаршрутные алгоритмы определяют один маршрут движения информации к получателю, и не всегда этот маршрут оказывается оптимальным. Такие алгоритмы отличаются критерием оценки маршрута;
    - многомаршрутные алгоритмы вырабатывают множество маршрутов к одному и тому же получателю. Они делают возможной мультиплексную передачу трафика по многочисленным каналам связи, при этом обеспечивая значительно большую пропускную способность и надежность сети;
- одноуровневыми или иерархическими:
    - в одноуровневой сети нет различия между разными ее частями. Все сегменты сети находятся на одном логическом уровне. Иерархическая сеть, как правило, состоит из двух частей. Маршрутизаторы нижнего уровня обычно служат для связи с определенными конкретными частями сети, ее сегментами. Маршрутизаторы верхнего уровня образуют особую часть сети, называемую магистральной (опорной). Маршрутизаторы магистральной сети передают пакеты между сетями нижнего уровня;
    - иерархическая структура больших и сложных сетей способствует упрощению управления сетью, облегчает изоляцию сегментов сети и т. д. Кроме того, преимуществом иерархической маршрутизации является то, что она имитирует организацию предприятия и, как следствие, хорошо поддерживает его схему трафика;
- централизованными, распределенными или локализованными:
    - в централизованных алгоритмах маршрутизации выбор всех маршрутов осуществляется в центральном узле. Вся информация, касающаяся топологических изменений, передается в этот специальный узел, называемый центром управления сети. Центр хранит в памяти маршрутные таблицы всей системы и обновляет их после получения новой информации о топологических изменениях. После этого он приступает к передаче информации, необходимой для выполнения локальных операций по маршрутизации, всем узлам, которым эта информация адресована;
    - в распределенных алгоритмах вычисление маршрутов распределяется среди узлов сети, если необходимо, при помощи обмена информацией. В каждом узле что-то вычисляется, и результаты передаются в другие узлы;
    - в локализованном алгоритме каждому узлу надо знать о текущей топологии сети и вычислять маршруты, ко всем возможным узлам назначения, основываясь на этой информации. Для того чтобы располагать самой свежей информацией, все узлы сети передают сообщения об их связности соседним узлам и так по цепочке;
- одноадресными или групповыми:
    - одноадресные алгоритмы маршрутизации строят один или несколько маршрутов к одному получателю;
    - многоадресные алгоритмы способны осуществить передачу информации многим получателям одновременно.

Когда маршрутизатор получает пакет, он выясняет пункт его назначения и определяет, по какому маршруту его отправить. Обычно маршрутизаторы хранят данные о нескольких возможных маршрутах. Решение о маршрутизации зависит от нескольких факторов, в том числе от:

- применяемой системы измерения длины маршрута, или метрики маршрута;
- основного алгоритма, используемого протоколом;
- топологии сети.

Классификация протоколов маршрутизации

Существует три основные группы протоколов маршрутизации. Деление на группы определяется типом реализуемого алгоритма определения оптимального маршрута.

К этим группам относятся:

- векторные протоколы (известные также как алгоритмы Бэллмана-Форда) самые простые и самые распространенные. Свое название этот тип протокола получил от способа обмена информацией. Маршрутизатор с определенной периодичностью копирует адреса получателей и метрику из своей таблицы маршрутизации и помещает эту информацию в рассылаемые соседям сообщения об обновлении. Соседние маршрутизаторы сверяют полученные данные со своими собственными таблицами маршрутизации и вносят необходимые изменения. После этого они сами рассылают сообщения об обновлении. Таким образом, каждый маршрутизатор владеет информацией о маршрутах во всей сети. При очевидной простоте алгоритма говорить о полной его надежности нельзя. Он может работать наиболее эффективно только в небольших сетях. Это связано с тем, что крупные сети не могут функционировать без периодического обмена сообщениями для описания сетевой топологии, и, к сожалению, большинство из них избыточны. Как следствие, возникают проблемы при выходе каналов связи из строя из-за того, что несуществующие маршруты могут оставаться в таблице маршрутизации в течение длительного времени. Трафик, направленный по такому маршруту, не достигнет своего адресата. В данную группу входит протокол RIP (Routing Information Protocol);
- протоколы состояния связей (канала) впервые предложены в 1970 году Дейкстрой. Они значительно сложнее, чем векторные протоколы. Вместо рассылки соседям содержимого своих таблиц маршрутизации, каждый маршрутизатор осуществляет широковещательную рассылку списка маршрутизаторов, с которыми он имеет непосредственную связь, и списка напрямую подключенных к нему локальных сетей. Эта информация является частью данных о состоянии связей и рассылается в специальных сообщениях. Кроме того, маршрутизатор рассылает сообщения о состоянии связей только в случае изменения информации о них или по истечении заданного интервала времени. Протоколы состояния связей трудны в реализации и нуждаются в значительном объеме памяти для хранения информации о состоянии каналов. Примерами этих протоколов являются OSPF (Open Shortest Path First) и IS-IS (Intermediate System to Intermediate System);
- к третьей группе протоколов относятся протоколы политики (правил) маршрутизации. Они наиболее эффективно решают задачу доставки получателю информации по разрешенным путям. Эта категория протоколов используется в Интернете и позволяет операторам получать информацию о маршрутизации от соседних операторов на основании специальных критериев. Алгоритмы, используемые для политики маршрутизации, опираются на алгоритмы длины вектора, но информация о показателях путей базируется на списке операторов магистрали. Примерами протоколов данной категории могут служить BGP (Border Gateway Protocol) и EGP (Exterior Gateway Protocol).

Протокол вектора расстояний

Протокол вектора расстояний (Routing Information Protocol - RIP) относится к векторным протоколам. Основное преимущество алгоритма вектора расстояний - его простота. Действительно, в процессе работы маршрутизатор общается только с соседями, периодически обмениваясь с ними копиями своих таблиц маршрутизации. Получив информацию о возможных маршрутах от всех соседних узлов, маршрутизатор выбирает путь с наименьшей стоимостью и вносит его в свою таблицу.

Протокол вектора расстояний поддерживает только самые лучшие маршруты к пункту назначения. Если новая информация обеспечивает лучший маршрут, то она заменяет

старую маршрутную информацию. Когда имеют место изменения в топологии сети, то они отражаются в сообщениях о корректировке маршрутизации. Например, когда какой-нибудь роутер обнаруживает отказ одного из каналов или другого роутера, он повторно вычисляет свои маршруты и отправляет сообщения о корректировке маршрутизации.

Достоинство этого элегантного алгоритма - быстрая реакция на хорошие новости (появление в сети нового маршрутизатора), а недостаток - очень медленная реакция на плохие известия (исчезновение одного из соседей).

За этот недостаток отвечает проблема, получившая название проблемы возрастания до бесконечности (count-to-infinity problem). Даже если узел узнал об исчезновении соседа и установил путь до него в таблице маршрутизации равным бесконечности, он тут же будет обманут другим своим соседом, который не знает о таком исчезновении и имеет конечный путь до того узла. Эта проблема является основной причиной того, что протокол вектора расстояний считает маршрут длиной более чем в 15 транзитных узлов бесконечным. Такой путь будет немедленно удален из таблицы маршрутизации.

Для предотвращения образования ложных маршрутов предлагаются несколько методов:

- метод расщепления горизонта заключается в том, что если известно, что путь до узла X лежит через соседний узел Y, то узлу Y не надо посылать объявления маршрута до X. Однако даже при минимальном усложнении топологии правило расщепления горизонта перестает действовать;
- правило отказа от приема запрещает маршрутизатору, получившему сообщение об отказе узла, принимать объявления маршрута до этого узла в течение некоторого времени;
- правило принудительных объявлений заключается в том, что, узнав об изменении метрики маршрута, маршрутизатор обязан немедленно сообщить об этом соседям;
- метод корректировки отмены маршрута ("отравленного маршрута" - route-poisoning) расценивает значительно выросшую стоимость маршрута как признак образования петли. Такой маршрут удаляется из таблицы маршрутизации. Какое изменение стоимости маршрута понимать как "значительное", зависит от администратора.

Протокол вектора расстояний использует механизм принудительных объявлений, а также функцию временного отказа от приема сообщений для обеспечения большей стабильности работы в условиях изменяющейся топологии. Цена за такую стабильность - увеличение времени определения новых маршрутов, т. к., заблокировав изменение некоторого маршрута вследствие отказа какого-либо узла из опасения "дезинформации" со стороны соседей, маршрутизатор отбрасывает и корректные объявления. Многие реализации протоколов позволяют отключить функцию отказа от приема сообщений. В этом случае, из-за распространения ложной информации петли будут возникать чаще, но эффективность работы сети может и повыситься.

Протокол выбора кратчайшего пути

В упрощенной форме принципы работы маршрутизаторов в соответствии с протоколом выбора кратчайшего пути (Open Shortest Path First Protocol - OSPF) можно сформулировать в виде пяти несложных правил. Итак, каждый маршрутизатор в сети должен:

- узнать всю информацию о топологии сети, измерить метрики каналов, соединяющих собственные физические интерфейсы с соседями, и далее, с помощью алгоритма Дейкстры вычислить кратчайшие пути ко всем остальным узлам и внести полученные результаты в таблицу маршрутизации;
- узнать стоимость пути до каждого из соседей (т. е. узнать о состоянии каналов). Для этого он рассылает через все свои физические интерфейсы специальные пакеты с приветствием. Получив такой пакет, соседний узел должен ответить, сообщив данные о себе. Узнав данные о соседях, маршрутизатор принимается за второй пункт данной программы - тестирование каналов связи с целью выяснения

метрики каждого канала. Под метрикой может пониматься пропускная способность, время задержки, надежность (количество ошибок на единицу переданной информации), загрузка канала. Задержку канала можно определить, послав специальный пакет, который принимающая сторона должна немедленно отправить обратно. Разделив время отклика пополам, маршрутизатор вычисляет приблизительную величину задержки канала;
- подготовить пакет-объявление, содержащий полученную информацию;
- разослать этот пакет всем соседям. В процессе работы маршрутизатора пакеты с объявлениями маршрутов рассылаются обычно лишь в случае каких-либо изменений в сети. В остальное же время протоколы состояния канала молчат и не загружают каналы служебной информацией, лишь изредка обмениваясь небольшими пакетами с приветствием. Обмен информацией осуществляется с помощью веерной рассылки, т. е., получив пакет, маршрутизатор сохраняет копию в своей базе данных и посылает пакет дальше всем остальным соседям;
- построить дерево кратчайших расстояний до всех остальных маршрутизаторов ("карту" сети). Для этого создается ориентированный граф, отражающий топологию сети. Имея его в памяти, маршрутизатор применяет алгоритм Дейкстры для выбора пути с наименьшей суммарной стоимостью до каждой из вершин графа (т.е. до каждого узла сети). По результатам этих вычислений и строится таблица маршрутизации, используемая далее при переключении трафика.

Узким местом такого подхода является необходимость обязательной синхронизации баз данных всех маршрутизаторов в пределах автономной системы. Если разные узлы будут по-разному представлять себе топологию сети, с которой они работают, то это приведет к образованию петель и к другим проблемам.

Бесклассовая междоменная маршрутизация

Маршрут может относиться к полному IP-адресу, т. е. всего к одной машине. Однако обычно маршрутизация осуществляется на уровне сетей, с использованием лишь части адреса пункта назначения. В прошлом межсетевая маршрутизация выполнялась на основе классов адресов, а подсети использовались только в пределах внутреннего шлюза. Это приводило к необходимости отдельного маршрута для сетей класса C, из-за чего пропускная способность базовой магистрали падала.

Схема бесклассовой доменной маршрутизации (Classless Inter-Domain Routing - CIDR) предлагает определять общие маршруты, которые обслуживали бы несколько сетей одновременно [Немет, Снайдер, Сибасс, Хейн 1999]. CIDR скрывает младшие биты номера сети и тем самым группирует несколько сетей в один маршрут. Эту схему, сформулированную в 1993 году, называют также схемой организации подсетей.

7.2.3.5. Формирование сети

Процесс формирования сети состоит из нескольких этапов, таких как:

- определение физической и логической топологии сети;
- установка сетевых аппаратных средств;
- назначение IP-адресов узлам сети;
- настройка всех основных машин на конфигурирование сетевых интерфейсов;
- настройка демонов маршрутизации, определение статических маршрутов.

Сетевая информационная служба (Network Information Services - NIS) - распределенная база данных для хранения информации. Она появилась в ответ на проблему необходимости наличия единого информационного центра, поднимавшуюся системными администраторами. Выделенный узел хранит большое количество информации, в том числе:

- о текущем количестве узлов;
- о параметрах пользователей;

- о группах пользователей;
- о временной зоне.

Может существовать резервный сервер, с которым основной периодически обменивается информацией.

7.2.3.6. Средства коммуникации высокого уровня

Средства коммуникации высокого уровня изолируют приложение от специфики сетевого взаимодействия, включающего создание коммуникационных узлов, установление и завершение связи.

Удаленный вызов процедур

Удаленный вызов процедур (Remove Procedure Call - RPC) основан на том, что процесс, исполняющийся на одном компьютере, запускает процесс на удаленном компьютере. Фактически осуществляется вызов процедуры, которая реально находится и поддерживается на другом компьютере. Удаленный вызов внешне очень похож на локальный. Естественно, существенно отличаются действия по передаче параметров и получению результата - все выполняется с помощью передач информационных пакетов по сети. Более подробно эти действия выглядят так [Робачевский 1997]:

- программа-клиент производит локальный вызов процедуры, называемой "заглушкой", при этом клиенту кажется, что, -вызывая "заглушку", он на самом деле вызывает процедуру сервера. В действительности, задача "заглушки" - принять аргументы, адресуемые вызываемой процедуре, преобразовать их в некоторый формат и сформировать сетевой запрос;
- сетевой запрос пересылается по сети на удаленную систему, при этом, например, используется стек протоколов TCP/IP;
- на сервере все происходит в обратном порядке: "заглушка" сервера ожидает запрос и при его получении извлекает из него параметры;
- "заглушка" сервера выполняет вызов настоящей процедуры (которой адресован запрос клиента), передавая ей нужные параметры;
- после выполнения процедуры при передаче результатов ее работы управление опять возвращается в "заглушку" сервера, которая формирует сообщение-отклик;
- отклик передается клиенту;
- отклик принимается "заглушкой" клиента, которая извлекает необходимые данные и передает их программе. Процесс завершен.

Фактически механизм заглушек составляет ядро системы удаленного вызова процедур. Удаленный вызов процедур передает только данные по значению (это серьезное ограничение). Более сложные среды распределенного программирования лишены подобного ограничения (например, среда CORBA). Прежде чем клиент сможет вызвать удаленную процедуру, необходимо связать его с удаленной системой, которая располагает требуемым сервером. Задача связывания состоит из нахождения удаленного узла с требуемым сервером и необходимого серверного процесса на данном узле. Укажем ряд проблем, связанных с данным средством:

- проблемы семантики вызова (например, невозможность установить, когда будет выполнена процедура и выполнится ли она вообще);
- проблемы представления данных (например, различное представление символов и вещественных чисел в различных архитектурах).

Коммуникации в группах

Основные проблемы групповой коммуникации заключаются в необходимости упорядочивания и синхронизации поступления сообщений. Даже небольшие задержки по времени в доступе до дальних узлов могут привести к нарушению порядка поступления сообщений на ряде узлов. Одним из удачных средств коммуникаций в

группах является система ISIS. Это надстройка над Unix, содержащая ряд коммуникационных примитивов. В ней реализована совокупность алгоритмов, базирующаяся на идее векторов из n компонентов (по количеству процессов), управление которыми осуществляется системой. В начале работы каждый процесс получает пустой вектор. 1-й компонент вектора является номером последнего сообщения, полученного в текущей последовательности от процесса под номером i. Рассмотрим, как эти векторы используются на практике (рис. 7.12). Пусть процесс P1 отправил в момент времени t1 сообщение всем другим процессам. Процесс P2, получив сообщение от P1 в момент времени t2, сам посылает остальным процессам сообщение в момент времени t3. Может оказаться так, что процесс P3 получит сообщение от процесса P2 в момент времени t4, т. е. раньше, чем от P1. Третий процесс анализирует вектор, пришедший от второго процесса (1, 1, 0), сравнивая со своим текущим (0, 0, 0). Обнаружив факт неполучения сообщения от первого процесса, он откладывает прием сообщения от второго процесса. После получения сообщения от процесса P1 в момент времени t5, процесс P3 примет отложенное сообщение в момент времени t6.

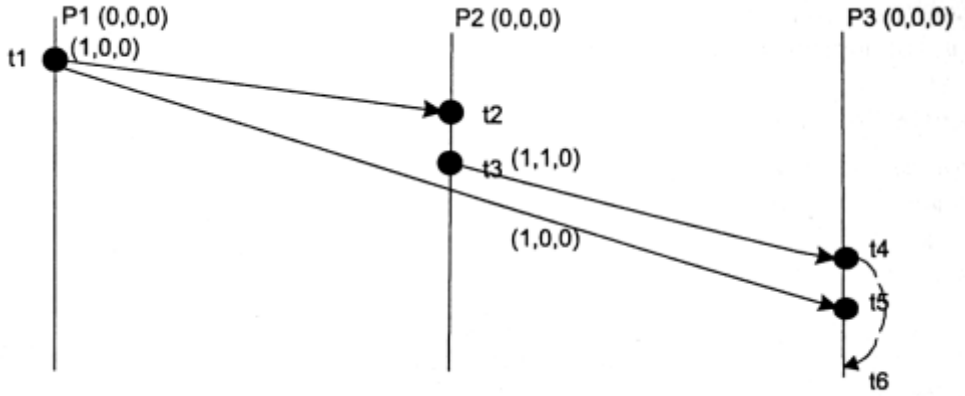

Рис. 7.12. Коммуникации в группах

7.2.4. Синхронизация процессов в распределенных системах

7.2.4.1. Основные подходы к синхронизации

В традиционных архитектурах синхронизация выполнялась через общий для всех процессов участок. Процессы на разных узлах борются за критический участок. Синхронизация в распределенных системах использует децентрализованные алгоритмы и становится гораздо более сложной, чем в централизованных.

Такие алгоритмы обладают, как правило, следующими свойствами:

- относящаяся к делу информация распределена между множеством компьютеров;
- процессы принимают решения только на основе локальной информации;
- не должно быть единой критической точки, выход из которой приводил бы к краху алгоритма;
- не существует общих часов или другого источника точного глобального времени.

Первые три свойства говорят о том, что недопустимо собирать всю необходимую информацию для принятия решения в одном месте.

Алгоритм Лампорта

Алгоритм был предложен Л. Лампортом (L. Lamport) в 1978 году. Основная идея заключалась в том, что абсолютной синхронизации не требуется, т. е. если два процесса не взаимодействуют, то единого времени и не требуется [Tanenbaum 1995].

Процессам достаточно иметь согласованное (логическое) время. Лампорт вводит специальный предикат "произошло до" - A->B означает, что "событие A произошло до события в".

В двух случаях это соотношение очевидно.

- Если оба события произошли в одном процессе.
- Если событие A есть операция посылки сообщения одним процессом, а событие в есть прием этого сообщения другим процессом.

Если два события случились в разных процессах, которые не обмениваются сообщениями, то соотношения A->B, B->A являются неверными, а события A и в - одновременными. Введем логическое время c, которое будем определять таким образом:

```
если A->B, то C(A) < C(B).
```

Часы ci увеличивают свое значение с каждым событием в процессе pi следующим образом: ci = ci + d, где d > o и обычно d = 1. Если событие a есть посылка сообщения m процессом PI, тогда в это сообщение вписывается временная метка tm = ci(a), и в момент получения этого сообщения процессом PJ его время корректируется таким образом: cj = max(cj, tm+d). Можно после ci указывать номер процесса. Недостатки логического времени проявляются в системах реального времени.

Выборы координатора

Очень многие алгоритмы в распределенных системах требуют наличия координатора - одного из процессов, который должен выполнять специальные функции по инициации, координации или решения других задач.

В качестве координатора выбирается процесс с максимальным номером. Можно применять разные алгоритмы выборов, но если процедура началась, то она должна закончиться согласием всех процессов о выборе нового координатора.

Алгоритм хулигана (задиры)

Один из процессов, обнаруживший отсутствие координатора, инициирует выборы. Алгоритм выбора выглядит так:

- процесс посылает свое сообщение всем процессам с номером большим, чем у него - "а не избрать ли нам координатора?";
- процессы с большими номерами отвечают инициатору, и его участие в выборах заканчивается. Эти процессы посылают сообщения "а не избрать ли нам координатора?" процессам с еще большими номерами;
- в конце алгоритма останется один процесс, который не дождется ответа. Он и объявляет себя координатором.

Круговой алгоритм

Круговой алгоритм использует физическое или логическое кольцо без маркера. При этом каждый процесс знает следующего за ним в круговом списке. Когда какой-либо из процессов обнаруживает отсутствие координатора, он посылает следующему за ним сообщение "а не избрать ли нам координатора?" и указывает свой номер. Если следующий не отвечает, то сообщение посылается процессу, идущему за ним. Каждый процесс добавляет в список свой номер. Рано или поздно список вернется процессу, инициировавшему выборы. Он посылает новое сообщение, в котором извещает о том, что он стал новым координатором, и указывает свой номер.

7.2.4.2. Взаимные исключения в распределенных системах

Централизованный алгоритм

Все процессы запрашивают у координатора разрешения на вход в критическую секцию и ждут разрешения. Координатор обслуживает запросы в порядке их поступления. Получив разрешение координатора, процесс входит в критическую секцию. Завершив работу, процесс докладывает об этом координатору.

Алгоритм с круговым маркером

Все процессы образуют логическое кольцо, при этом каждый знает, кто следует за ним. По кольцу циркулирует маркер, дающий право на критическую секцию. Получив маркер, процесс либо входит в критическую секцию, держа маркер у себя, либо переправляет маркер дальше, если ему не нужна критическая секция. Дважды подряд входить в критическую секцию, не отдавая маркер, нельзя.

Децентрализованный алгоритм на основе временных меток

Алгоритм требует глобального упорядочивания всех событий в системе по времени.

Когда процесс желает войти в критическую секцию, он посылает всем остальным процессам сообщение с именем критической секции, своим номером и текущим временем. После этого процесс ждет разрешения от всех процессов. Только после этого можно входить в критическую секцию.

Получив сообщение-запрос, процесс, в зависимости от своего состояния к указанной критической секции, действует одним из следующих способов:

- если получатель не находится внутри данной критической секции и не запрашивал разрешения на вход в нее, то он возвращает сообщение "разрешаю";
- если получатель находится в критической секции, то он не отвечает на запрос, но запоминает его;
- если получатель уже выдал запрос на вхождение в критическую секцию, но еще не вошел в нее (не дождался разрешения), то он сравнивает временные метки. Если у пришедшего запроса метка более ранняя, то он возвращает сообщение "разрешаю".

После выхода из критической секции процесс посылает сообщение "разрешаю" всем процессам, запросы от которых он запомнил, а затем все запомненные запросы стирает.

7.2.4.3. Высокоуровневые средства синхронизации

Атомарные транзакции

Атомарные транзакции - высокоуровневые средства синхронизации, представляющие логические единицы работы. Логическая единица работы, как правило, является согласованием ряда операций. Система, поддерживающая процесс транзакции, гарантирует, что если во время выполнения некоторых операций произошла ошибка, то все обновления будут аннулированы. В результате транзакция либо полностью выполняется, либо полностью отменяется. Транзакции пришли в программирование из области деловых отношений, когда переговоры могут вернуться к своему начальному состоянию в любой момент до заключения соглашения.

Транзакции обеспечиваются администратором транзакций с помощью следующих операторов:

- BEGIN_TRANSACTION - объявляет начало транзакций;
- END_TRANSACTION - объявляет успешное завершение транзакции и утверждает ее;
- ABORT_TRANSACTION - указывает на неудачное завершение транзакции, в результате чего все изменения должны быть отменены, и восстанавливает начальные значения;

- READ - служит для чтения данных;
- WRITE - служит для записи данных.

Существуют два основных метода поддержки реализации транзакций.

- Метод собственного пространства транзакции. Все действия ведутся в собственном "частном" пространстве транзакции. Изменения попадут в "реальное" пространство только в случае успешного завершения транзакции.
- Метод протоколирования списка действий. Все действия протоколируются в специальном списке, называемом "лист намерений". При изменении значения каждой переменной в список заносится имя переменной, а также ее начальное и конечное значения. Если транзакция завершается неудачно, то исходные значения переменных на начало транзакции легко восстановить по этому списку.

Транзакции обладают четырьмя важными свойствами [Дейт 2000].

- Атомарность. С точки зрения окружающего мира транзакция выглядит единой и неделимой. Выполняется либо все, либо ничего.
- Постоянство (согласованность). Если в системе есть некоторые инварианты, то если они существовали до выполнения транзакции, то они должны остаться и после ее выполнения. Одно согласованное состояние переводится в другое. Обязательной поддержки согласованности в промежуточных точках нет.
- Сериализация (изоляция). Транзакции отделены одна от другой и преобразуются в последовательную форму. Если существует несколько транзакций, выполняющихся параллельно, то они должны выглядеть так, как если бы они выполнялись последовательно.
- Прочность (долговечность). Если транзакция завершена, то ее результаты надежно вписываются в состояние системы. Даже если в следующий момент произойдет сбой системы.

Двухфазный протокол утверждения

Двухфазный протокол утверждения важен в тех случаях, когда транзакция может взаимодействовать с несколькими независимыми администраторами ресурсов. Каждый из них руководит своим собственным набором восстанавливаемых ресурсов и поддерживает собственный файл регистрации. Для транзакции не имеет смысл выполнять оператор завершения отдельно для каждого из администраторов ресурсов. Вместо этого должна быть выполнена единая общесистемная команда ENDJTRANSACTION или ABORTJTRANSACTION. Процесс, запустивший транзакцию, объявляется координатором. Остальные процессы - координируемыми. Координатор обеспечивает выполнение единой команды благодаря двухфазному протоколу утверждения.

Рассмотрим пример работы координатора. Пусть транзакция успешно завершена и ее нужно утвердить. Координатор осуществляет следующий двухфазный процесс:

- координатор записывает в протокол "транзакция готова к утверждению". Координируемые процессы, получив сообщение, записывают в свой протокол "готов к транзакции" и отсылают координатору сообщение "готов". Координатор в конце первой фазы собирает все сообщения о готовности;
- когда все ответы собраны, координатор записывает в протокол "транзакция утверждена" и рассылает всем процессам сообщение "транзакция утверждена". Координируемые процессы, получив сообщение, записывают в свой протокол "транзакция утверждена" и отсылают координатору "готов". Если все ответы во время первой фазы не удалось собрать, или хотя бы один из ответов, содержит отказ от транзакции, то координатор дает команду ABORTJTRANSACTION.

7.2.4.4. Тупики в распределенных системах

Тупики в распределенных системах подобны тупикам в централизованных системах, только их еще сложнее обнаруживать и предотвращать. Иногда выделяют особый вид тупиков в распределенных системах - коммуникационные тупики. Стратегии, применяемые при борьбе с тупиками в распределенных системах, таковы:

- полное игнорирование проблемы ("алгоритм страуса"). Этот подход в распределенных системах так же популярен, как и в централизованных;
- обнаружение тупиков. Для этого существует два основных метода:
    - централизованное обнаружение тупиков заключается в распространении идеи графа размещения ресурсов на распределенную систему. Координатор строит единый граф для всех узлов системы и на нем выполняет редукцию (приведение);
    - распределенное обнаружение тупиков. В этом методе инициирование задачи обнаружения тупика начинает процесс, который подозревает, что система находится в тупике. Он посылает тем процессам, которые держат нужный ему ресурс, сообщение из трех информационных полей. Первое поле содержит номер процесса, инициировавшего поиск, и оно не меняется при дальнейшей пересылке. Второе и третье поля содержат соответственно номер процесса, который зависит от ресурса, и номер процесса, который держит этот ресурс. Если, в конце концов, сообщение вернется к тому процессу, который инициировал поиск, то он приходит к выводу, что система зациклена и он должен предпринять действия по ликвидации тупика;
- предотвращение тупиков в распределенных системах базируется на идее упорядочивания процессов по их временным меткам. Допустимой будет ситуация, когда "старший" процесс ждет ресурсы, которые захватил "младший" процесс. В обратном случае "младший" процесс должен отказаться от ожидания.

Тупики в распределенных системах являются той' областью исследования, где наука и практика резко расходятся с самого начала, и многие алгоритмы никогда не будут воплощены на практике.

7.2.4.5. Распределение процессоров в распределенных системах и планирование

Модели организации процессоров в распределенных системах

Процессоры в распределенных системах могут быть организованы следующим образом:

- модель рабочих станций. Это модель системы, состоящей из рабочих станций, объединенных в локальную сеть;
- модель процессорного пула. Данная модель состоит из массива процессоров (процессорного пула) и X-терминалов;
- гибридная модель, соединяющая в себе особенности двух предыдущих.

Модель рабочих станций может включать рабочие станции нескольких категорий.

- Бездисковые. И для загрузки операционной системы, и для дальнейшей работы такая рабочая станция обращается к серверу. Особенности этой категории в низкой стоимости и достаточно легком сопровождении.
- С диском, используемым для хранения временных файлов и области подкачки страниц.
- С диском, используемым для хранения временных файлов, области подкачки страниц и системных файлов.
- С диском, используемым для хранения временных файлов, области подкачки страниц, системных файлов и кэшированных файлов (локальных копий для редактирования).
- С полноценным жестким диском.

Для работы с удаленным узлом может использоваться примитив remote a.out - запуск на простаивающем узле задачи a.out. При разработке алгоритмов работы в такой модели

следует продумать решение трех вопросов.

- Как определить простаивающую рабочую станцию?
- Как сообщить системе, что данная рабочая станция теперь не простаивает?
- Если на занятой нашим заданием удаленной рабочей станции начал работать пользователь, то что следует делать?

Как правило, для решения данных вопросов в модели рабочей станции применяется идея координатора, организующего работу процессоров.

- Простаивающая рабочая станция должна сйобщить координатору о том, что она не занята. Координатор поддерживает у себя информацию о незанятых рабочих станциях.
- Процесс, желающий воспользоваться простаивающей рабочей станцией, обращается к координатору.
- Если у координатора есть свободные рабочие станции, то он посылает заинтересованному процессу информацию о такой станции.
- Процесс обращается к свободной рабочей станции с просьбой выделить ему ресурсы. На каждой рабочей станции есть менеджер, управляющий распределением ресурсов.
- Менеджер рабочей станции обращается к координатору с просьбой удалить данную рабочую станцию из списка свободных.
- Процесс выполняет установку окружения на удаленной рабочей станции.
- Процесс запускает работу на удаленной рабочей станции.
- Процесс выполняет работу на удаленной рабочей станции.
- Процесс завершает работу на удаленной рабочей станции.
- Процесс посылает координатору сообщение о завершении им работы на удаленной станции.

В случае модели процессорного пула основной задачей является выстраивание процессов в очередь к этому пулу. Пусть x - количество задач, генерируемых в секунду пользователями, а y - количество задач, которое процессорный пул способен обработать в секунду. Л. Клейнрок (L. Kleinrock) доказал, что среднее время между возникновением запроса и получением ответа на него может быть оценено как T = 1/(y - x). Пусть есть n процессорных пулов. Тогда T' = 1/(yn - xn) = T/n. Данная формула показывает, что замена n небольших ресурсов одним большим, в n раз более мощным, позволит сократить время ответа в n раз.

Распределение процессов по процессорам

Наиболее интересной и распространенной моделью является модель рабочих станций. Далее будем считать, что все рабочие станции имеют одинаковую архитектуру. Однако они могут различаться как по быстродействию, так и по количеству памяти. Различают два основных класса стратегий распределения.

- Немигрирующие стратегии, в которых однажды помещенный на рабочую станцию процесс, больше не меняет это место.
- Мигрирующие стратегии, в которых передача процесса с одной рабочей станции на другую возможна несколько раз.

На чем может основываться идея распределения процессов? В ее основе может лежать:

- минимизация времени простоя процессоров;
- минимизация времени исполнения процессов.

Алгоритмы распределения могут быть сведены к пяти основным проектным решениям.

- Детерминированные (четко определенные) или эвристические алгоритмы. В случае детерминированных алгоритмов система должна вести себя все время одинаково,

т. е. все о процессах должно быть известно заранее.
- Централизованная или распределенная реализация алгоритма. Централизованные алгоритмы учитывают присутствие координатора, с помощью которого легче принимать решения.
- Оптимальные или приемлемые алгоритмы. В оптимальных алгоритмах все построения и вычисления выполняются с огромной точностью, не при этом тратится большое количество времени.
- Локальная или глобальная политика переноса процессов.
- Управление распределением процессов инициируется источником (т. е. рабочей станцией, на которой выполняется много процессов) или простаивающей рабочей станцией.

Реально существующие распределенные системы обычно используют эвристические распределенные приемлемые алгоритмы.

Некоторые реализационные особенности алгоритмов

Если имеется переполненная процессами рабочая станция, то часть процессов необходимо перенаправить на другие рабочие станции. Проблема заключается в том, чтобы выбрать кандидата. Существуют три основных подхода к его поиску.

- Узел, на который будет переведен процесс, определяется случайным образом. Далее возможны два варианта: либо новая рабочая станция слабо загружена и процесс начинает работу на ней, либо станция загружена сильно и тогда продолжается поиск нового узла. Для того чтобы количество пересылок не стало слишком большим, обычно вводят ограничения на их количество.
- Новый узел также выбирается случайным образом, но вместо пересылки туда процесса высылается только запрос. Если ответ отрицательный, то продолжаем искать незагруженный узел. Ограничение на число пересылок есть и в этом случае.
- Одновременно посылаются запросы ко всем узлам сети, а затем анализируются ответы. В результате анализа можно выбрать минимально загруженный узел.

Примеры алгоритмов распределения

Рассмотрим три различных алгоритма, демонстрирующих, как следует выделять процессоры в распределенных системах [Tanenbaum 1995].

- Детерминированный алгоритм на графах. Этот алгоритм в первую очередь используется в системах, где заранее известно число процессоров, объем памяти на каждой рабочей станции и величина ее загрузки. В качестве количественной характеристики берется связь процесса с другими процессами. Причем она учитывается только в том случае, если процессы находятся на разных рабочих станциях. Согласно этому алгоритму следует переносить процессы с одной рабочей станции на другую, чтобы нагрузку уменьшить. -
- Централизованный алгоритм. На одном из узлов для каждой рабочей станции ведется анализ ее загруженности. В простейшем случае для каждого узла строится график. Если система не загружена, то график опускается вниз. При загрузке системы график поднимается вверх, причем движение осуществляется скачками по числу работающих в системе процессов.
- Экономический алгоритм. Компьютерная система рассматривается с точки зрения экономических моделей. Некоторый координатор собирает о рабочих станциях данные, включающие информацию о процессоре, памяти и других характеристиках. В результате определяется стоимость использования рабочих станций, выраженная некоторыми числами. Координатор принимает решение, опираясь на стоимость и некоторые дополнительные факторы, связанные с процессом (например, его цель).

В каждом узле возникает задача диспетчеризации, которая в подавляющем большинстве случаев решается локально. Некоторые проблемы могут возникнуть лишь

в том случае, если группа сильно взаимодействующих процессов попадает на разные рабочие станции.

7.3. Память

Если ты что-то записал в компьютерной памяти, запомни, где ты это записал.
Лео Бейзер

Иерархия памяти современных компьютеров с точки зрения архитектуры была рассмотрена в разд. 6.3.2. Здесь мы рассмотрим то, как операционная система работает с данным ресурсом.

7.3.1. Основная память

7.3.1.1. Привязка адресов

На рис. 7.13 изображен процесс отображения информации исходной программы в оперативную память.

Рис. 7.13. Привязка адресов

Больше всего нас будут интересовать две функции.

- Функция именования, которая однозначно отображает данное пользовательское имя в идентификатор информации, к которому относится имя. Эта функция реализуется обычно редактором связей.
- Функция памяти, которая однозначно отображает определение идентификатора в истинные номера ячеек памяти, в которых он будет находиться. Эта функция реализуется частью операционной системы, называемой загрузчиком.

Функция именования

Функцию именования реализует редактор связей - профамма, объединяющая объектные модули в один модуль загрузки. Практически всегда редактор связей является частью операционной системы. Достаточно часто эта профамма входит в состав компилятора, позволяя эффективным образом объединять объектные модули

именно этого компилятора. Каждая операционная система имеет свои соглашения о связях, т. е. правила оформления структуры файла, который является результатом работы редактора связей.

Перечислим некоторые из проблем, возникающих перед редактором связей.

- Редакторы связей сталкиваются с проблемой установки соответствия между одноименными объектами в языках с блочной структурой.
- При трансляции отдельного модуля должна быть предусмотрена специальная таблица с информацией о глобальных объектах, которые используются в данном модуле, но не определены в нем.
- Редактор связей обычно не выполняет проверку типов и количества параметров процедур и функций. Если надо объединить объектные модули программ, написанные на языках со строгой типизацией, то необходимые проверки должны быть выполнены дополнительной утилитой перед запуском редактора связей.

Функция памяти

Функция памяти выделяет переместимому модулю загрузки реальные ячейки памяти.

Распределение памяти может быть:

- статическим, когда привязка к конкретным ячейкам памяти выполняется либо до, либо во время загрузки модуля;
- динамическим, когда задание может перемешаться в памяти, допуская настройку и получение абсолютных значений ячеек памяти при каждом обращении к ним.

Существуют следующие основные способы реализации отображения памяти:

- оверлеи (overlay) - возможность расположить модули в памяти таким образом, чтобы один из них (корневой) постоянно находился там, а остальные - попеременно загружались в ходе выполнения программы в одну и ту же область, сменяя и перекрывая друг друга;
- свопинг (swapping) - разрешение системе вводить в память и выводить из нее задания целиком;
- поблочное отображение - возможность группировать элементы информации в блоки (если блоки имеют одинаковый размер, то это страницы, если разный, то сегменты).

Как правило, все современные архитектуры имеют мощную аппаратную поддержку работы с блоками памяти. Это объясняется следующими факторами:

- во-первых, с помощью аппаратной поддержки легко убедиться, что мы пытаемся адресоваться именно в те участки, в которые разрешен доступ;
- во-вторых, имеются средства обеспечения быстрого доступа к нужному участку с помощью ассоциативной памяти, реализованной на высокоскоростном кэше (translation look-aside buffer - TLB). Во многих случаях эти средства играют решающую роль в обеспечении эффективности работы. Даже наличие 8- или 16-регистрового кэша позволяет поднять ее на 90%.

Обычно используется так называемая сегментно-страничная организация памяти, в которой сегменты разбиваются на страницы фиксируемой длины.

7.3.1.2. Управление виртуальной памятью

Виртуальная организация памяти - процесс расширения логической памяти за пределы физической.

Виртуальная память - техника, позволяющая исполнять процессы, которые могут находиться в памяти не полностью.

Виртуальная память реализована во многих операционных системах. Однако есть операционные системы, не поддерживающие ее по разным причинам:

- MS-DOS - по причине распространенности на старых процессорах со слабой аппаратной поддержкой;
- операционные системы на архитектуре Cray - с целью экономии времени.

Виртуальная память решает две основные задачи.

- Возможность одновременной работы с несколькими приложениями.
- Возможность отсутствия программы в основной памяти целиком.

Конкретный механизм виртуальной памяти зависит от того, как реализуются три основных стратегии.

- Стратегия размещения (placement policy) определяет, в какое место основной памяти будет помещена подкачиваемая страница или сегмент.
- Стратегия вталкивания (fetch policy) определяет, в какой момент времени страница или сегмент должна быть помещена в основную память.
- Стратегия вытеснения (replacement policy) определяет, какую из страниц или сегментов нужно удалить из основной памяти, чтобы поместить на ее место новую.

Стратегия размещения

В случае страничной организации памяти стратегия размещения реализуется тривиально, поскольку все участки памяти имеют один и тот же размер. В случае сегментной организации памяти выделяют два подхода.

- Ведение списка свободной памяти. Список можно организовать по возрастанию адресов или размера фрагментов. Известной стратегией ведения списков является метод близнецов в случае блоков размером 2 в некоторой степени.
- Сборка мусора, представляющая собой уплотнение памяти. Сборка мусора имеет достаточно много отрицательных моментов, например, при ней приходится приостанавливать выполнение текущих заданий.

Стратегия вталкивания

Существует две основные стратегии вталкивания (подкачек).

- Вталкивание по запросу, осуществляемое в тот момент, когда требуется отсутствующая страница.
- Вталкивание с опережением. Оно может быть применено, если имеется возможность предсказать поведение программы. Это может быть сделано в очень редких случаях (например, при работе с большим массивом, который обрабатывается последовательно).

В том случае, когда нужна отсутствующая в памяти страница, операционная система генерирует прерывание. Далее происходит обращение к менеджеру виртуальной памяти. Он считывает с жесткого диска нужную страницу, записывает ее в свободный участок памяти, корректирует таблицу страниц и дает процессору команду на повторное выполнение действия.

Стратегия вытеснения

Для определения вытесняемой из основной памяти страницы используются алгоритмы, приведенные в книге [Цикритизис, Бернстайн 1977].

- Принцип оптимальности (принцип Биледи). Следует вытеснять ту страницу, к которой не будет обращения в течение наиболее длительного времени. Этот принцип нереализуем на практике, но может быть использован как эталон при оценке других алгоритмов.
- Вытеснение случайно выбранной страницы. Недостаток метода в слепом везении, достоинство - в отсутствии дискриминации.
- Вытеснение первой загруженной страницы. Достоинство метода - в его легкой реализации, недостаток - в неэффективности при большой загрузке системы. Существует аномалия данного метода, когда определенные последовательности обращений к страницам приводят к увеличению количества прерываний по отсутствию страницы в памяти при увеличении количества свободных фреймов.
- Вытеснение дольше всего неиспользуемой страницы. Этот метод требует дополнительной аппаратной или программной поддержки для хранения последнего времени доступа.
- Вытеснение реже всего используемой страницы. Этот метод требует дополнительной аппаратной или программной поддержки для хранения числа обращений.
- Вытеснение не используемой в последнее время страницы. Поддержка заключается в хранении двух битов. Первый содержит признак обращения к странице, второй - признак модификации страницы. Первоначально все биты сбрасываются в ноль. Существует четыре комбинации битов, которые можно выстроить по приоритету. Сначала, например, удалять страницы с комбинацией 0-0, потом с 0-1, 1-0 и, наконец, 1-1.
- Принцип локальности. Локальность - свойство, заключающееся в том, что распределение запросов процессов на обращение к памяти имеет, как правило, неравномерный характер с высокой степенью локальной концентрации, как временной, так и пространственной.
    - Временная локальность заключается в том, что если к некоторым ячейкам памяти недавно были обращения, то с высокой вероятностью будут обращения к ним и в будущем.
    - Пространственная локальность заключается в том, что если к некоторым ячейкам памяти недавно были обращения, то с высокой вероятностью можно ожидать обращения и к ближайшим к ним ячейкам. Рабочее множество - набор страниц, к которым процесс активно обращается. Для того чтобы программа выполнялась эффективно, необходимо, чтобы ее рабочее множество находилось в основной памяти.

7.3.1.3. Распределенная общая память

С момента возникновения идеи распределенных вычислений явно предполагалось, что программы на машинах без физической общей памяти исполняются в разных адресных пространствах. Только в 1986 году К. Ли (K. Li) создал концепцию распределенной общей памяти - РОП (Distributed Shared Memory - DSM). Основная идея заключается в том, что набор рабочих станций в сети разделяет единое страничное виртуальное адресное пространство.

Достоинства систем с распределенной общей памятью в том, что в них могут исполняться программы, написанные для мультипроцессорных систем. Такие системы легко масштабируются. Недостатком является достаточно низкая производительность, связанная с накладными расходами на выполнение коммуникационных протоколов (trashing).

Путь оптимизации систем с распределенной общей памятью может заключаться в разделении только тех данных, которые требуются для записи, а данные для чтения - не разделяются (следовательно, эти страницы не перегоняются по сети).

Типы архитектур и соответствующие им структуры, поддерживающие распределенную общую память, приведены на рис. 7.14 [Tanenbaum 1995].

Рис. 7.14. Распределенная общая память

Алгоритмы реализации распределенной общей памяти

Существуют четыре основных алгоритма [Tanenbaum 1995].

- Алгоритм с центральным сервером. Все разделяемые данные поддерживает центральный сервер. Клиенты обращаются к нему с запросами. Если данные нужны для чтения, то они посылаются клиентам. Если для записи, то сервер их корректирует и посылает клиентам в ответ квитанции, подтверждающие модификацию.
- Миграционный алгоритм. Основное отличие данного алгоритма от предыдущего в том, что отсутствует централизованная поддержка данных сервером. Разделяемые данные пересылаются (мигрируют) на тот узел, который их запросил. Алгоритм предусматривает возможность задать время, в течение которого страница насильственно удерживается в узле для того, чтобы можно было выполнить несколько обращений к ней до миграции в другой узел.
- Алгоритм разделения для чтения. Разделяются концепции чтения и записи. Страницы для чтения могут храниться хоть на каждом узле. Страницы для записи должны управляться по предыдущему алгоритму. Производительность алгоритма повышается за счет возможности одновременного доступа по чтению. Запись данных требует больших затрат для уничтожения всех устаревших копий блока данных или их коррекции.
- Алгоритм полного размножения. Этот алгоритм - расширение предыдущего. Он реализует протокол многих читателей и многих писателей. Поскольку много узлов могут писать данные параллельно, требуется контролировать доступ к ним для поддержания согласованности.

Все перечисленные алгоритмы недостаточно эффективны. Для повышения эффективности следует менять не алгоритмы работы с памятью, а семантику (способ) работы с ней.

Модель консистентности (логичности)

Модель консистентности представляет собой некий договор между программами и памятью, в котором говорится, что при соблюдении программой определенных правил работы, содержимое модуля памяти будет корректно, а если требования к программе будут нарушены, то память не гарантирует правильность выполнения операций "чтение - запись". Рассмотрим основные типы консистентности.

- Строгая консистентностъ. Операция "чтение ячейки памяти с адресом x" должна возвращать значение, записанное самой последней операцией "запись" с адресом x. В системе со строгой консистентностыо должно присутствовать физическое (реальное) время, что нереально. Для большинства систем это нереализуемо.
- Последовательная консистентность. Впервые определена Лампортом в 1979 году. Результат выполнения должен быть тот же, как если бы инструкции операторов всех процессов выполнялись бы в некоторой последовательности, определяемой

программой для этого процессора. При параллельном выполнении все процессы должны видеть одну и ту же последовательность записей в память (разрешаются запаздывания для чтения).
- Причинная консистентность. Не требует, чтобы все процессы видели одну и ту же последовательность записей в память, проводя различие между потенциально-зависимыми (запись в одни и те же ячейки) и потенциально-независимыми (запись в различные ячейки) операциями записи.
- Водопроводная консистентность. Операции записи, выполняемые одним процессором, видны всем остальным процедурам в таком порядке, в котором они выполнялись. Операции записи, выполняемые разными процессорами, могут быть видны в произвольном порядке.
- Слабая консистентность. Эту консистентность определяют три правила:
    - доступ к синхронизационным переменным определяется моделью последовательной консистентности;
    - доступ к синхронизационным переменным запрещен, пока не выполнены все предыдущие операции записи;
    - доступ к данным по записи или чтению запрещен, пока не выполнены все предыдущие обращения к синхронизационным переменным для данного (локального) процессора.

7.3.2. Внешняя память

Единственной мерой времени является память.
Владислав Гжегорчик

7.3.2.1. Управление внешней памятью

В управлении внешней памятью выделяют две подзадачи.

- Управление файловой системой.
- Управление устройствами ввода-вывода, представляющими часть вычислительной системы, действия которой ориентированы на обмен информацией с центральным процессором.

Отметим, что схема управления файлами образует некоторую иерархию.

- Уровень системы управления базами данных. Это высший уровень иерархии.
- Методы доступа.
- Логическая система управления файлами. На этом уровне выполняются такие логические команды, как например - открыть файл.
- Базисная система управления файлами. Это уровень работы с сегментами данных.
- Система ввода-вывода. Например, прочитать физический блок памяти.
- Уровень аппаратуры. Это действия, ориентированные на устройства. Примером может быть команда нижнего уровня - перемотать ленту.

7.3.2.2. Файлы и файловые системы

Файл - поименованная упорядоченная совокупность данных. Файл обычно имеет следующий минимум атрибутов:

- имя файла. Это строка символов, поддерживающая информацию в виде, удобном для восприятия человеком;
- тип файла. Данная информация требуется многим утилитам и системам для первичной идентификации файла;
- местонахождение. Это информация об устройстве, на котором файл расположен, и полный путь к нему в рамках этого устройства;
- размер файла;
- информация о защите файла от чтения, записи и исполнения;
- информация о пользователе, создавшем файл, включая дату и время создания;

- информация о пользователе, который последним изменял файл, включая дату и время модификации.

Операции, выполняемые с файлами, достаточно очевидны:

- создание файла;
- запись в файл;
- чтение из файла;
- позиционирование в файле;
- удаление файла.
- удаление некоторой части файла, начиная с определенной позиции. Информация, связанная с открытым файлом, такова:
- описатель файла;
- количество открытий этого файла;
- информация о местоположении файла на диске. При этом фактически копия открытого файла (или его части) хранится в основной памяти.

Файл является частью следующей иерархии объединения данных:

- комбинация битов;
- байт;
- поле - группа взаимосвязанных байтов;
- запись - группа взаимосвязанных полей. Записи могут быть физические - блоки, и логические - рассматриваемые как единое целое с точки зрения пользователя;
- файл - группа взаимосвязанных записей.

Организация файлов - способ расположения записей файлов во внешней памяти. Различают пять основных способов организации.

- Последовательная, в которой записи следуют одна за другой. Примером может служить организация записей на магнитной ленте.
- Библиотечная, в которой блоки объединены в разделы.
- Прямая (произвольная), в которой доступ к записям осуществляется по их физическим адресам.
- Индексно-последовательная, в которой используются ключи для поиска физических записей.
- Виртуально-последовательная, в которой записи упорядочены по ключу и состоят из справочника (как правило, B-дерева) и области данных.

Метод доступа - сочетание типов файлов и стандартных программных средств, обеспечивающих определенный режим передачи данных между файлом и основной памятью. Существует два метода.

- Метод доступа с очередями. Применяется, когда последовательность обработки записей можно предвидеть (например, при последовательном доступе).
- Базисный метод доступа. Этот метод доступа применяют в том случае, когда последовательность обработки предвидеть нельзя.

7.3.2.3. Распределенные файловые системы

При создании распределенных файловых систем обращают внимание на две главные задачи [Tanenbaum 1995].

- Сетевая прозрачность, которая заключается в обеспечении тех же возможностей доступа к файлам, как и в централизованных системах.
- Высокая доступность, заключающаяся в том, что ошибки и системные сбои не должны приводить к проблемам доступа к файлам.

Файловый сервис - интерфейс с файловой системой, то, что предоставляет файловая система.

Файловый сервер - процесс, который представляет файловый сервис.

Пользователь не должен знать, сколько в системе файловых серверов и где они расположены. В системе могут функционировать разные процессы (с разных операционных систем), следовательно, файловый сервис должен уметь работать со всеми.

Архитектура распределенных файловых систем

Распределенная файловая система обычно имеет два существенно отличающихся компонента.

- Файловый сервис. Модели, на которых может основываться файловый сервис, таковы:
  - модель "загрузки-разгрузки". В этом случае осуществляется пересылка файла клиенту целиком;
  - модель удаленного доступа реализуется без пересылки файла клиенту;
- Сервис каталогов (директорий), обеспечивающий операции создания и удаления каталогов, именования файлов и их переименования и перемещения.

Ключевые решения, которые необходимо принять при разработке распределенных файловых систем, должны отвечать на вопросы.

- Должны или не должны все процессы видеть иерархию каталогов одинаково?
- Должен ли быть единый корневой каталог? Существуют две формы прозрачности именования.
- Прозрачность расположения, определяющая, как легко мы можем обратиться к файлу.
- Прозрачность миграции, когда изменение расположения файла не требует изменения имени.

Подходы к именованию файлов таковы:

- имя машины и путь к файлу на ней;
- монтирование удаленных файловых систем в локальную иерархию файлов;
- все процессы видят все файлы одинаково (для реализации нужен мощный механизм).

Большинство систем используют ту или иную форму двухуровневого именования: файлы имеют символическое имя, которое видит пользователь, и внутреннее двоичное'имя, которое используется самой системой.

Семантика разделения файлов такова:

- семантика однопроцессорного компьютера: если за операцией записи следует операция чтения, то результат определяется последней операцией записи;
- семантика сессий: изменения открытого файла видит только тот процесс, который эти изменения проводит, и лишь после закрытия файла изменения становятся доступными всем остальным процессам;
- семантика транзакций: монополизация доступа, после возврата - обновление.

Реализация распределенных файловых систем

Рассмотрим различные аспекты реализации распределенных файловых систем [Tanenbaum 1995].

- Использование файлов. Для разработчика главное - понять, как система будет использовать файлы. Вот некоторые данные статистики.
    - Большинство пользовательских файлов имеет размер менее 10 Кбайт. Средний размер файла равен 4 Кбайт. Вывод может быть таким, что следует осуществлять перекачку файлов целиком.
    - Чтение встречается чаще, чем запись. Вывод таков: следует активно применять кэширование.
    - Чтение и запись осуществляются последовательно, произвольный доступ очень редок. Вывод: можно вести упреждающее кэширование.
    - Большинство файлов имеет короткое время жизни. Вывод: временные файлы надо создавать на машине клиента.
    - Лишь небольшое количество файлов разделяется процессами. Вывод: кэширование надо делать на машине клиента и применять семантику сессий.
    - В системах существуют различные классы файлов с различными свойствами. Вывод: для разных классов файлов в системе могут поддерживаться различные механизмы.
- Структура системы. Для ее определения должны быть получены' ответы на три вопроса.
    - Есть ли разница между клиентом и сервером? Здесь есть два варианта: либо любая машина может предоставить файловый сервис, либо есть специализированные средства сервера.
    - Следует ли объединять файловый сервер и сервер каталогов? Разумным может быть следующее разделение: можно иметь различные серверы каталогов (например, для операционных систем Unix и DOS) и один файловый сервер.
    - Должен ли сервер хранить у себя информацию о клиентах? В зависимости от принятого решения в системе будут либо серверы с состоянием, либо без состояния.
- Кэширование. В клиент-серверной схеме с памятью и диском имеются четыре места для хранения файлов или их частей.
    - Диск сервера. Данное место удобно тем, что здесь легко поддержать консистентность, и нет размножения файлов, но при этом теряется эффективность, и возникают проблемы передачи данных.
    - Память сервера. При этом возникает дополнительный вопрос - помещать файлы в память целиком или частями.
    - Диск клиента. Это место не дает преимуществ перед первыми двумя, но при этом в несколько раз усложняется работа системы.
    - Память клиента. Это место является наиболее разумным для хранения файлов или их частей.

Оценить способ и обосновать выбор можно лишь при учете характера приложений и данных о быстродействии процессоров, памяти и сети.

ЛИТЕРАТУРА

Не существует ответов, есть лишь ссылки.
Библиотечный закон Винера

1. Алексеев, Захарова, Русаков 2000 - Алексеев И. В., Захарова М. Н., Русаков А. И. Технологии IPv6 - Интернет нового поколения в мире и России. // Информационное общество, № 2, 2000.

2. Баурн 1986- Баурн С. Операционная система Unix. - М.: Мир, 1986.

3. Блэк1990- БлэкЮ. Сети ЭВМ: протоколы, стандарты, интерфейсы.- М.: Мир, 1990.

4. Дейт 2000 - Дейт К. Дж. Введение в системы баз данных (6-е изд.). - К.; М.; СПб: Издательский дом "Вильяме", 2000.

5. Дейтел 1987 - Дейтел Г. Введение в операционные системы. - М.: Мир, 1987.

6. Дунаев 1995 - Дунаев С. Операционная система UNIX System V Release 4.2. Изд-во МИФИ - Диалог, 1995.

7. Дунаев 1998- Дунаев С. UNIX сервер. Том 1, 2. Изд-во МИФИ- Диалог, 1998.

8. Зегжда, Ивашко 1997- Зегжда Д. П., Ивашко А. М. Как построить защищенную информационную систему. - СПб.: Мир и семья-95, 1997.

9. Кейлннгерт 1985- Кейлингерт П. Элементы операционных систем. - М.: Мир, 1985.

10. Кейслер1986- Кейслер С. Проектирование операционных систем для малых ЭВМ. - М.: Мир, 1986.

11. Колик 1975' - Колин А. Введение в операционные системы. - М.: Мир, 1975.

12. Корнеев1999- Корнеев В. В. Параллельные вычислительные системы.- М.: "Нолидж", 1999.

13. Кулаков, Луцкий 1998 - Кулаков Ю. А., Луцкий Г. М. Компьютерные сети.-К: Юниор, 1998.

14. Немет, Снайдер, Сибасс, Хейн 1999 - Немет Э., Снайдер Г., Сибасс С., Хейн Т. UNIX: руководство системного администратора. - СПб.: BHV - Санкт-Петербург, 1999.

15. Николаев 1997 - Николаев Ю. И. Проектирование защищенных информационных технологий. - СПб.: Изд-во СПбГГУ, 1997.

16. Олифер, Олифер 2001 - Олифер В. Г., Олифер Н. А. Сетевые операционные системы. - СПб.: Питер, 2001.

17. Робачевский 1997 - Робачевский А. М. Операционная система UNIX. - СПб.: BHV - Санкт-Петербург, 1997.

18. Соломон, Руссинович 2001 - Соломон Д., Руссинович М. Внутреннее устройство Microsoft Windows 2000. Мастер-класс. - СПб.: Питер, М.: Издательско-торговый дом "Русская Редакция", 2001.

19. Хант 1997- Хант К. Персональные компьютеры в сетях TCP/IP. - К.: Издательская группа BHV, 1997.

20. Цикритизис, Бернстайн 1977 - Цикритизис Д., Бернстайн Ф. Операционные системы. - М.: Мир, 1977.

21. Lewine 1995- Donald A. Lewine. POSIX Programmer's Guide. O'Reilly & Associates, Inc. 1995.

22. Lewis, Berg 1995 - Bil Lewis, Daniel J. Berg. A Guide to Multithreaded Programming. Threads Primer. SunSoft Press. A Prentice Hall Title. 1995.

23. Salus 1994 - Peter H. Salus. A Quarter Century of UNIX. Addison-Wesley Publishing. 1994.

24. Silberschatz, Galvin 1995 - Silberschatz A., Galvin P. Operating system concepts. Addison-Wesley Publishing. 1995.

25. Stevens 1994 - W. Richard Stevens. TCP/IP Illustrated: the protocols. Addison-Wesley Publishing. 1994.

26. Tanenbaum 1995 - Andrew S. Tanenbaum. Distributed operating system. Prentice-Hall, Inc. 1995.

технологии программирования к оглавлению к высокоуровн. языкам - 3GL к визуальным средам - 4GL

Знаете ли Вы, что **класс, Class - Класс в программировании** - это множество объектов, которые обладают одинаковой структурой, поведением и отношением с объектами из других классов.