

A.Shaw **The Logical
Design
of Operating
Systems**

Computer Science Group
University of Washington

PRENTICE-HALL, INC.
ENGLEWOOD CLIFFS, N. J.
1974

A. Shoy Логическое проектирование операционных систем

Перевод с английского

B. B. Макарова и B. D. Никитина

под редакцией

Г. Н. Соловьева

651 914

ИЗДАТЕЛЬСТВО «МИР»

Москва 1981

ББК32.973

Ш81

УДК681.142.2

Шоу А.

Ш81 Логическое проектирование операционных систем: Пер. с англ.— М.: Мир, 1981.— 360 с., ил.

Монография по проектированию и разработке операционных систем, написанная американским специалистом. Материал ориентирован на подготовку специалистов по созданию математического обеспечения ЭВМ. Содержание книги близко известным монографиям по операционным системам С. Мэдника, Дж. Денисона (М.: Мир, 1978) и Д. Цикритзиса, Ф. Беристайна (М.: Мир, 1977) и удачно дополняет их.

Для специалистов по математическому обеспечению, аспирантов и студентов.

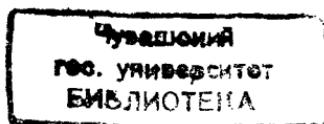
Ш 20204-032 32-81, ч.1
041(01)-81

2405000000

ББК 32. 973
6Ф 7.3

Редакция литературы по математическим наукам

516173



© 1974 by Prentice-Hall, Inc., Englewood Cliffs N. J.
© Перевод на русский язык, «Мир», 1981

Предисловие редактора перевода

В последнее время вниманию читателей был предложен целый ряд интересных переводных монографий, посвященных операционным системам (ОС). Каждая из этих монографий носит оригинальный характер, и это свидетельствует о многогранности проблемы. Следует отметить, что большинство авторов используют при изложении функциональный и прагматический подходы. Читатель из совокупности книг получает информацию, достаточную для изучения основных понятий и принципов, которым подчиняется поведение операционных систем. Вопросы же их проектирования до сих пор в литературе отражены недостаточно. Большая часть книги А. Шоу посвящена именно вопросам проектирования ОС. Поэтому выход в свет настоящей книги — явление желаемое и далеко не ординарное.

Книга выгодно отличается от ранее изданных как методом изложения, так и набором рассматриваемых вопросов. Систематичность изложения принципов проектирования достигается построением общей схемы ОС, основанной на концепциях процесса и ресурса. Это позволяет автору с должной степенью формализации и в то же время в доступной форме рассматривать вопросы проектирования, инвариантные по отношению к архитектуре и составу аппаратурных средств, т. е. рассматривать логический уровень проектирования. По существу в книге представлены и решены основные задачи разработки алгоритмического обеспечения и баз данных систем управления ресурсами, процессами и информацией. В этой связи читателю предлагается большое количество разнообразных алгоритмов, которые описаны на модифицированном АЛГОЛе, обладающем средствами описания параллельных процессов.

Особенно следует подчеркнуть значимость и глубину проработки, а также новизну для читателя таких принципиальных вопросов, как взаимодействие процессов и проблема тупиков. Так, глава 8, в которой представлены формализованные методы предотвращения, обнаружения и ликвидации тупиков, является примером теоретического и практического изучения некоторых сторон ОС и дает представление о возможной форме теории ОС.

Несомненным положительным качеством книги является ее учебный характер, что проявляется в хорошей методической

проработке, последовательности и простоте изложения, большом количестве контрольных вопросов и примеров, наличии обширной библиографии; весьма полезен пример мультипрограммного проекта для выполнения студентами.

Следует отметить некоторые особенности перевода. Во-первых, возникли определенные трудности при описании алгоритмов. Автор широко использует смысловую окраску при именовании переменных, процедур, параметров и т. д. При этом построение имен основывается либо на сокращении английских слов, либо на конкатенации. Введение в предложения АЛГОЛА русской нотации именованных объектов было признано неудобным и нецелесообразным, поэтому все алгоритмы было решено оставить на языке оригинала, а перевести лишь комментарии. Для удобства восприятия алгоритмов читателями перевод используемых в них словосочетаний вынесен в приложения. Из-за отсутствия устоявшейся терминологии, особенно в области управления процессами и ресурсами, были предложены некоторые новые термины. При переводе исправлены обнаруженные в тексте оригинала некоторые неточности.

Книга Шоу несомненно будет полезна специалистам, занятым разработкой и внедрением как программных, так и аппаратурных средств. Она может служить хорошим пособием для студентов и аспирантов при углубленном изучении операционных систем, а также будет полезна преподавателям при подготовке лекций и проведении практических и лабораторных занятий.

При переводе учтены замечания и рекомендации А. П. Гагарина, которые во многом улучшили этот перевод.

Г. Н. Соловьев

Предисловие

Операционные системы ЭВМ относятся к числу самых сложных «систем», созданных человеком, и только недавно мы смогли понять и по достоинству оценить эту сложность. Эта книга рассказывает о принципах операционных систем, причем особое внимание в ней уделено мультипрограммированию. Я попытался описать концепции и аппарат, требуемые для проектирования и понимания этих систем, а не обсуждать подробно, как некоторая операционная система *x* реализована на машине *y*; однако для иллюстрации применения частных принципов в книге приводится много примеров реальных систем. Заголовок «Логическое проектирование» подчеркивает мою заинтересованность в логической организации и взаимодействии элементов операционных систем, а также в методах их «обоснования».

Книга предназначена для студентов, изучающих вычислительную технику, и специалистов, владеющих основами знаний организации ЭВМ, языка ассемблера, языков программирования и структур данных. Необходимые знания могут быть получены после прослушивания вводного односеместрового курса по каждому из упомянутых выше предметов, приблизительно эквивалентного курсам В2, И1, И2 и И3 учебной программы, предлагаемой Ассоциацией по вычислительным машинам¹⁾ в Программе 68. При написании книги я использовал ее как основной текст в односеместровом курсе в Корнеллском и Вашингтонском университетах. Книга может быть использована в одно- или двухсеместровом курсе для студентов как младших, так и старших курсов, и содержит почти все темы, предложенные в курсе 14 Программы 68 и в более позднем докладе COSINE²⁾.

¹⁾ ACM Curriculum Committee on Computer Science. *Curriculum 68, recommendations for academic programs in computer science*. Comm. ACM, 11, 3 (March 1968), 151—197.

²⁾ COSINE Committee of the Commission on Education. An undergraduate course on operating system principles (Denning P. J., chairman). Commission on Education, National Academy of Engineering, Washington, D. C., 1971.

С моей точки зрения предмет операционных систем наиболее удобно разделять на три связанные области: управление процессами, управление ресурсами и файловые системы. Каждая из девяти глав этой книги касается некоторых аспектов одной или более из указанных областей. В главе 1 проводится обзор организации систем аппаратуры и программного обеспечения, включая историю вопроса. В главе 2 в качестве примера использована простая реализация системы пакетной обработки (по одному заданию во времени) для пояснения некоторых основных принципов работы связывающих загрузчиков и методов ввода-вывода. Модель взаимодействующих процессов как средство описания систем и как основа для решения проблем связи и синхронизации процессов (включая некоторые проблемы, введенные в главе 2) развита в главе 3. Глава 4 является введением в системы мультипрограммирования и построена на материале, разработанном в предыдущих главах; в ней обсуждаются требования к аппаратуре и программному обеспечению для мультипрограммирования, «виртуальные» машины, воспринимаемые пользователями и системными программистами, и методология проектирования. Методы управления реальной и виртуальной памятью исследованы в главе 5; в следующей главе (гл. 6) продолжается изучение ресурса основной памяти, обсуждаются проблемы разделения единственной копии информации в системах реальной и виртуальной памяти. Идеи управления ресурсами и процессами изложены в главе 7, где полное ядро системы рассматривается в качестве модели для изучения системных структур данных процессов ввода-вывода, управления прерываниями и методов распределения. В главе 8 подробно описываются системные тупики; методы определения, восстановления и предотвращения тупиков описаны как для последовательно используемых, так и для потребляемых ресурсов. В последней главе (гл. 9) рассматриваются основные элементы файловых систем, включая раздел о восстановлении из аварийных ситуаций.

Книга содержит много упражнений, которые настоятельно рекомендуется выполнить читателю. При изучении новых идей, связанных с системами ЭВМ, особенно важно, чтобы студентам была дана возможность практически реализовать эти идеи.

с помощью программирования проектов. Составить нетривиальные, но объяснимые проекты нелегко; по этой причине в книгу включено приложение, содержащее подробную спецификацию большого, но управляемого мультипрограммного проекта, который я с успехом использовал несколько раз на практике.

Я старался оснастить весь материал точными ссылками, чтобы читатель мог глубже изучить интересующую его область или ознакомиться с другой точкой зрения, и поэтому указал источники каждого метода или идеи. Я искренне сожалею о любых ошибках или пропусках в указании этих источников. Все источники приведены в конце книги; ссылки на них в тексте даются по фамилии автора, сопровождаемой датой, например: Дейкстра, 1965б.

Благодарности

Я очень признателен ряду сотрудников за их помощь, поддержку и одобрение во время подготовки этой рукописи. В. Ф. Миллер первый открыл для меня мир радостей и удовлетворения научными исследованиями и достижением знаний и дал мне поддержку в самом начале работы над книгой по операционным системам. Я имел счастье быть ассистентом Н. Вирта по его курсу системного программирования в Стенфордском университете в 1965/66 г. и сделал ряд заметок на основе его лекций¹⁾; эти заметки содержали ряд главных идей, на которых базируется проектирование и конструирование операционных систем и компиляторов. Я также признателен Вирту, указавшему мне, что системное проектирование и программирование могут быть научной дисциплиной.

Дж. Джордж и Дж. Хорнинг прочитали рукопись и внесли много конструктивных предложений. В основу книги были положены лабораторные работы курса операционных систем в Корнеллском и Вашингтонском университетах; я благодарен студентам, слушавшим этот курс, за их энергию, любознательность, чувство юмора и готовность помочь созданию новой научной области. Дж. Эндрюс, Р. Холт, Н. Вейдерман и Т. Уиллокс оказали особенно большую помощь не только в указанном

¹⁾ Shaw A. C., Lecture Notes on a Course in Systems Programming. Tech. Report No. 52, Computer Science Dept., Stanford, Calif. Dec. 1966.

выше качестве, но также как мои активные сотрудники. В частности, в главе 7 используются результаты докторской работы Вейдермана, а глава 8 основана на докторской Холта.

И наконец, я выражаю благодарность множеству исследователей и практиков, которые внесли вклад в развитие операционных систем. Наибольшее влияние на меня оказали опубликованные работы Э. В. Дейкстры, и в этой книге я повсюду отмечаю его вклад.

Алан К. Шоу
Сиэтл, шт. Вашингтон

1. Организация вычислительных систем

Термин *логическое проектирование* используется разработчиками ЭВМ для описания систематической методологии, которая основана на проектировании переключающих сетей с помощью аппарата булевой алгебры. В данной книге этот термин используется в более широком смысле для обозначения общего метода *рассмотрения* операционных систем, который делает возможным их систематическое проектирование, а также изучение их организации и поведения. Мы будем выделять общие принципы, а не какие-либо специальные приемы; таким образом, мы не будем рассматривать подробно методы программирования или изучать конкретную коммерческую систему, хотя многие примеры взяты из таких систем для иллюстрации отдельных положений.

Эта глава является вводной. В ней рассмотрено историческое развитие компонентов аппаратуры и программного обеспечения, дается краткое описание организации и функций вычислительных систем и обсуждаются системные программы о различных точек зрения. Прежде всего определим некоторую основную терминологию.

1.1. Некоторые определения

Такие выражения, как «операционная система», «разделение времени» и «мультипрограммирование», не имеют общепринятых точных определений, за исключением, возможно, случаев, когда они употребляются в контексте теоретического изучения систем в некотором узком аспекте. Вместе с тем эти термины обозначают определенные типы организаций, функций, поведения и методов работы. Имея это в виду, мы неформально определим некоторые важные термины, которые широко используются для описания систем.

Операционная (супервизорная, мониторная, исполнительная) *система* (ОС) есть *организованная* совокупность программ (систем), которая действует как интерфейс между аппаратурой ЭВМ и пользователями. Она обеспечивает пользователей набором средств для облегчения проектирования, программирования, отладки и сопровождения программ и в то же время управляет распределением ресурсов для обеспечения эффективной работы.

Существуют три категории «чистых» ОС, каждая из которых может быть охарактеризована определенным типом взаимодействия между пользователем и его заданием и ограничениями на время ответа системы.

1. *ОС пакетной обработки* является системой, в которой задания пользователей предоставляются на обработку в виде последовательных пакетов на входных устройствах и в которой не существует взаимодействия между пользователем и его заданием во время обработки. Пользователь полностью изолирован от своего задания и поэтому считает, что время ответа системы есть время полного выполнения задания. Это время обычно расценивается как удовлетворительное, если может быть измерено несколькими минутами или часами. Следовательно, в ОС может быть использована относительно гибкая дисциплина планирования.

2. *ОС разделения времени* — это система, которая обеспечивает одновременное обслуживание многих пользователей (работающих «в линию» с ЭВМ¹⁾), позволяя каждому пользователю взаимодействовать со своими вычислениями²⁾. Эффект одновременного доступа достигается разделением времени процессора и других ресурсов между несколькими пользователями таким способом, который гарантирует ответ на каждую команду пользователя в течение нескольких секунд. ЭВМ предоставляется каждому процессу пользователя в течение небольшого «кванта времени» обычно в миллисекундном диапазоне; если процесс не завершился к концу своего кванта, он прерывается и помещается в очередь ожидания, уступая другому процессу свою очередь на ЭВМ.

3. *ОС реального времени* — это система, которая обслуживает внешние процессы, развивающиеся на устройствах, работающих «в линию» с ЭВМ, имеющие жесткие ограничения на время ответа. Действиями системы управляют сигналы прерывания от внешних процессов; если они не будут быстро обработаны (в течение микросекунд, миллисекунд или секунд в зависимости от процесса), то ход внешнего процесса может исказиться. Эти системы часто проектируются для частного применения, например для управления технологическим процессом.

¹⁾ В оригинале „on-line-users”. Работа одного объекта с другим в режиме on-line будет далее называться работой «в линию», что предполагает параллельную работу этих объектов с взаимным обменом информацией по мере работы. — Прим. ред.

²⁾ В оригинале „computations”. Этот английский термин не имеет подходящего аналога в отечественной литературе. Здесь понимается в широком смысле как набор программ и данных в активном или пассивном состоянии. — Прим. перев.

Конкретная ОС может обеспечить выполнение как определенного вида задания (пакетной обработки, разделения времени или реального времени), так и всех видов заданий одновременно. Например, как системы разделения времени, так и системы реального времени обычно выполняют пакетные задания в фоновом режиме, когда нет работы «в линию».

Самым общим методом функционирования ОС является мультипрограммирование. *ОС с мультипрограммированием*, или мультипрограммная ОС (МС), — это система, которая обеспечивает одновременное нахождение в основной памяти нескольких программ пользователей, разделяя процессорное время, пространство памяти и другие ресурсы между активными пользовательскими заданиями. Это разделение ресурсов распространяется на операционную систему; программы, составляющие ОС, сами подвержены мультипрограммированию в большинстве крупных систем.

ОС может обеспечить одновременную обработку нескольких заданий другим способом — посредством *сплонга*¹⁾. Операционная система, использующая сплонг, в любое время обеспечивает одновременное нахождение нескольких заданий во вспомогательной памяти и только одного задания — в основной памяти; система переключается на другое задание посредством перемещения текущего задания из основной памяти во вспомогательную и загрузки выбранного задания из вспомогательной памяти в основную. Если предыдущее задание не выполнено до конца, оно будет позднее вновь возвращено в основную память. Этот метод до сих пор использовался главным образом в малых системах разделения времени.

Наше последнее определение, мультиобработка, относящееся к конфигурации аппаратуры системы, иногда путают с мультипрограммированием. *Вычислительная система с мультиобработкой* — это комплекс аппаратуры ЭВМ с несколькими независимыми обрабатывающими устройствами. Она включает в себя центральные процессоры, процессоры ввода-вывода, каналы данных и процессоры специального назначения, такие, как арифметические устройства. Наиболее часто этот термин относится к совокупности центральных обрабатывающих устройств.

В этой книге рассматриваются в первую очередь мультипрограммные операционные системы — преимущества и недостатки этой организации по сравнению с другими, а также методики и требования к разделению времени, пространства памяти и других ресурсов.

¹⁾ В оригинале „swapping”, Известен перевод этого термина как обмен, перезагрузка. — Прим. перев.

1.2. Нотация для алгоритмов

Язык программирования АЛГОЛ-60 (Наур, 1963) и его признанные варианты будут нашими основными средствами для записи алгоритмов.

Мы выбрали именно АЛГОЛ, а не английский язык, язык диаграмм, язык ассемблера или какой-нибудь другой язык более высокого уровня по следующим причинам:

1. Синтаксис и семантика АЛГОЛА ясно описаны в литературе, содержат мало неоднозначностей.

2. Язык успешно использовался в течение многих лет как международный язык для публикаций алгоритмов.

3. При желании, чтобы опустить многие второстепенные детали, алголоподобные описания могут быть сделаны достаточно «высокоуровневыми». С другой стороны, язык может быть использован «низкоуровневым» образом, так что тексты непосредственно отображаются в машинный язык¹⁾.

4. Автор, многие его студенты и коллеги считают, что АЛГОЛ является достаточно мощным средством для общего представления, выражения структуры и анализа алгоритмов. [Читатель может найти введение в АЛГОЛ-60 и копию первоначального доклада в книге Розена (1967, стр. 48—117).]

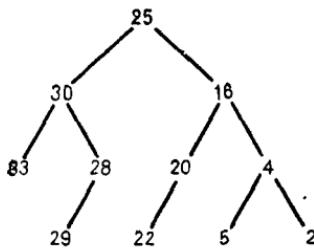
Пример алгольной процедуры

В операционных системах часто используются древовидные структуры данных, например, для представления иерархий процессов (гл. 7) или файловых справочников (гл. 9).

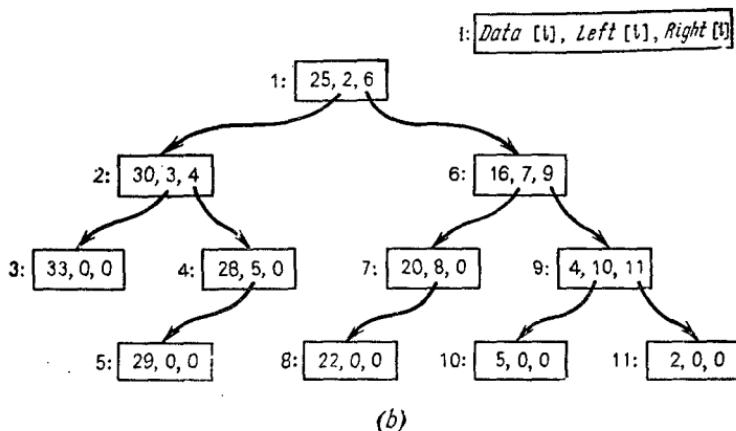
Двоичное дерево состоит из конечного набора вершин, который либо пустой, либо может быть разделен на корневую вершину и два отдельных двоичных дерева, а именно на левое и правое поддеревья (Кнут, 1968). Пусть каждая вершина n в двоичном дереве будет представлена триадой ($Data[n]$, $Left[n]$, $Right[n]$), где $Data[n]$ — целое положительное число, $Left[n]$ — неотрицательный указатель на корень левого поддерева и $Right[n]$ — неотрицательный указатель на правое поддерево. Зарезервируем значение указателя вершин $n = 0$ для пустого дерева. Положим, что для каждой вершины n

- (1) $Data[n] < Data[x]$ для всех $x \in$ левое поддерево(n)
- $Data[n] > Data[x]$ для всех $x \in$ правое поддерево(n)

¹⁾ Хотя мы разделяем общие принципы структурного программирования (см., например, Дейкстра, 1969 и SIGPLAN, 1972), читатель все же пайдет предложение `go to` в некоторых из наших программ. Предложения `go to` не заменены полностью другими конструкциями потому, что они полезны для наглядного описания действий на машинном уровне и для явного представления передачи управления, а также потому, что они могут использоваться по определенным правилам, гарантирующим получение хорошо структурированных программ,



(a)



(b)

Рис. 1.1. Двоичное дерево для быстрой сортировки, поиска и вставки:
а) — двоичное дерево; (б) — внутреннее представление.

На рис. 1.1 приведен пример такого дерева. [Указанным способом иногда организуются символьные таблицы, так как при этом они могут быть расширены и легко доступны для поиска (Грис, 1971).] Ниже представлена алгоритмическая процедура *Treesearch(root, arg, m)*, которая осуществляет поиск по дереву с корнем *root* такой вершины *n*, что *Data[n] = arg*; при успешном завершении поиска она возвращает значение *true* и присваивает *m* значение, равное номеру найденной вершины, в противном случае она возвращает значение *false*. Алгоритм непосредственно использует рекурсивное определение двоичного дерева.

Boolean procedure *Treesearch(root, arg, m);*

value *root, arg;* **integer** *root, arg, m;*

comment *Data [], Left [], Right []*

предполагаются глобальными для этой процедуры. Найти в дереве с корнем *root* такую вершину *n*, что *Data[n] = arg*:

```

if root = 0 then Treesearch := false else
begin
  integer d;
  d := Data[root];
  if arg = d then begin m := root; Treesearch := true end
  else
    if arg > d then Treesearch := Treesearch(Left[root], arg, m)
    else Treesearch := Treesearch(Right[root], arg, m)
end

```

Упражнение

Напишите алголиную процедуру *Addtotree*(*root*, *n*), которая берет двоичное дерево с корнем *root* и порядком расположения вершин, удовлетворяющим условию (1), и добавляет изолированную вершину *n* к нему, сохраняя это условие. Напишите еще одну процедуру, которая печатает данные о дереве в восходящей последовательности; используйте любую удобную базовую операцию (примитив), такую, как *Write*(*x*), в качестве обращения к выводу на печать переменной *x*.

1.3. Исторический аспект

В этом разделе кратко описывается эволюция аппаратуры и программного обеспечения ЭВМ. Более подробное рассмотрение этого предмета и список литературы можно найти в работах С. Розена (1969) и Р. Розина (1969).

1.3.1. Ранние системы

Начиная с 1949 г., когда заработала первая цифровая вычислительная машина с запоминаемой программой, и до 1956 г. устройство и способ действия вычислительных машин оставался относительно постоянным (с некоторыми перспективными, но большей частью неудачными исключениями). Их классическая фон неймановская архитектура была основана на строго последовательном выполнении команд, включая операции ввода-вывода. При загрузке и выполнении программ пользователи обычно работали за консолью непосредственно «в линию» с машиной, устанавливая значения регистров, «шагая» по командам, проверяя ячейки памяти и вообще взаимодействуя со своими вычислениями на самом низком машинном уровне. (Развитие систем разделения времени обусловлено признанием преимуществ работы в такой форме, но на более высоком уровне, чем «голая» аппаратура.) Программы писались на абсолютном ма-

шинном языке (десятичная и восьмеричная нотация) и при вводе им предшествовал абсолютный загрузчик¹).

Поучительно сделать обзор процедур для абсолютной загрузки, поскольку даже теперь с выполнения этих процедур начинается работа любой системы программного обеспечения на «голой» машине. Любая ЭВМ имеет эквивалент кнопки загрузки; нажатие этой кнопки оператором вызывает чтение вычислительной машиной записи входных данных в некоторый фиксированный набор смежных ячеек памяти и затем передачу управления (т. е. счетчик команд машины приравнивается фиксированному адресу в пределах этого набора, обычно первому).

Пример

Пусть основная память простейшей ЭВМ обозначена $M[0]$, $M[1]$, $M[2]$, ..., где каждая ячейка $M[i]$ может содержать один байт (8 битов) информации. Предположим, что нажатие кнопки «Загрузка» вызовет чтение 80-колонной карты с 80 байтами информации в $M[0]$, ..., $M[79]$, сопровождающееся «обнулением» счетчика команд, т. е.

```
PressLoad: Read(for i:=0 step 1 until 79 do M[i]);
Transferto(M[0]);
```

Для того чтобы считать абсолютную программу, первая карта (карта, считанная по *PressLoad*) должна содержать машинные команды для считывания последующих карт (или по крайней мере следующей карты). Пусть каждый адрес, команда и элемент данных в нашей простейшей машине занимает 1 байт. Предположим, что абсолютная программа отперфорирована на картах в следующем формате:

Колонки карты	Содержимое
1	Адрес загрузки LA для 1-го байта поля карты, обозначенного на рис. 1.2 как „программа/данные”.
2	Число байтов n , которое нужно загрузить; $n < 78$.
3 до $(n + 2)$	Часть поля карты, обозначенного на рис. 1.2 „программа/данные”, абсолютная программа.

¹) Абсолютный загрузчик находился в начале колоды перфокарт, составляющих программу (см. рис. 1.2). — Прим. перев.

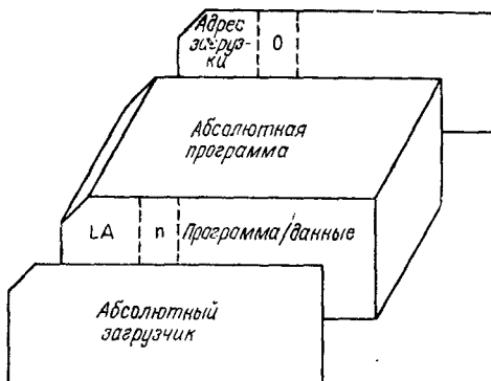


Рис. 1.2. Абсолютный загрузчик и карты программы.

Последняя карта содержит $n = 0$, а «адрес загрузки» интерпретируется как начальная команда, т. е. точка входа в программу. На рис. 1.2 показаны требуемые карты в порядке их следования. Наконец, пусть ячейки памяти $M[r], \dots, M[r + 79]$ резервируются для последующего считывания в них, где r — произвольно назначенная начальная ячейка области считывания. Тогда однокарточный абсолютный загрузчик, находящийся на первой карте, выполняет следующие действия:

```

Load: Read(for i:=0 step 1 until 79 do M[r+i]);
      LA := M[r]; n := M[r+1];
      if n=0 then Transfer(LA);
      for i:=0 step 1 until n-1 do
          M[LA+i] := M[r+2+i];
      go to Load;
    
```

Процесс загрузки является живым примером «бутстреппинга» (поднимание себя самого за собственные шнурки от ботинок)¹⁾.

В те ранние годы средства автоматизации программирования либо отсутствовали, либо были минимальными — лишь на наиболее сложных установках имелись простые ассемблеры и интерпретаторы с незначительным использованием библиотечных стандартных программ. Когда была признана важность символьического программирования и ассемблирующие системы

¹⁾ Читателю более знакомо аналогичное для описания данной ситуации действие барона Мюнхгаузена, который вытащил самого себя за волосы из болота. — Прим. перев.

получили более широкое распространение, выработалась стандартная операционная процедура: ассемблер считывается загрузчиком; ассемблер преобразует в абсолютную форму «символические колоды» исходных программ пользователя с добавлением библиотечных стандартных программ; ассемблированная программа записывается на ленту или карты, а загрузчик снова используется для считывания карт этой программы в основную память; абсолютная программа затем выполняется. Каждый шаг требует манипуляций со стороны оператора и занимает много времени, в особенности по сравнению со временем, которое требуется ЭВМ для обработки карт на этом шаге.

«Первое поколение» операционных систем было вызвано к жизни упомянутой выше неэффективностью, а также другими соображениями. Эти дополнительные факторы включали: затраты на управление оборудованием; наличие других языков (самой известной была система программирования на ФОРТРАНе); развитие библиотечных программ и вспомогательных средств, в особенности относящихся к операциям ввода-вывода; неудобство трансляции в абсолютную программу, которая требовала, чтобы *все* программные секции и подпрограммы, необходимые для прогона, транслировались вместе первоначально и всегда при внесении изменений в любую из программ. Первые пакетные системы автоматизировали стандартную последовательность «загрузка/трансляция/загрузка/выполнение» с помощью центральной управляющей программы, которая отыскивает и загружает требуемые системные программы (ассемблер, компилятор, загрузчик или библиотечные подпрограммы), а также управляет переходами от задания к заданию. Языковые трансляторы были переписаны для создания программ в перемещаемой, а не в абсолютной форме. Были разработаны связывающие загрузчики, позволяющие смешивать колоды, содержащие программы в исходной и перемещаемой объектной форме, затем также стало возможным хранить и библиотечные программы в перемещаемой объектной форме. Со стороны человека-оператора требовалось управление физическими устройствами, предназначенными для ввода и вывода пакетов, обслуживание нестандартных заданий и восстановление системы при отказах. Автор задания был удален, по крайней мере в принципе, из машинного зала¹⁾, и постепенно пришли к выводу, что лучше всего рассматривать вычислительный комплекс как большой ящик ввода-вывода. В этих ОС защита была самой трудной и насущной проблемой; система могла быть легко разрушена самопроизвольно или пользователем, кроме того, какой-либо пользователь мог легко прочитать в составе своего задания

¹⁾ Запрещен непосредственный доступ к ЭВМ. — Прим. перев.

следующее задание в пакете. Распределение ресурсов, в первую очередь основной памяти и устройств ввода-вывода, было в большей степени задачей языковых процессоров и программ пользователей, нежели ОС.

1.3.2. Второе поколение аппаратуры и программного обеспечения

Поколения аппаратуры ЭВМ определялись до сих пор в терминах технологий их компонентов — вакуумные лампы в первом поколении, транзисторы — во втором и интегральные схемы — в третьем. С точки зрения перспективы развития операционных систем эти различия менее важны, чем различия, которые могут быть обнаружены в архитектуре аппаратуры и программного обеспечения; периоды же времени приблизительно совпадают.

В период с 1959 до 1963 гг. несколько значительных аппаратных разработок получили широкое распространение и стимулировали развитие ОС. Вероятно, наиболее важным аппаратным новшеством был канал данных — простейшая ЭВМ со своей собственной системой команд, регистрами и устройством управления, которое управляет связью и передачей данных между основной ЭВМ и устройствами ввода-вывода. При получении запроса на ввод-вывод от центрального обрабатывающего устройства (ЦОУ) канал выполняет операцию ввода-вывода асинхронно и параллельно с продолжающейся работой ЦОУ; становится возможным совмещение во времени операций ввода-вывода и ЦОУ. Основная память разделяется ЦОУ и каналом, поскольку она хранит программы и данные для них. Вначале ЦОУ могло лишь опрашивать состояние канала, но вскоре стало ясно, что можно повысить эффективность, если бы канал с целью передачи сообщения мог также прерывать работу ЦОУ в большинстве случаев при завершении операции ввода-вывода.

Были написаны более сложные системы программирования ввода-вывода с целью использования преимуществ потенциальной эффективности этой новой архитектуры. В их состав вошли программные средства буферизации для обеспечения автоматического считывания данных, прежде чем они понадобятся программе, и для помещения выходных данных в очередь, с тем чтобы отложить их вывод, а также стандартные программы обработки прерываний, которые должны реагировать на прерывания ввода-вывода и возвращать управление прерванным программам.

Прерывания стали использоваться, чтобы сигнализировать об исключительных внутренних условиях, таких, как арифметическое переполнение, и были добавлены команды для выборочного разрешения (включения), запрещения (выключения) и

приостановки (задержки действия) механизмов прерывания. Стали доступными внутренние часы, которые могли быть запрограммированы для прерывания ЦОУ через определенный интервал времени; они позволили супервизорной программе управлять значением интервала времени ЦОУ, выделяемого каждому пользователю, что сделало возможным автоматическое распознавание некоторых ошибочных или слишком длинных программ.

Естественно, что пользователи, с одной стороны, передали центральной системе управление прерываниями и обслуживанием ввода-вывода, а руководство комплекса, с другой стороны, начало настаивать на применении этих средств пользователями. Искушенный программист все еще часто предпочитал писать свои собственные пакеты. Системная защита оставалась серьезной и нерешенной проблемой; систему слишком легко было разрушить как опытному, так и начинающему программисту.

В течение этого периода были значительно расширены библиотеки за счет включения программ-утилит, таких, как программы сортировки или программы переноса данных с карт на ленту, а также за счет дополнительных языковых процессоров. Вместо магнитной ленты для хранения системы и библиотек стали использоваться файлы прямого доступа (обычно диски). По мере пополнения и усложнения задач ОС (воспринимаемой как «машина» типичным пользователем, который все больше отдаляется от реальной аппаратуры) оказалось необходимым определить более систематическим способом характеристики заданий и требований к ним; для этой цели были введены языки управления заданиями.

Мы можем обобщенно характеризовать это поколение как период становления последовательной пакетной обработки со многими плодотворными попытками эффективно использовать каналы данных, прерывания и вспомогательную память. Однако обработка заданий по одному в каждый момент времени по-прежнему приводила к низкой активности канала для заданий с интенсивными вычислениями и к низкой активности ЦОУ для заданий с интенсивным вводом-выводом, даже если было достигнуто максимальное совмещение операций ЦОУ и канала.

1.3.3. Системы третьего и последующих поколений

С 1962 по 1969 г. в больших ОС стал почти повсеместно использоваться новый метод обработки заданий — *мультипрограммирование*, был также разработан новый способ выполнения вычислений, *разделение времени*, как альтернатива пакетной обработке. Большие дисковые устройства с быстрым доступом обеспечили непосредственно управляемую память для

систем и библиотечных программ пользователей, а также для ожидающих обработки заданий пользователей. Во многие ЭВМ были введены аппаратные средства защиты команд и памяти; в некоторых больших машинах появились системы основной памяти с аппаратными средствами настройки адресов, которые допускали реализацию большой «виртуальной» памяти; стали более распространенными мультипроцессорные конфигурации. Проблемы распределения ресурсов и защиты стали более острыми и трудноразрешимыми в мультипрограммной среде, где много процессов могут одновременно запрашивать как совместное, так и монопольное использование ресурсов системы, часто должны передавать сигналы друг другу и могут являться потенциально «злонамеренными»¹⁾ или ошибочными. Именно в этот период исследование операционных систем стало одним из главных направлений в области вычислительной техники.

Многие аппаратурные и программные новшества появились уже в первых машинах середины 40-х г. и темп их появления не снизился за последние годы. Достижения в области технологии производства больших интегральных схем и в области массового производства стандартных компонентов ЭВМ сделали возможным строить недорогие, но мощные ЭВМ; число и разнообразие их будут увеличиваться в ближайшем будущем, что требует экономичного программного обеспечения, соответствующего этой аппаратуре. В то же время начинают предлагать коммунальные услуги большие сети ЭВМ, построенные на основе линий связи; сети предъявляют еще большие требования к системному программному обеспечению. Другие изменения в аппаратуре, такие, как появление управляющей памяти с возможностью перезаписи микропрограмм, иерархий устройств основной памяти с автоматической передачей информации между уровнями, экономичных ассоциативных запоминающих устройств, а также углубление параллелизма в обрабатывающих устройствах, в том числе программируемого, приведут к новым требованиям в отношении программного обеспечения и заставят глубже понимать принципы ОС. К современным направлениям в области разработки программного обеспечения можно отнести: языки высокого уровня для системного программирования; выявление адекватных примитивов ОС для планирования ресурсов и процессов, которые могли бы быть реализованы с помощью аппаратуры, микропрограмм или стандартного программного обеспечения; человеко-ориентированные языки управления заданиями; языки и системы, обеспечивающие параллелизм; подсистемы для измерения действия ОС; более универсальные схемы защиты.

¹⁾ Процессы, стремящиеся осуществить некие действия в ущерб другим процессам. — Прим. перев.

Лишь недавно стали в ограниченной степени доступными систематические методы проектирования, анализа и моделирования операционных систем; эти методы находятся в стадии широкого исследования. Мы были здесь умышленно кратки, поскольку в последующих главах рассматриваются операционные системы на уровнях третьего и более поздних поколений.

1.4. Некоторые аспекты операционных систем

Некоторые ранние типы системных программ были языковыми процессорами — это ассемблеры, макроассемблеры, интерпретаторы и компиляторы. Они более не рассматриваются как компоненты ОС, а трактуются как прикладные программы; они пользуются услугами, предоставляемыми ОС, и выполняются под их управлением. (Важным применением языковых процессоров является трансляция операционных систем и подсистем, написанных на ассемблере и на языках более высокого уровня, включая случай ОС, работающей под управлением другой ОС.)

Важнейшие системные модули, составляющие ОС, могут быть разделены на два частично перекрывающихся класса. Первый составляют те, которые могут быть вызваны прямо или косвенно во время выполнения программы пользователя. Наиболее очевидными примерами являются стандартные программы ввода-вывода и другие средства обслуживания файловых систем. Второй класс состоит из тех программ, которые вызываются или явно, посредством директив, или неявно, посредством деклараций, а также посредством предложений языка *управления*, выдаваемых пользователями или оператором машины. Эти два класса представляют большую часть ОС¹⁾ и включают:

1. Планировщики для назначения центрального обрабатывающего устройства (устройств) заданиям, частям задания и системным процессам.
2. Стандартные программы управления памятью, которые управляют распределением основной памяти.
3. Контроллеры ввода-вывода, которые обслуживают запросы ввода-вывода для всех пользователей системы.
4. Процедуры связывания и загрузки для настройки адресов программ загрузки и связывания набора программ.
5. Программы файловых систем для управления доступом, накоплением и перемещением файлов данных в запоминающей среде ЭВМ.

¹⁾ Мы опустили те модули, которые предназначены для генерации и сопровождения (поддержки работоспособности) системы.

6. Обработчики прерываний для обслуживания внешних и внутренних прерываний.

Эти программы вызывают друг друга и взаимодействуют сложными и часто хитроумными способами.

1.4.1. Виртуальные машины, трансляция и распределение ресурсов

Полезно различать реальную аппаратуру ЭВМ и принципы ее работы, с одной стороны, и «машины», воспринимаемые различными классами пользователей, — с другой. Последние мы будем называть *виртуальными машинами*, так как они редко являются реальными ЭВМ, а наиболее часто — результатом наслоений программного обеспечения на аппаратуру ЭВМ.

Примеры

1. Рассмотрим подсистему компиляции, загрузки и запуска для программ, представленных на языке высокого уровня, например на языке АЛГОЛ. Программист на АЛГОЛе обращается к ЭВМ с этой подсистемой как к машине, которая непосредственно выполняет АЛГОЛ. Следовательно, пользователь этой виртуальной машины не имеет дела с проблемами распределения регистров и памяти, программами канала, прерываниями, функциями размещения массивов и другими задачами, которые возникают на «нижнем» уровне.

2. Разработчики подсистемы, описанной в п. 1, возможно, использовали макроассемблерную систему, которая включает ряд «системных» макросов, например, для ввода-вывода. Программисты на языке макроассемблера могут не беспокоиться о деталях распределения памяти для своих программ, если их виртуальная машина автоматически обеспечивает это; однако им необходимо проектировать стандартные программы управления памятью для АЛГОЛ-машины, например для обслуживания стека во время выполнения.

3. Системный программист, работающий над базовым модулем, таким, как система буферизации ввода-вывода, обычно использует виртуальную машину, состоящую частично из реальной машины, но дополненную некоторыми примитивами «ядра» операционной системы, например макросами для передачи сообщений между процессами и для манипуляции с очередями.

4. Язык управления заданиями, на котором пользователь описывает свое задание и требования к ресурсам, представляет виртуальную машину с функциями операционных систем по отношению к заданиям и шагам заданий.

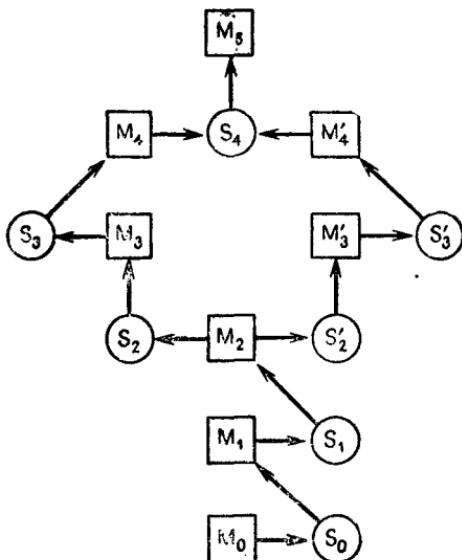


Рис. 1.3. Иерархия виртуальных машин (Вейдерман, 1971).

5. У пользователя можно вызвать иллюзию работы с очень большой центральной памятью; система в свою очередь реализует эту виртуальную память на значительно меньшей и, может быть, по-другому организованной реальной основной памяти.

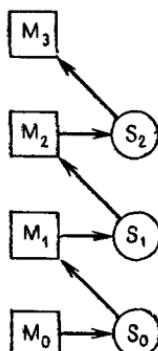
Пользователь, таким образом, имеет доступный ряд машин, из которых он может создать более абстрактные машины. На самом нижнем уровне расположена реальная ЭВМ. Системный программист, использующий машину на уровне i и ниже, пишет программное обеспечение для построения новой виртуальной машины на уровне $i + 1$. Вейдерман (1971) графически изобразил эту иерархию виртуальных машин, как показано на рис. 1.3; M_i обозначает машину, а S_i — программное обеспечение на уровне i . На этом рисунке программный модуль S_0 , представленный на языке реальной ЭВМ M_0 , реализует новую виртуальную машину M_1 . Аналогично, программное обеспечение S_2 , написанное для виртуальной машины M_2 , дает новую виртуальную машину M_3 , в то время как S'_2 , написанное для той же самой машины, что и S_2 , дает другую виртуальную машину M'_3 .

Так как операционные системы и языковые процессоры создают виртуальные машинные интерфейсы для пользователей ЭВМ, то их основная функция может быть определена как функция трансляции; они последовательно транслируют машины с более высоких уровней в более низкоуровневые эквиваленты, пока, наконец, не будет достигнута реальная ЭВМ. Рассмотрим возможную последовательность трансляций, начиная

с машины на языке высокого уровня. Пусть M_1, M_2 и M_3 — машины, произведенные программным загрузчиком S_0 , ассемблером S_1 и компилятором S_2 соответственно; эта иерархия показана на рис. 1.4, а. Программа на языке высокого уровня x_4 транслируется последовательно в программу на языке ассемблера x_3 , перемещаемую машинную программу x_2 и в абсолютную программу на машинном языке x_1 . Последняя затем транслируется (выполняется, интерпретируется) для получения выходной строки x_0 (рис. 1.4, б).

Языковые процессоры — это по своему существу трансляторы, но трансляция — упрощенное и второстепенное по смыслу понимание функций ОС (что, возможно, противоречит точке зрения пользователя). Более существенная функция ОС состоит в распределении и управлении ресурсами; мы придаём термину «ресурсы» общий смысл, который охватывает не только аппаратные ресурсы ЭВМ, но и программные ресурсы, такие, как разделяемые программы и данные, а также сообщения, передаваемые между процессами. Когда ресурс является *активным устройством*, таким, как центральный процессор или канал данных, распределитель (allocator) обычно называется планировщиком (scheduler).

Здесь уместно подчеркнуть, что самой важной задачей любой системы является оказание помощи пользователям ЭВМ в решении их проблем. Это достигается путем предоставления им языковых средств, которые позволяют эффективно выражать



(a)

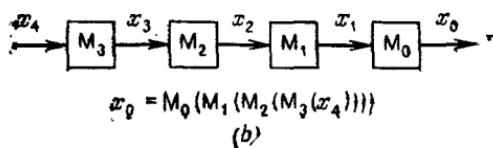


Рис. 1.4. Системы как трансляторы: (а) — последовательность виртуальных машин до языкового процессора высшего уровня; (б) — трансляция x_4 в x_0 .

алгоритмы, путем централизованного распределения ресурсов и выполнения служебных функций так, чтобы их проблемы могли быть решены экономично, а также путем планирования их процессов таким образом, чтобы обеспечить быстрый ответ.

1.4.2. Четыре ключевые проблемы

Существуют четыре связанных между собой проблемы, которые свойственны мультипрограммным ОС (Миллер, 1968). Мы кратко рассмотрели две из них: *преобразование виртуальных машин и распределение ресурсов*. Проблема трансляции виртуальной машины возникает и в *статической* форме, как в примере из предыдущего раздела (рис. 1.4), и в *динамической* форме, как, например, настройка адресов команд и данных во время выполнения. Эту трансляцию можно рассматривать как установление соответствия имени/ресурс. Предположим, например, что периферийные устройства обозначены символически и что адреса памяти перемещаемы¹⁾ на некотором виртуальном уровне; тогда они должны быть преобразованы в абсолютные адреса устройств и памяти соответственно. Распределение ресурсов и планирование становится проблемой обеспечения эффективного использования аппаратуры при тех ограничениях, которые пользователь накладывает на время получения ответа.

Третья проблема — это проблема *защиты*. Мы должны гарантировать сохранность ОС и программ пользователей от случайного или умышленного повреждения или ошибочного обращения к ним со стороны пользователей или самой ОС. Это не только вопрос предотвращения разрушения информации, но и вопрос сохранения секретности и охраны собственности. Частные пользовательские файлы или части операционной системы могут потребовать гарантий от несанкционированного считывания или выполнения. Кроме того, владелец файла (т. е. данных или программ) должен иметь возможность разрешать определенному классу пользователей считывание, запись или выполнение своего файла.

Четвертая проблема, которая характеризует эти системы, возникает из необходимости *синхронизовать и обеспечивать взаимодействие* резидентных системных процессов и иногда пользовательских процессов. Физически, а также абстрактно, многие процессы развиваются одновременно. Данная независимая программа, например задание пользователя, должна всегда вырабатывать один и тот же ответ, независимо от других программ, которые могут выполняться в то же самое время. Когда

¹⁾ Адреса, которые необходимо настроить, чтобы осуществить процесс *перемещения* программы в память. — Прим. перев.

программы требуют синхронизации (например, когда программа ввода поставляет буферизованные данные для основной программы), необходимо быть внимательным, чтобы предотвратить ситуации, в которых не выполняется ни одна программа, потому что каждая ждет сигнала от другой, или ситуации, когда программа пропускает сигнал.

В этой книге читатель будет постоянно сталкиваться с указанными четырьмя проблемами и будет рассмотрено множество методов их решения.

1.5. Организация систем

Существует огромное число возможных конструкций аппаратуры; к счастью, большинство из них является расширениями и вариантами стандартной конфигурации. На рис. 1.5 представлена блок-схема простейшего устройства такой ставшей теперь классической машины. В дальнейшем, если не оговорено особо, будем полагать, что наша программная ОС спроектирована *по меньшей мере* для такой ЭВМ.

Центральное обрабатывающее устройство, как обычно, состоит из внутренних и программируемых регистров, арифметического устройства (или устройств) и устройства управления системой. Последнее управляет порядком следования и выполнения команд, прямо или косвенно приводя в действие остальную аппаратуру ЭВМ. Ниже представлен основной алгоритм, выполняемый устройством управления ЦОУ; алгоритм может быть реализован микропрограммой или логической схемой:

loop; *Выбрать следующую команду;*
 Увеличить счетчик команд;
 Выполнить команду;
 go to loop;

Разрешенные прерывания от устройств ввода-вывода, контроллера, канала и (или) самого цикла исполнения обслуживаются в конце цикла команды после третьего шага. Если в течение одного цикла одновременно поступило несколько прерываний, то аппаратный механизм приоритетов может проигнорировать их все, кроме одного, в течение по крайней мере еще одного цикла. Предполагается, что ЦОУ имеет внутренний таймер, который может быть установлен программой, чтобы вызвать сигнал прерывания по прошествии определенного интервала времени.

Основная память рассматривается как набор ячеек с адресами, образующими непрерывную возрастающую последовательность. Каждая из этих ячеек имеет единичную емкость

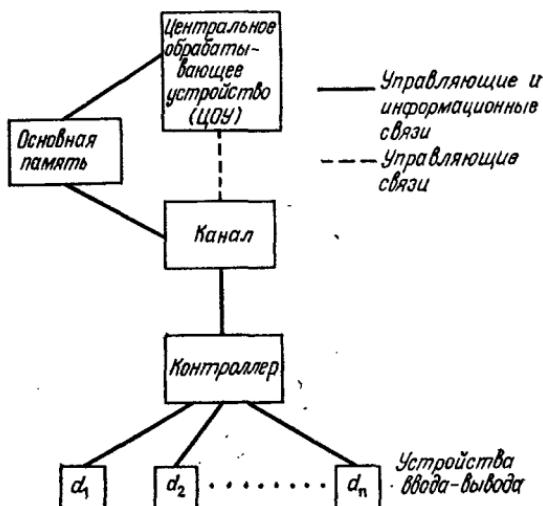


Рис. 1.5. Машина с минимальной конфигурацией.

(бит, байт, слово, цифра, знак...). Канал данных (разд. 1.3.2) может передавать данные или управляющую информацию из основной памяти (а иногда из регистров ЦОУ) в контроллер ввода-вывода и пересыпать из последнего данные или информацию о состоянии обратно в основную память или ЦОУ; в общем случае это выполняется параллельно с работой ЦОУ. Контроллер — это процессор, настроенный на определенный тип устройств, который выбирает устройства с определенными адресами и управляет их работой. Устройства ввода-вывода могут включать пакетные периферийные устройства, такие, как устройства чтения с перфокарт, перфораторы, перфоленточные устройства и устройства построчной печати, а также интерактивные терминалы, в частности пишущие машинки и графические устройства; они, конечно, будут включать некоторые устройства вспомогательной памяти большой емкости, например на барабанах или дисках и, возможно, даже на магнитных лентах.

Некоторые из особенностей этой аппаратуры подробно рассмотрены в последующих главах, в тех случаях, когда они оказывают значительное влияние на состав программного обеспечения.

Для операционных систем не существует стандартных общепринятых принципов организации. Однако мы можем привести некоторые общие замечания по поводу того, как они бывают устроены. Неформально определим *состояние* ОС как совокупность состояний всех процессов и ресурсов в системе. Состояние процесса означает, готов ли он начаться, развивается ли он на

процессоре или логически заблокирован, ожидая выделения запрошенного ресурса; состояние класса ресурсов означает их текущее распределение и список процессов, ожидающих удовлетворения своих запросов на ресурсы. Большинство изменений состояния ОС происходит в результате прерываний, которые вызывают процессы в ОС. Можно сказать в этом смысле, что программы обработки прерываний представляют собой движущую силу, обеспечивающую функционирование всех систем, и мы говорим, что операционные системы являются *управляемыми по прерываниям*¹⁾.

Изменение состояния осуществляется такими компонентами ОС, как *управление ресурсами* и *управление процессами*. Третьей важной частью является *файловая система*, так как обычно вся передача информации через память и устройства ввода-вывода регулируется централизованно. Остальные важные компоненты ОС могут быть описаны в терминах процесса, ресурса, а также компонентов файловых систем; известным примером является часть системы «управление заданиями», которая управляет инициированием, выполнением, завершением и учетом заданий. Некоторое необходимое программное обеспечение, такое, как стандартные программы распечатки содержимого памяти или утилиты для работы с периферийными устройствами системы, часто могут рассматриваться как прикладные программы, даже когда их запрашивает сама операционная система.

¹⁾ Существуют системы, которые не являются управляемыми по прерываниям, а вместо этого полагаются на отдельные аппаратные процессоры, которые выставляют «заявки» к центральному процессору при возникновении запросов на изменение состояния; примером могут служить машины CDC 6400/6500/6600 (CDC, 1969). Хотя мы будем предполагать работу с управлением посредством прерываний, большинство рассматриваемых методов не зависит от этого выбора и также применимо к ситуации опроса заявок и к системам, где простой вызов системной стандартной программы также может изменить состояние процесса и ресурса.

2. Системы пакетной обработки

2.1. Введение

Термин «пакетная обработка» происходит от способа, по которому производится накопление и выполнение заданий. В ранних системах этого типа представленные для выполнения задания собирались в пакеты, или наборы, и обрабатывались на ЭВМ последовательно, по одному заданию. Основные задачи, выполняемые операционной системой, обеспечивающей пакетную обработку, следующие:

1. *Управление заданиями.* Эта самая основная и высокоразвитая функция обеспечивает инициирование и прекращение выполнения как заданий, так и шагов заданий. Первичным действием здесь является интерпретация управляющих карт задания с целью (а) запуска и прекращения учетных процедур для задания, (б) установления соответствия между действительными адресами устройств ввода-вывода и символическими адресами, использованными в задании, и (с) вызова соответствующих системных программ для выполнения задачи, запрашиваемой шагом задания.

2. *Загрузка и связывание программы.* Программам распределяется память, они перемещаются¹⁾ в соответствии с этим распределением и загружаются в память. В то же время организуется связывание независимо оттранслированных программ, которые используются совместно. Эти программы могут быть объектными программами пользователя, модулями операционной системы или элементами общей библиотеки пользователей.

3. *Управление вводом-выводом.* За исключением особых случаев, система ответственна за весь ввод-вывод. Запросы на ввод-вывод посылаются системным программам, которые планируют и инициируют операции, а затем управляют ими. Эта задача часто включает в себя распределение устройств и вспомогательной памяти, а также программную буферизацию. Целью такой централизации является улучшение как эффективности, так и защиты системы, а также освобождение пользователя от излишних деталей программирования ввода-вывода.

¹⁾ Процесс перемещения программы (далее просто перемещение программы) заключается в определенной ее обработке, связанной с подготовкой программы для выполнения в распределенном для нее месте памяти. Обработка заключается в настройке адресов команд и данных. — Прим. ред.

В этой главе рассматриваются важные принципы, лежащие в основе построения связывающих загрузчиков и программирования ввода-вывода в рамках систем пакетной обработки. Это не означает, что дается исчерпывающее описание классических систем пакетной обработки. Нашей целью является введение некоторых важных понятий и методов, которые полезны сами по себе и будут применены в последующих главах к мультипрограммным системам.

2.2. Связывание и загрузка

Статическая структура программы может быть представлена деревом, как показано на рис. 2.1, где каждая вершина P_i обозначает или процедуру или программный сегмент. P_0 — основная программа, а вершины-преемники любой P_i содержат процедуры, вызываемые P_i , и программные сегменты, на которые передается управление из вершины P_i . Например, на рисунке P_0 вызывает P_1 и в свою очередь P_1 вызывает P_4 и P_5 . Многочленное использование одной и той же процедуры, в том числе и рекурсивное, изображается тем, что две или более вершин имеют одно и то же имя. Процедуры могут быть библиотечными стандартными или пользовательскими программами. Выполнение программы требует, чтобы было предусмотрено средство для загрузки любой процедуры статического дерева в память и для совместного связывания вызывающих и вызываемых процедур.

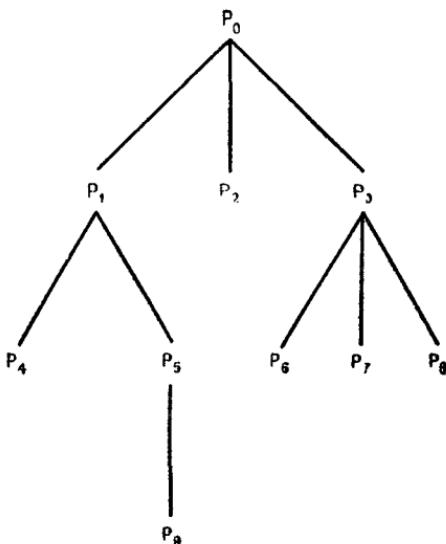


Рис. 2.1. Статическая структура дерева программы.

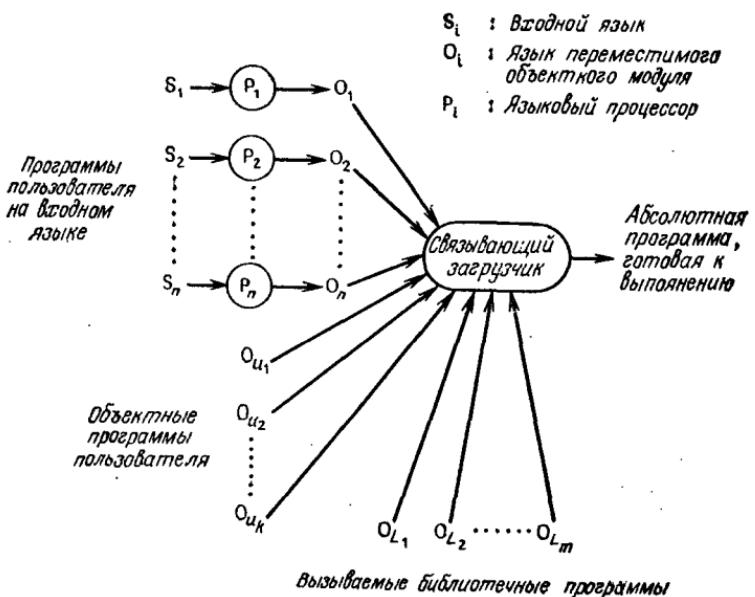


Рис. 2.2. Связывающие загрузчики.

Ранние языковые процессоры производили объектные программы в форме *абсолютных* машинных кодов; ячейки памяти, где должны располагаться программы и данные во время выполнения программы, определялись языковыми процессорами. Это означало, что *все* процедуры, которые встречались в статической структуре программы, должны были быть оттранслированы вместе, чтобы избежать возможных конфликтов, возникающих при распределении памяти. В общем случае независимо оттранслированные программы не могли быть объединены, так как им могли быть выделены перекрывающиеся участки памяти. Таким образом, пользователь был вынужден включать тексты на исходном языке для всех библиотечных и прикладных программ, на которые есть хотя бы косвенные ссылки в его программе.

Эти трудности были преодолены тем, что языковые процессоры стали выдавать программы в перемещаемой объектной форме¹⁾, содержащей символьную информацию для связывания. Каждая процедура или программный сегмент транслируется независимо. Программам распределяется память, они перемещаются в соответствии с этим распределением, загружаются

¹⁾ Форма программы, предполагающая дальнейший этап перемещения программы для настройки ее адресных частей на распределенное место. — Прим. перев.

и совместно связываются непосредственно перед выполнением. Набор системных программ, которые выполняют перемещение, связывание и загрузку, называется *связывающим загрузчиком*; процесс называется *статическим* перемещением, связыванием и загрузкой, так как эти задачи полностью выполняются перед выполнением программы. (Эти задачи могут также решаться динамически во время выполнения; такой случай будет рассмотрен в гл. 5 и 6.) Весь процесс изображен на рис. 2.2.

2.2.1. Статическое перемещение

Так как память не распределена до момента загрузки, языковые процессоры будут транслировать исходные программы в последовательность записей, представляющих команды и данные, относительно некоторой произвольной базовой ячейки, обычно 0. Затем эти записи перемещаются при назначении памяти.

Оттранслированная запись о команде обычно является вариантом следующей формы:

Loc *oc* $a_1 a_2 \dots a_n$

где *Loc* — относительный адрес команды, *oc* — код операции и a_i — поля операндов команды. Запись *Loc* может быть опущена, за исключением случая, когда в $(i+1)$ -й по порядку записи относительный адрес не следует за адресом в i -й записи. Если объектная программа должна загружаться в непрерывную область памяти, начинающуюся с ячейки α , а программа была оттранслирована относительно ячейки β , то командная часть приведенной выше записи будет занесена в ячейку $Loc + k$, где константа *перемещения* $k = \alpha - \beta$. Теперь рассмотрим адресные поля. Каждое a_i может быть именем регистра, непосредственным операндом, адресом операнда или адресом команды в памяти (мы до некоторой степени упрощаем положение, поскольку может присутствовать и другая информация, такая, например, как флаги косвенной адресации). В последних двух случаях, настройка адресов может оказаться необходимой в зависимости от структуры машины и исходной программы. Адресные поля могут быть всегда относительными, например, если при вычислении эффективного адреса¹⁾ используются базовые регистры; здесь не требуется никакой настройки. Однако, когда адресные поля рассматриваются машиной как абсолютные, адрес a_i должен быть изменен загрузчиком на $a_i + k$. Записи,

¹⁾ В оригинале *effective address*, что соответствует переводу «исполнительный адрес». Однако автор далее использует этот термин в более широком смысле, более близком к смыслу понятия «математический адрес». Поэтому далее в тексте используется обобщенный термин «эффективный адрес». — Прим. перев.

представляющие команды, могут быть дополнены индикаторами, определяющими, какое поле a_i надо настроить. Кроме того, для указания на эти операнды может быть использован глобальный словарь перемещений, связанный с программой.

Данные представляются подобным способом. Эти записи имеют общую форму:

$Loc \ d_1d_2\dots d_m$

где Loc — относительный адрес данных и d_i — элементы данных. Как и прежде, элементы данных будут загружаться, начиная с ячейки памяти $Loc + k$. Поле d_i может также содержать относительный адрес; d_i должен быть также изменен на k .

2.2.2. Процесс связывания

Сегмент программы или данных может как (а) определять символы для возможной ссылки на них из других программ, так и (б) ссылаться на символическую информацию, определенную другими модулями. Примерами (а) являются имя программы, точки входа в программу и имена областей данных; ссылки на внешние символы [(б)] реально встречаются при вызовах процедур и использовании глобальных данных (рис. 2.3). Как внутренне определенные символы, так и внешние используемые символы должны быть явно указаны в перемещаемых объектных модулях для использования их связывающей программой.

С каждым перемещаемым объектным сегментом должна быть связана таблица определения символов и таблица использования символов (рис. 2.4). Таблица определения перечисляет все символы класса (а). Каждая запись таблицы есть пара $(Dname[i], Dval[i])$, где $Dname$ содержит символы, а $Dval$ — соответствующие им значения; значение символа является обычно перемещаемым адресом команды или данных в пределах сегмента. Когда сегмент перемещается связывающим загрузчиком, элементы сегмента с перемещаемым адресом $Dval$ настраиваются единообразно на место в памяти с абсолютным адресом, определенным символом этого элемента. Эта таблица затем становится частью глобальной таблицы символов, которая используется, чтобы отыскать адреса для внешне определенных символов.

Таблица использования состоит в минимальном своем варианте из списка всех внешних символов, на которые имеются ссылки в сегменте. Существуют два общих метода для управления внешними ссылками, когда создаются перемещаемые сегменты. Первый заключается в использовании ячеек, где ссылки

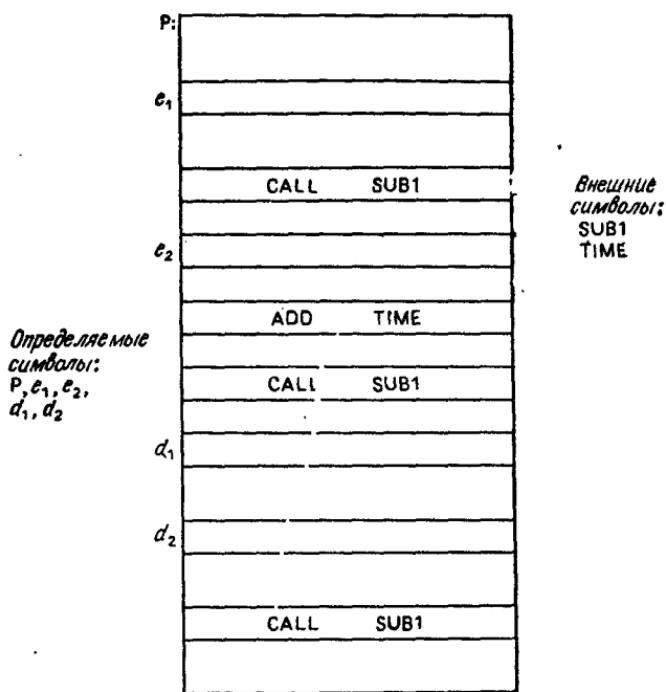


Рис. 2.3. Определяемые и внешние символы в сегменте программы или данных.

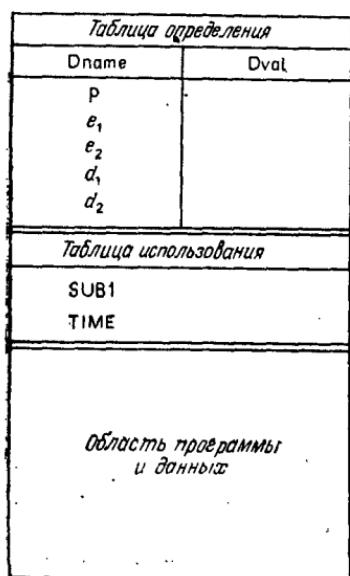


Рис. 2.4. Форма перемещаемого сегмента.

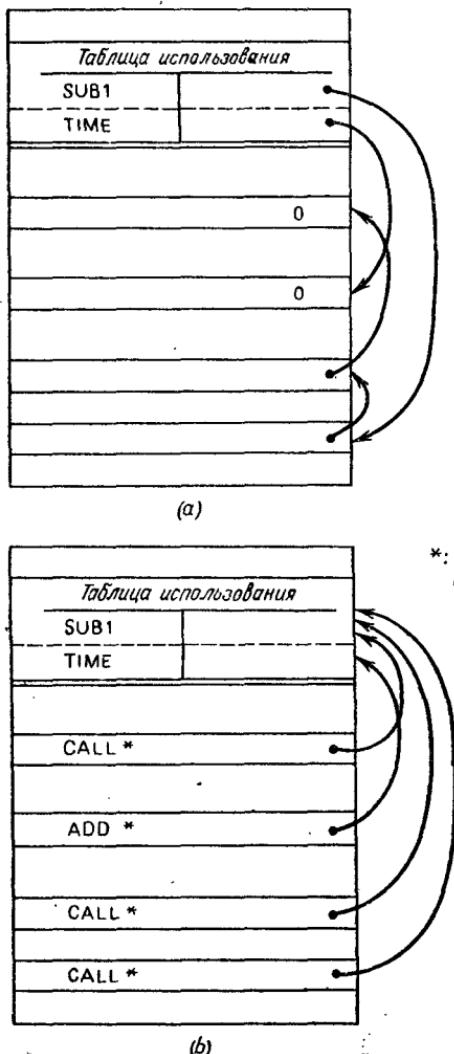


Рис. 2.5. Обработка ссылок к внешним символам: (а) — метод цепочек; (б) — косвенная адресация.

встречаются в качестве звеньев в цепочке, начинающейся записью таблицы использования (рис. 2.5(а)). Когда символ определен, абсолютные адреса могут быть легко вставлены в такие корректируемые ячейки с помощью просмотра этой цепочки; последний элемент в цепочке будет иметь нулевое звено, скажем 0.

Пример

Пусть

- a) P есть программный сегмент, который был перемещен на константу перемещения k и загружен в память M ,
- b) таблица использования сегмента P содержит n записей:

$(Uname[i], Uval[i]), i = 1, \dots, n,$

- c) $Uval[i]$ указывает на последнее использование $Uname[i]$ в P (предположим, что $Uval[i]$ есть перемещаемый адрес) и является головным элементом цепочки использования этого символа и

- d) $Uname[i]$ содержит не символ, а соответствующее ему значение.

Тогда внешние ссылки проставляются в P следующим алгоритмом:

```

for i := 1 step 1 until n do
begin
  Fixup := Uname[i];
  Next := Uval[i];
  for adr := Next + k while Next ≠ 0 do
  begin
    Next := M[adr];
    M[adr] := Fixup
  end
end;

```

Второй метод устраняет только что рассмотренную процедуру с помощью косвенной адресации для всех внешних символов (рис. 2.5(b)). Когда символ определен, его значение¹⁾ заносится в ячейку, к которой допустима косвенная адресация; последняя иногда называется *вектором передачи*. Связывание происходит быстрее и проще, но во время выполнения для ссылки на символ требуется два цикла обращения к памяти вместо одного.

Связывание и загрузка может быть выполнена за один или два прохода. Двухпроходный метод предполагает выполнение следующих шагов:

A. Проход 1:

1. Получить очередной перемещаемый объектный сегмент P ,
2. Переместить и загрузить P .

¹⁾ Значение символа — это его абсолютный адрес. — Прим. перев.

3. Откорректировать глобальную таблицу символов в соответствии со значениями символов, определяемых в P .
4. Повторить шаги 1—3 для каждого сегмента P .

В. Проход 2:

1. Получить таблицу использования для очередного сегмента P .
2. Разрешить внешние ссылки в P поиском определений каждой записи таблицы использования в глобальной таблице символов.
3. Повторить шаги 1—2 для каждого загружаемого сегмента P .

Эта символьная обработка подобна обработке, применяемой в стандартном двухпроходном ассемблере. Аналогия с ассемблером также применима к однопроходному способу; основная часть обработки заключается в разрешении трудностей, вытекающих из ссылок на символы вперед по отношению к текущему месту обработки (см. упражнение ниже). Необходимо уяснить себе, что методы связывания в значительной степени касаются создания, поиска и манипуляции с таблицами символов.

Операции связывания и загрузки часто разделяются на две отдельные части. Первая часть вырабатывает «загрузочный модуль», состоящий из всех объектных сегментов, связанных и перемещаемых вместе относительно стандартного базового адреса; загрузочный модуль обычно помещается во вспомогательную память. Другая часть, операция загрузки, загружает модуль в основную память, настраивая адреса в соответствии с распределением памяти для модуля.

Основные компоненты связывающего загрузчика были описаны отдельно, без упоминания их взаимодействия с другими частями операционной системы. В частности, чтобы разместить или отыскать требуемый набор перемещаемых объектных модулей, должны многократно использоваться ввод-вывод и файловые системы; мы также не рассматривали процесс распределения основной памяти, который должен быть полностью завершен перед загрузкой.

Статические способы перемещения и связывания, описанные в предыдущих разделах, применяются во многих операционных системах. Более развитые и динамические методы, описанные в гл. 5 и 6, требуют подобной информации о перемещении, а также таблиц определения и использования символов в объектных программах.

Упражнение

Напишите алгольную процедуру для однопроходного связывающего загрузчика. Процедура должна последовательно читать перемещаемые объектные программы, составляющие задание, и выполнять требуемое перемещение, связывая и загружая. Каждый объектий сегмент Q_i должен быть представлен в форме:

$$(d, Dname[1], Dval[1], Dname[2], Dval[2], \dots, Dname[d], Dval[d], u, Uname[1], Uval[1], Uname[2], Uval[2], \dots, Uname[u], Uval[4], t, Text[1], Text[2], \dots, Text[t]),$$

где $d \geq 0$, $u \geq 0$, $t \geq 1$.

Каждое значение в таблице определения $Dval$ указывает на относительную позицию (по отношению к 0) ассемблируемой программы в O_i . Каждое значение таблицы использования $Uval$ указывает на последний элемент в цепочке использования соответствующего неопределенного символа. Массив $Text$ содержит ассемблированные команды и информацию о перемещении. Вам предоставляется стандартная программа $Relocate(t, Text)$, которая перемещает и загружает программу $Text$ в память M и модифицирует счетчик ячеек LC ; в любое время LC содержит адрес следующего слова, которое нужно занести в память; т. е. следующее слово будет помещено в $M[LC]$. Предположим, что глобальная таблица символов состоит из трех массивов, $Name$, Adr и $Flag$, имеющих следующий смысл:

1. Если $Flag[i] = \text{false}$, то символ, данный в $Name[i]$, не определен. Элемент $Adr[i]$ будет тогда указывать на первую ячейку в M из цепочки всех использований $Name[i]$.

2. Если $Flag[i] = \text{true}$, то символ $Name[i]$ определен. Элемент $Adr[i]$ будет тогда его абсолютным адресом в M ; т. е. $Adr[i]$ содержит значение этого символа.

Дана также программа $Search(arg, index)$, которая будет искать в таблице символов совпадение между arg и некоторым $Name[i]$. Если таковое найдено, $Search$ возвращает управление с признаком true , а $index$ указывает найденную позицию в таблице. В противном случае $Search$ возвращает управление с признаком false , а $index$ указывает следующую свободную позицию в таблице символов.

2.3. Методы ввода-вывода

Устройства памяти в большой вычислительной системе могут быть упорядочены в соответствии со скоростями, с которыми они обеспечивают доступ:

1. Процессорная память: индексные регистры, накопители, командные регистры, ...
2. Основная память: на сердечниках, на тонких пленках, на интегральных схемах, ...
3. Вспомогательная память: барабан, диск, магнитная лента, ...
4. Периферийные устройства: устройство чтения с перфокарт, перфораторы, перфоленточные устройства, дисплеи, телетайпы, ...

Скорость передачи информации между устройствами в пределах этой иерархии разнится приблизительно в 10^9 раз — от одного знака в секунду в некоторых периферийных устройствах до миллиардов знаков в секунду в центральном процессоре. Каждое из упомянутых устройств в некотором контексте может рассматриваться как устройство *ввода-вывода*; например, информация часто передается прямо или косвенно туда и обратно между устройством вспомогательной памяти и центральным процессором, другими устройствами вспомогательной памяти или периферийными устройствами.

Операции ввода-вывода могут быть разделены на три основные категории:

1. Чтение/запись. Эти операции передают данные из одного запоминающего устройства в другое.

2. Управление. Это команды, посылаемые на контроллеры устройств для выполнения таких действий, как установка механизма доступа диска, перемотка или пропуск записей на ленте, прогон страниц на печатающем устройстве. В общем случае операции управления подготавливают запоминающие устройства для чтения или записи.

3. Проверка состояния. С помощью этих команд может быть опрошено состояние устройств ввода-вывода и процессоров; наиболее общим примером операции этого класса является проверка занятости вводом-выводом.

Когда вызываются и выполняются машинные команды, операции ввода-вывода между регистрами обрабатывающего устройства, между процессором и основной памятью планируются и управляются автоматически с помощью аппаратуры; несоответствие между временем доступа к основной памяти и скоростью процессора преодолевается во многих ЭВМ посредством просмотра команд вперед и памяти с расслоением¹⁾). Проблема ввода-вывода в системном программировании — это задача оптимальной организации, планирования, управления и передачи данных между элементами основной памяти, вспомогательной памяти и периферийными устройствами. Конечной целью является достижение максимального перекрытия операций ввода-вывода так, чтобы устройства на каждом уровне в иерархии памяти могли работать непрерывно со своими номинальными скоростями. Система пытается достигнуть этого разнообразными программными и аппаратными методами, такими, как

¹⁾ В оригинале „interleaved storage” — оперативная память, организованная из нескольких независимых (логически) блоков и допускающая одновременное обращение к ним, что позволяет увеличить ее пропускную способность — Прим. перев.

буферизация ввода-вывода, прерывания, использование каналов и сопрограмм. Мы рассмотрим эти методы в оставшейся части настоящей главы.

Представленные методы и проблемы ввода-вывода не только важны сами по себе, но также служат примерами и вскрывают содержание более общих проблем, которые встречаются при распределении ресурсов и мультипрограммировании.

2.3.1. Прямой ввод-вывод

Большинство ранних ЭВМ и некоторые из небольших современных ЭВМ имеют *прямые* команды ввода-вывода. Под «прямыми» мы подразумеваем то, что вся операция ввода-вывода управляет непосредственно и последовательно центральным процессором в его нормальном цикле выполнения команд. Этот цикл включает в себя иницирование операций, управление областями памяти, используемыми при вводе-выводе, подсчет переданных знаков и проверку на отсутствие ошибок; ЦОУ недоступно до тех пор, пока команда не выполнена полностью. Время t_p для полного прохождения программы p : $t_p = t_{io} + t_c$, где t_{io} — время, требуемое для завершения операций ввода-вывода, затребованных в p , а t_c представляет время вычислений — время для выполнения внутренних, не относящихся к вводу-выводу, операций в программе p .

Аппаратурные буфера часто используются как интерфейсы между основной памятью и периферийными устройствами ввода-вывода с фиксированными длинами записей, такими, как устройства печати и устройства чтения с перфокарт. Команда ввода (вывода) освобождает (наполняет) буфер, обеспечивая передачу информации между ним и основной памятью, и активизирует устройство для автоматического наполнения (освобождения) буфера в то время, когда работает программа. Поэтому устройство ввода (вывода) всегда работает с опережением (отставанием) на одну операцию ввода-вывода по отношению к выполнению программы. Такой буфер незаметен для программиста и позволяет параллельно работать электромеханическому периферийному устройству и ЦОУ. Если устройство занято наполнением или освобождением буфера и получен запрос на ввод-вывод, ЭВМ должна *ждать*, пока не закончится операция. Самый лучший результат, который можно получить при работе программы p , выполняющей, скажем, только ввод, и аппаратурного буфера, — это $t_p \approx \max(t_{io}, t_c) + t_b$, где t_{io} — общее время передачи от вводного устройства в буфер и t_b — общее время передачи между буфером и основной памятью. Программист должен тщательно разнести свои операции ввода-вывода в про-

грамм, чтобы они не шли подряд, и тем самым устраниТЬ ожидания процессора и достичь максимального перекрытия.

Значительно более качественное управление получается, если в аппаратуре есть программно-адресуемый *флажок занятости*. Это однобитовый регистр, который автоматически устанавливается, когда устройство ввода-вывода становится занятым, и сбрасывается, когда устройство освобождается. Для простой ЭВМ с аппаратурными буферами *in* и *out* команда ввода-вывода вызывает следующие *аппаратурные* действия.

Команда ввода

a: if deviceflag = 1 then go to a;
inarea := in;
deviceflag := 1;
Initiate Input Device; (Иницировать устройство ввода)

Команда вывода

a: if deviceflag = 1 then go to a;
out := outarea;
deviceflag := 1;
Initiate Output Device; (Иницировать устройство вывода)

где *inarea* и *outarea* — это области памяти для вводимых и выводимых записей. Так как флажок адресуемый, программист может проверять его и переходить на программы, не содержащие ввода-вывода, вместо того, чтобы ждать, когда устройство освободится; например,

if deviceflag = 1 then Computeonly;
IO instruction;

Однако, если должно быть получено существенное перекрытие этих двух частей, все равно необходимо тщательно организовывать чередование команд ввода-вывода и «только вычислительных» разделов программы.

2.3.2. Косвенный ввод-вывод

Почти все вычислительные системы имеют некоторую форму *косвенного* ввода-вывода, так что легко может быть достигнута одновременная работа центрального процессора и устройств ввода-вывода. Центральный процессор только инициирует

ввод-вывод и может затем перейти к выполнению следующей команды; фактически операцией управляет независимое устройство — канал или процессор ввода-вывода.

Команда ввода-вывода центрального процессора состоит из запроса в канал и может быть записана в форме:

startio(channelnumber, channelprogramarea)

где *channelnumber* определяет конкретный канал и *channelprogramarea* указывает адрес области основной памяти, содержащей программу или набор команд, которые канал выполняет для осуществления желаемой операции. Для операций чтения/записи программа канала определяет адрес основной памяти и объем данных, которые нужно переслать по этому адресу; она будет в общем случае также определять контроллер конкретного устройства и содержать приказы контроллеру выбрать и активизировать желаемое устройство ввода-вывода. Как канал, так и центральный процессор, работая параллельно, конкурируют за использование циклов ЗУ, причем канал обычно имеет высший приоритет, т. е. ЦОУ будет простоявать, пока не будет удовлетворен запрос на память со стороны канала. Если канал занят, когда выдано *startio*, то будем полагать, что ЦОУ *ждет*, пока канал не сможет принять запрос на ввод-вывод¹⁾.

Центральный процессор может в любое время опросить состояние канала. В памяти или в регистре имеется флагок, который установлен, когда канал занят, и сброшен, когда канал свободен. Мы будем использовать булеву переменную *busy[i]* для определения состояния занятости канала *i*; *busy[i] = true*, если канал *i* занят и *false* в противном случае. Доступна также и другая информация о состоянии канала, контроллера и устройства, например информация о наличии ошибок ввода-вывода или о том, является ли устройство занятым, или даже — подсоединенено ли оно. Однако здесь эту информацию мы рассматривать не будем.

Второй метод связи между каналом и центральным процессором основан на использовании прерываний. В этом случае канал будет прерывать ЦОУ при завершении операции ввода-вывода или при условии возникновения ошибки. Разрешенное прерывание, если оно возникло, обычно обрабатывается в конце цикла команды и приводит к принудительной передаче управления в некоторую фиксированную ячейку, адрес которой связан с причиной прерывания. Состояние машины непосредствен-

¹⁾ Такая реакция на занятость не нашла всеобщего применения. Чаще ЦОУ завершает выполнение *startio* во всех случаях без задержки, указывая специальными битами состояния причину, по которой операция ввода-вывода не была запущена.

но перед прерыванием (в минимальном объеме — это содержимое счетчика команд) автоматически сохраняется. Ниже приводится упрощенное описание аппаратурного цикла команды, включающего действия прерывания: M — основная память; ic — счетчик команд: $int_1, int_2, \dots, int_n$ — фиксированные ячейки в M , которым передается управление по прерываниям типа C_1, C_2, \dots, C_n соответственно. Предположим, что все прерывания разрешены.

```

Next:  $w := M[ic];$ 
         $ic := ic + 1;$ 
        Execute(w);
        if Interrupt then
            begin
                Store(ic);
                if cause = C1 then  $ic := int_1$ 
                else
                    if cause = C2 then  $ic := int_2$ 
                    else
                        .
                        .
                else  $ic := int_n$ 
            end;
            go to Next;
    
```

Для устранения излишних сложностей при последующем обсуждении алгоритмов буферизации, мы будем полагать, что команды чтения/записи имеют дело с записями *фиксированной длины* и последовательными устройствами, а также будем использовать *Read(ch, loc)* и *Write(ch, loc)* вместо команды *startio* и подробной программы канала, где ch — номер канала и loc — адрес первой ячейки основной памяти фиксированного блока, участвующего в передаче данных. Таким образом, *Read* и *Write* приводят, если это возможно, к *иницированию* команды центральным обрабатывающим устройством и к выполнению каналом операции ввода-вывода; ЦОУ освобождается немедленно после иницирования канала.

2.4. Программное обеспечение буферизации ввода-вывода

В последующих четырех подразделах детально разобрана серия примеров организации ввода-вывода, иллюстрирующих методы и проблемы буферизации ввода-вывода, параллельную

работу ЦОУ и канала, принципы обработки прерываний и разделение памяти несколькими программами.

2.4.1. ЦОУ опрашивает канал

Предположим, что ЦОУ опрашивает канал посредством использования флагка занятости и прерывания не используются.

Идеальный способ, который обеспечивал бы максимальное перекрытие между работой канала и ЦОУ, представляется последовательностью операторов:

... *Read(ch, loc); Compute(loc); ...*

Оператор *Compute(loc)* определяет только вычислительные (незванные с вводом-выводом) операции с блоком памяти *loc*. Вычисления с содержимым области ввода-вывода — *Compute(loc)* — начинаются сразу же после того, как канал запущен (инициирован). Серьезная проблема здесь состоит в том, что одна и та же область памяти одновременно изменяется как каналом, так и ЦОУ, которые имеют к ней доступ. В результате обычно возникает полный беспорядок — для ЦОУ при доступе в конкретную ячейку области ввода-вывода практически оказывается невозможным удостовериться, изменилось ли уже ее содержимое операцией *Read*. Этот недостаток можно устранить, изменив последовательность

```

    :
    :
    :
    Read(ch, loc);
loop: if busy[ch] then go to loop;
      Compute(loc);
    :
    :
```

Тогда время, необходимое для завершения любой программы, равно $t_p = t_{io} + t_c$ и параллельная работа ЦОУ и канала невозможна. Если всегда использовать этот метод, то нет никакого смысла иметь отдельный процессор для ввода-вывода.

Получить преимущества от использования канала, очевидно, можно путем имитирования аппаратурной буферизации. Пусть буфер ввода — это массив из n слов, *Bufin* [$1 : n$]. Следующая процедура *Get(Area)* оказывает такое же действие в системе, как и в случае аппаратурной буферизации и использования опе-

рации *read*, показанной в разд. 2.3.1:

```
procedure Get(Area);
integer array Area;
begin
  integer i;
  procedure MoveBufin;
    for i := 1 step 1 until n do Area[i] := Bufin[i];
  loop: if busy[ch] then go to loop;
  Move Bufin;
  Read(ch, Bufin)
end Get
```

В этом случае система всегда читает одну запись перед тем, как она потребуется. Типичная последовательность чтение/вычисление в программе теперь принимает вид:

```
...Get(loc); Compute(loc); ...
```

Первоначально *Bufin* должен быть загружен путем выполнения *Read(ch, Bufin)*. Подобная же процедура, *Put(Area)*, может быть написана для вывода. Время на передачу содержимого буфера в желаемую ячейку памяти (*MoveBufin* или, скажем, *MoveBufout*) составляет основные накладные расходы в стандартных программах *Get* и *Put*. Когда достигается максимальное перекрытие, время выполнения программы приблизительно равно $\max(t_c, t_{io}) + t_b$, где t_b — общее время, требуемое для передачи во входной и из выходного программных буферов, t_c — время, требуемое для выполнения *Compute(loc)*; $t_{io} = \max(t_{input}, t_{output})$, если используются отдельные каналы для ввода и вывода, $t_{io} = t_{input} + t_{output}$, если используется один канал.

Для устранения длительных буферных передач можно использовать несколько буферов и работать непосредственно в буферных областях. Пусть имеются два буфера *Buf[1]* и *Buf[2]*, каждый содержащий *n* слов. Когда операция чтения использует *Buf[1]*, операция вычисления может обращаться к *Buf[2]*, и наоборот. Буфера меняются ролями при завершении ввода-вывода. Мы изменяем стандартную программу *Get* таким образом, что при возврате ее аргумент будет содержать индекс буфера:

```
procedure Get(i);
integer i;
begin
  Read(ch, Buf[i]);
  i := if i = 1 then 2 else 1
end Get
```

Тогда типичная программа может оказаться такой:

```

    :
    :
comment инициализирует ввод-вывод и индекс буфера;
Read(ch, Buf[1]); i := 2;
    :
    :
comment основной цикл;
loop: Get(i); Compute(Buf[i]);
    go to loop;
    :
    :

```

Заметьте, что проверка занятости в стандартной программе *Get* не требуется, поскольку мы предположили, что ЦОУ будет «ждать» на команде ввода-вывода, пока не освободится канал, к которому он обращается. Время для чтения и обработки одной записи в приведенной выше программе равно $\max(t_{c1}, t_{rio})$, где t_{c1} и t_{rio} — времена (предполагаемые постоянными) для вычислительной обработки и считывания одной записи соответственно. Это лучший результат, который мы можем получить при работе с этим типом структуры программы. Рассмотренный метод называется *буферным сполингом*.

К сожалению, структуры программ редко бывают такими простыми. Чаще встречается последовательность выполнения, которая состоит из бурных «всплесков» ввода-вывода, за которыми следует много вычислений, а за ними снова следуют бурные «всплески» ввода-вывода и т. д. Такими часто бывают стандартные программы статистического анализа данных и языковые трансляторы. Программы этого типа могут иметь такую схему:

```

    :
    :
for k := 1 step 1 until n1 do
    begin Get(i); Compute(Buf[i]) end;
    Computeonly;
for k := 1 step 1 until n2 do
    begin Get(i); Compute(Buf[i]) end;
    Computeonly;
    :
    :
for k := 1 step 1 until nm do

```

```

begin Get(i); Compute(Buf[i]) end;
Computeonly;
:
:
```

где часть *Compute* в циклах относительно короткая, а программные секции *Computeonly* относительно длинные. В этом случае стандартные программы *Get* будут часто ожидать при операции *Read*, пока не будет завершено последнее считывание. Очевидным решением является добавление большего числа буферов и попытка распределить операции ввода-вывода более однородно по программе без нарушения структуры программы при определяющем считывании более чем одной записи.

2.4.2. Составные буферы и сопрограммная структура программ¹⁾

Пусть существует набор из n буферов, $Buf[0]$, $Buf[1]$, ..., ..., $Buf[n-1]$, ($n \geq 1$), явно связанных в циклический список, так что за $Buf[i]$ следует $Buf[(i+1) \bmod n]$, $i = 0, 1, \dots, n-1$. [Мы будем использовать обозначения $x +_n y$ вместо $(x+y) \bmod n$.] Основная программа будет запрашивать буфер путем вызова стандартной программы *GetBuf*, затем производить вычисления с буфером, возвращенным программой *GetBuf* (обозначим его, например, $Buf[current]$), и, наконец, освобождать $Buf[current]$ вызовом программы *ReleaseBuf*. Предполагается, что последовательность выполнения основной программы может быть представлена в следующей форме:

```

 $C_0; GetBuf; C_1(Buf[current]); ReleaseBuf;$ 
 $C_2; GetBuf; C_3(Buf[current]); ReleaseBuf;$ 
 $C_4; \dots$ 
 $\vdots$ 
 $C_{2n-2}; GetBuf; C_{2n-1}(Buf[current]); ReleaseBuf;$ 
 $C_{2n}; \dots$ 

```

где вычисления C_{2i} , $i = 0, 1, \dots$, не вызывают обращений к буферу. Если мы имеем дело только с операциями ввода, то *GetBuf* отыщет следующий заполненный при вводе буфер, а *ReleaseBuf* определит $Buf[current]$ как свободный (доступный для последующего ввода); в случае вывода *GetBuf* возвращает пустой или свободный буфер, а *ReleaseBuf* возвращает заполненный для вывода буфер.

¹⁾ При изложении материала этого раздела автор следовал Кнуту (1968) стр. 214—221.

Предположим, что определены только операции *ввода*. В то же время, когда выполняется главная программа (в дальнейшем называемая *Compute*), мы попытаемся сохранить занятость канала наполнением буферов. Это достигается посредством программы *ввода-вывода*, которая управляет каналом и работает в квазипараллельном режиме по отношению к программе *Compute*. Необходимы два указателя на буфера — один, *nextget*, который указывает на очередной доступный буфер для программы *Compute*, и другой, *nextio*, который указывает на очередной свободный буфер для программы ввода-вывода. Типичная ситуация показана на рис. 2.6(а). Здесь $n = 5$ и не было сделано ни одного запроса на буфер. Буфера *G* заполнены информацией опережающего ввода, в то время как буфера *R* пусты и доступны для операции чтения. На рис. 2.6(б) отображено состояние буфера после нормального выполнения *GetBuf*, а именно:

$$\text{current} := \text{nextget}; \quad \text{nextget} := \text{nextget} +_5 1;$$

Символ *C* обозначает текущий буфер. Операция *Read* обычно в это же время заполняла бы $\text{Buf}[\text{nextio}]$; когда *Read* завершается, существует новый буфер *G*, и мы имеем

$$\text{nextio} := \text{nextio} +_5 1;$$

Освобождение $\text{Buf}[\text{current}]$ по *ReleaseBuf* изменит тип буфера *C* на *R*. На рис. 2.6(с) показано новое состояние буферов после завершения как *Read*, так и *ReleaseBuf*.

Два указателя будут непрерывно «преследовать» друг друга по буферному циклу. Алгоритмы должны быть спроектированы аккуратно, так чтобы один указатель не догонял и не перегонял другой. Если *nextget* догоняет и опережает *nextio*, то буфера, найденные по *GetBuf*, больше не будут буферами, заполненными при вводе; это можно предотвратить, запрещая *GetBuf* переходить к обработке буфера, когда $\text{nextget} = \text{nextio}$. Говорят, что система является системой с *ограниченными вычислениями*, если вычислительная часть программы запрашивает ввод-вывод быстрее, чем часть ввода-вывода может выполнить его. Аналогично, если *nextio* догоняет и опережает *nextget*, то система будет перезаписывать буфера, заполненные при вводе, которые не были еще использованы, т. е. должен быть пустой буфер, доступный до того, как выдана команда ввода-вывода. Система называется системой с *ограниченным вводом-выводом*, если программа ввода-вывода производит ввод-вывод быстрее, чем вычислительная программа может потреблять его.

Программа *Compute* и программа *ввода-вывода* проектируются как почти независимые стандартные программы, которые в принципе работают параллельно. Мы моделируем этот

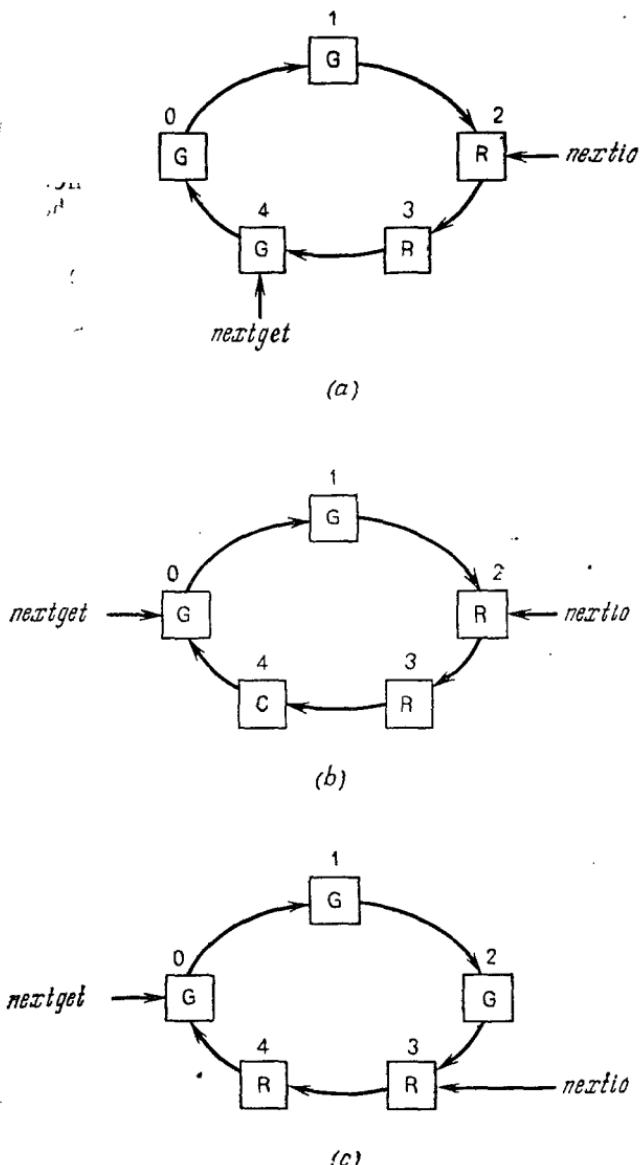


Рис. 2.6. Пять буферов в цикле: (а) — нет закрепленного буфера; (б) — после выполнения *GetBuf*; (с) — после выполнения *ReleaseBuf* и завершения ввода-вывода.

параллелизм посредством программной структуры, называемой *сопрограммой*. Сущность сопрограмм выясняется при сопоставлении их с более знакомыми подпрограммами. В структуре

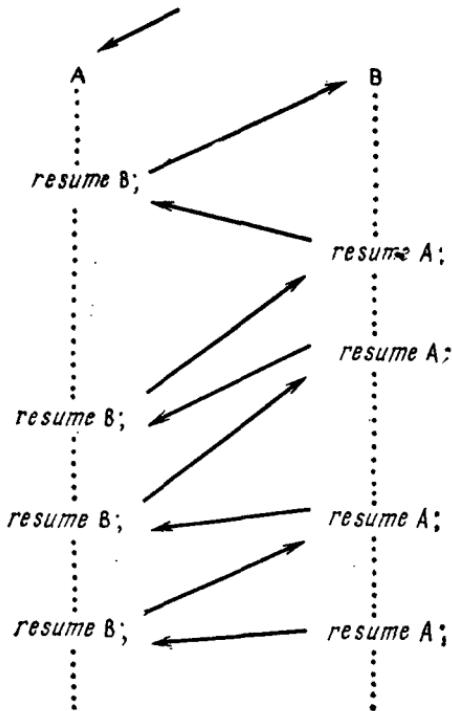


Рис. 2.7. Последовательность управления сопрограммами.

программ с *подпрограммами* существует несимметричное соотношение подчинения между вызывающей программой и ее подпрограммой; в общем случае в подпрограмму входят через одну из фиксированного числа точек входа (обычно имеется одна точка входа) и все переменные, которые не являются глобальными или параметрами, не определены при входе. С другой стороны, сопрограммы — это программы, которые могут вызывать друг друга, но между ними не определено отношение подчинения; отношение между ними симметричное, причем каждая сопрограмма является как ведущей, так и ведомой. При выходе из сопрограммы ее состояние запоминается; в следующий раз, когда сопрограмма вызывается, она продолжается точно с той точки, где она прервалась в предыдущий раз, причем все ее внутренние переменные остаются неизменными, т. е. предыдущее состояние восстанавливается. Обращение к сопрограмме *n* будет обозначаться как «*resume n*» (возобновить *n*) (Вегнер, 1968). Стрелки в примере на рис. 2.7 показывают передачи управления между двумя сопрограммами *A* и *B*, где *A* запускается первой. (Прежде чем сопрограмма может быть «возобновлена» в первый раз, ее состояние должно быть, конечно, инициализировано.)

Программа *Compute* (*CP*) и программа ввода-вывода (*IOP*) являются сопрограммами. Для поддержания как ЦОУ, так и канала в активном состоянии *CP* обычно возобновляет *IOP*, если канал ввода *ch* не занят, а *IOP* возобновляет *CP*, если канал занят. Кроме того, возобновления производятся, если не доступен буфер требуемого типа. Пусть *n* — общее число буферов и *r* — число буферов типа *R*. Две процедуры *GetBuf* и *ReleaseBuf* внутри *CP* есть:

```

procedure GetBuf;
begin
  c: if  $\neg$  busy[ch] then resume IOP;
  comment продолжить, если только имеется заполненный
        буфер ввода;
  if  $r = n$  then go to c;
  current := nextget;
  nextget := nextget + n 1
end GetBuf

procedure ReleaseBuf;
begin
  r := r + 1;
  if  $\neg$  busy[ch] then resume IOP
end ReleaseBuf

```

Наконец, повсюду в программе *CP* между предложениями *C_i*, *i* = 0, 1, 2, ... щедро расставляются предложения вида

if \neg busy[ch] then resume IOP;

Таким образом, мы стремимся поддерживать канал занятым, когда это логически возможно. Сопрограмма *IOP* есть:

```

io1: resume CP;
  comment продолжить только в том случае, если
        имеется пустой буфер;
io2: if  $r = 0$  then go to io1;
  Read(ch, Buf[nextio]);
  resume CP;
  nextio := nextio + n 1;
  r := r - 1;
  go to io2;

```

Первоначально тип всех буферов устанавливается в *R* (*r* = *n*), а как *nextget*, так и *nextio* имеют в качестве значения один и тот же индекс буфера, скажем 0. Если мы буферируем

вывод, можно использовать те же программы: *Read* в *IOP* заменяется на *Write*; в этом случае первоначально тип всех буферов устанавливают в $G(r = 0)$, так как стандартная программа *GetBuf* теперь будет выбирать пустые буферы, а *ReleaseBuf* будет возвращать заполненные буферы.

Какую величину должно иметь число буферов n ? Пусть t_{ci} — время, требуемое для вычисления C_i , $i = 0, 1, \dots$, в *CP*, и пусть t_{rio} — время ввода-вывода одной записи. Если $t_{c2i-1} + t_{c2i} < t_{rio}$ для всех i или $t_{c2i-1} + t_{c2i} > t_{rio}$ для всех i , $n = 2$ позволяет добиться максимального перекрытия операций ЦОУ и канала (времена выполнения *GetBuf* и *ReleaseBuf* игнорируются). Это сразу сводится к ситуации буферного свопинга, описанного ранее. Действительно, $n = 1$ могло бы отвечать требованиям, если $t_{c2i-1} + t_{c2i}$ всегда или значительно меньше или значительно больше, чем t_{rio} . Только когда суммы $t_{c2i-1} + t_{c2i}$, т. е. времена между последовательными *GetBuf* разбросаны по обе стороны от t_{rio} , $n > 2$ становится выгодным.

Система многократной буферизации, которую мы здесь представили, предполагает, что главная программа имеет особый вид во время ее выполнения, а также предполагает наличие связанного с ней набора буферов. Хотя это, конечно, приемлемо для многих программ, встречаются и другие способы организации. Для более эффективного использования вспомогательной памяти и программ ввода-вывода логические записи блокируются в большие физические записи. Мы можем тогда рассматривать операцию ввода-вывода или как заполнение нескольких буферов, по одной логической записи на буфер, или как заполнение одного буфера несколькими логическими записями. В последнем случае *GetBuf* и *ReleaseBuf* должны быть расширены для работы как внутри буферов, так и между буферами. Основной программе не нужно освобождать буфер прежде, чем она затребует другой, — на самом деле запросы на буфер и освобождения часто перемежаются произвольным образом. Для управления этой возможностью должны быть установлены явные связи между буферами, так как буферы будут освобождаться не обязательно в порядке, в котором они были предоставлены по запросам. Часто несколькими достаточно независимыми процессами ввода-вывода и вычислительными процессами используется «пул» буферов (пример этого кратко рассмотрен ниже). Наконец, мы должны заметить, что простой временной анализ не учитывает эффект захвата цикла во время одновременной работы центрального процессора и канала. Это проявляется в увеличении t_{ci} . Несмотря на то что рассмотренные методы, основанные на опросе процессором канала, применяются в некоторых системах и полезны для объяснения

принципов буферизации, наиболее общие подходы используют преимущества прерываний и избегают большинства вызовов типа `resume` (возобновление).

Упражнения

1. Докажите, что сопрограммы *IOP* и *CP* правильно взаимодействуют в достижении желаемой системы буферизации при $n \geq 1$. Как минимум, необходимо показать, что:

a) система начинает работать правильно независимо от того, какая из сопрограмм получает управление первой;

b) невозможно навсегда зациклиться и для *GetBuf* и для *IOP* при не-
получении полного или пустого буфера соответственно;

c) процедуры *GetBuf* и *IOP* всегда используют очередной заполненный
при вводе буфер и очередной пустой буфер соответственно.

(Для рассмотрения методов доказательства правильности и конечности про-
цесса выполнения программ отсылаем читателя к работе Элспаса и др., 1972.)

2. Рассмотрите замену двух предложений в *GetBuf*, начиная с метки *c*, на

If $r = n\theta \neg \neg busy[ch]$ then resume *IOP*:

где или $\theta = \vee$ или $\theta = \wedge$.

Приведут ли эти замены к получению правильной программы? Если первые два предложения из *IOP* заменить на

io2: If $r = 0$ then resume *CP*;

будет ли все еще *IOP* работать?

З Перепишите сопрограммы *IOP* и *CP* так, чтобы устраниТЬ все метки и
предложения *go to*. Предположите, что АЛГОЛ расширен включением пред-
ложения *while* вида

while *BE* do *S*;

где *BE* — булево выражение и *S* — предложение; это новое предложение эк-
вивалентно программе

w: If *BE* then begin *S*; go to *w* end;

где *w* — уникальная локальная метка.

4. Пусть каждая физическая запись состоит из *m* логических записей и
буфера обозначенны *Buf[i, j]*, $i = 1, \dots, n$; $j = 1, \dots, m$ ($m, n \geq 1$). Пред-
положим, что операция *Read(ch, Buf[i, *])* будет читать физическую запись
в *Buf[i, 1], Buf[i, 2], \dots, Buf[i, m]*. Сделайте необходимые изменения в
основной и *IOP* сопрограммах для обработки блокированных записей; т. е.
GetBuf и *ReleaseBuf* должны по-прежнему возвращать логические записи.

2.4.3. Канал прерывает ЦОУ

Предполагается, что канал будет прерывать ЦОУ при завер-
шении операции ввода-вывода; возможные прерывания от дру-
гих причин не будут рассматриваться. Прерывания допускают
значительно более эффективный контроль над параллель-
ностью работы ввода-вывода и ЦОУ и почти устраняют необ-
ходимость проверок занятости канала. Мы далее предполагаем,
что последовательность выполнения основной программы может
быть представлена в такой же форме, как и приведенная

в начале последнего раздела, и используется та же самая многобуферная структура данных. Сначала рассмотрен случай только с операциями *ввода*.

Тогда процедуры *GetBuf* и *ReleaseBuf* основной программы приобретают вид:

```
procedure GetBuf;
begin
  loop: if r = n then go to loop;
         current := nextget;
         nextget := nextget +n 1
  end

procedure ReleaseBuf;
begin
  r := r + 1;
  if  $\neg$  busy[ch] then Read(ch, Buf[nextio])
end
```

Здесь необходима проверка \neg busy[ch], так как перед вызовом *ReleaseBuf* значение *r* может быть равным 0.

При прерывании по завершении ввода-вывода выполняется следующая программа (*IR*):

```
nextio := nextio +n 1;
r := r - 1;
if r  $\neq$  0 then Read(ch, Buf[nextio]);
Restore state;
```

Когда происходит прерывание, предполагается, что состояние прерванной программы сохраняется аппаратурой; *Restore state* восстанавливает состояние ЭВМ, соответствующее прерванному процессу. Мы инициализируем систему с помощью следующей последовательности:

```
r := n;
nextget := nextio = 0;
Read(ch, Buf[nextio]);
```

Основная программа и стандартная программа прерываний (*IR*) независимы. Следует заметить, что операции ввода-вывода инициируются автоматически *всякий раз*, когда это логически возможно сделать.

Что неправильно в приведенных выше программах? Возникает новая проблема из-за асинхронной природы прерывания — оно может случиться в *любой* точке во время выполнения ос-

новной программы. Предположим, что прерывание происходит в некоторый момент во время выполнения предложения « $r := r + 1;$ » в *ReleaseBuf*. В результате компиляции этого предложения для одиоадресной ЭВМ с одним сумматором могут получиться три команды:

- (1) $LD \ r$
- ← I_1
- (2) $ADD \ "1"$
- ← I_2
- (3) $STO \ r$

Если предположить, что $r = 5$ на шаге (1), то прерывание в I_1 привело бы к следующему эффекту:

1. Команда (1) устанавливает $AC = 5$ (AC — сумматор).
2. Вторая команда в IR устанавливает $r = 4$.
3. Состояние основной программы восстанавливается ($AC = 5$).
4. Команды (2) и (3) устанавливают $r = 5 + 1 = 6$.

Тогда мы имеем $r = 6$ вместо $r = 5$. Та же самая ошибка возникает, если основная программа была прервана в I_2 . Это пример часто встречающейся проблемы, когда некоторым процессам разрешено одновременно иметь доступ и изменять общую область данных (в данном случае переменную r). В общем случае мы должны сделать такую секцию машинной программы совершенно неделимой¹⁾. (В следующей главе эта проблема рассматривается более подробно.)

Один из способов предотвращения упомянутых выше условий «состязания» — установить связь между *ReleaseBuf* и IR . Процедура *ReleaseBuf* даст IR знать, если прерывание произошло во время выполнения ее *критической секции* „ $r := r + 1;$ ” и, аналогично, IR будет сообщать процедуре *ReleaseBuf* о прерываниях. Пусть b и $flag$ — булевые переменные с исходными значениями, равными *false*. Заменим „ $r := r + 1;$ ” в *ReleaseBuf* на предложения

```
b := true;
r := r + 1;
b := false;
if flag then r := r - 1;
flag := false;
```

¹⁾ Пример, который мы выбрали, исключительно прост, так как на некоторых машинах имеются отдельные (неделимые) команды, которые инкрементируют содержимое ячеек памяти; использование такой команды для реализации „ $r := r + 1;$ ” устранило бы данную проблему. Однако наша цель — продемонстрировать проблему в простейшем возможном случае.

Заменим „ $r := r - 1$; if $r \neq 0$ then $Read(ch, Buf[nextio]);$ ” в IR на

```

if  $b$  then  $flag := true$ 
else
begin
   $r := r - 1;$ 
  if  $r \neq 0$  then  $Read(ch, Buf[nextio])$ 
end;

```

Если $ReleaseBuf$ прервана в своей критической секции, то b будет $true$ и IR и не уменьшит r , и не произведет считывания, а установит $flag$; тогда $ReleaseBuf$ проверит $flag$ и, если $flag$ был $true$, уменьшит r для IR . Прерывание вне этой критической секции обнаружит, что $b = false$, и будет выполнена нормальная обработка. Предполагается, что переменная b может быть установлена за одну машинную операцию.

Следует отметить, что наше решение допускает не более одного прерывания в критической секции. Если операции ввода-вывода очень быстрые или, что более реально, если основная программа может быть приостановлена в течение длительных периодов времени другими процессами, то может оказаться желательным разрешить в этом месте несколько прерываний.

Упражнение

Почему может произойти самое большое только одно прерывание в критической секции $ReleaseBuf$, когда $ReleaseBuf$ и IR модифицированы, как показано выше? Измените алгоритмы так, чтобы в IR новые операции ввода-вывода всегда инициировались до тех пор, пока имеется доступный буфер.

2.4.4. Объединение буферов в пул для ввода и вывода

Пусть мы используем набор из $n \geq 2$ буферов одинакового размера как *пул* (разделяемый ресурс) для удовлетворения запросов на пустые, заполненные при вводе или заполненные для вывода буфера. В главной программе теперь будут разрешены запросы как на ввод, так и на вывод. Преимуществом разделения буферов перед использованием отдельных буферных областей для ввода и вывода является экономия основной памяти; при разумно спланированных запросах на буфер n разделяемых буферов могут быть так же эффективны, как использование отдельных областей из n буферов каждая. (В более общих системах объединение буферов удобно для управления диском или абонентским пунктом.)

Для ввода основная программа с помощью вызова *Gbufin* будет получать очередной буфер (*curin*), заполненный при

вводе, и освобождать буфер с помощью *Rbufin*. Последовательность вывода будет состоять из вызова *Gbufout*, чтобы отыскать пустой буфер (*curout*), за которым позднее следует вызов *Rbufout*. По вызову буфер, заполненный для вывода, возвращается в пул для распечатки процессом вывода. Временная последовательность выполнения основной программы предполагается следующей:

C₀; *Gbufx₁*; *C₁(Buf[y₁])*; *Rbufx₁*; *C₂*; *Gbufx₂*; *C_{3(Buf[y₂])}*;
Rbufx₂; *C₄*; ...; *Gbufx_n*; *C_{2n-1(Buf[y_n])}*; *Rbufx_n*; ...

где

- 1) *x_i* — это либо „*in*”, либо „*out*”.
- 2) Если *x_i* = „*in*”, то *y_i* = „*curin*”; если *x_i* = „*out*”, то *y_i* = „*curout*”.
- 3) *C_i* имеет тот же смысл, что и в последних двух разделах.

Программы ввода-вывода логически выполняются параллельно с основной программой и обычно активизируются прерываниями по завершении ввода-вывода. Процесс ввода будет получать пустой буфер (*inio*) из пула и возвращать его как заполненный при вводе; процесс вывода будет искать в пуле очередной буфер, заполненный для вывода (*outio*), и возвращать его как пустой после того, как операция вывода завершена (рис. 2.8). Ввод и вывод будут происходить по отдельным выделенным каналам *ch 1* и *ch 2* соответственно.

Необходимо идентифицировать три различных типа буферов, которые могут быть в пуле — *empty* (пустой), *input-full* (заполненный при вводе) и *output-full* (заполненный для вывода). Поскольку три процесса получают и возвращают буферы почти асинхронным способом, наиболее удобна структура данных со связанным списком; буферы одного типа в пуле при этом связаны в цепочку. Пусть *F[em]*, *F[in]* и *F[out]* указывают в пуле на первый буфер типа *empty*, *input-full*, *output-full* соответственно, и пусть *L[em]*, *L[in]*, *L[out]* — указатели на последний буфер каждого типа соответственно. С каждым из *n* буферов мы связываем указатель вперед *F[i]*, который связывает буфер *i* со следующим буфером в цепочке того же типа, что *i*. Буфер

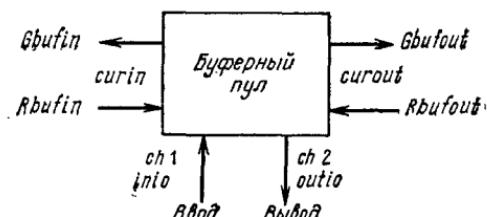


Рис. 2.8. Разделение буферов для ввода и вывода.

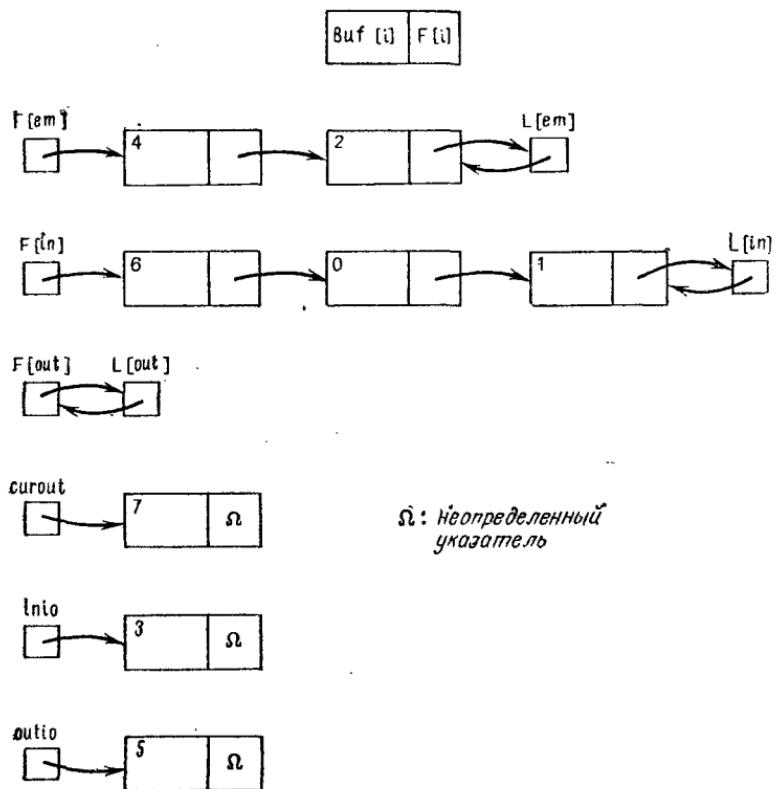


Рис. 2.9. Возможная конфигурация буферного пула.

возвращается в пул добавлением к концу соответствующего списка и отыскивается просмотром от начала списка; т. е. списки являются очередями с дисциплиной обслуживания типа «первым пришел — первым обслужен» (FIFO). На рис. 2.9 показан типичный буферный пул ($n = 8$) во время выполнения программы в некоторой точке в $C_{2k-1}(Buf[curout])$. Процедура C_{2k-1} заполняет $Buf[7]$ для дальнейшего вывода, причем канал 1 заполняет $Buf[3]$ входными данными, а канал 2 опорожняет $Buf[5]$. Два буфера (4 и 2) пусты, три занесены в список типа *input-full* (6,0 и 1), а список типа *output-full* пуст.

Манипуляция с буфером выполняется двумя общими стандартными программами, $AddBuf(type, number)$ и $TakeBuf(type)$, где $type \in \{em, in, out\}$ и $0 \leqslant number \leqslant n - 1$. Программа $AddBuf$ добавляет $Buf[number]$ в конец списка класса *type*, тогда как функция $TakeBuf$ удаляет первый буфер из списка и возвращает его индекс в качестве значения функции. Программа

**Таблица 2.1. Переменные, используемые
в программах объединения буферов в пул**

Переменная .	Значение
n	Число буферов; $n \geq 2$
$Buf[i]$	i -й буфер в пуле
$curin$	Текущий буфер ввода есть $Buf[curin]$
$curout$	Текущий буфер вывода есть $Buf[curout]$
$inio$	Буфер, используемый операцией ввода, есть $Buf[inio]$
$outio$	Буфер, используемый операцией вывода, есть $Buf[outio]$
em	Обозначение буферов типа <i>empty</i>
in	Обозначение буферов типа <i>input-full</i>
out	Обозначение буферов типа <i>output-full</i>
$F[t]$	Указатель первого буфера класса t , $t \in \{em, in, out\}$
$L[t]$	Указатель последнего буфера класса t , $t \in \{em, in, out\}$
eb	Число буферов типа <i>empty</i>
i/b	Число буферов типа <i>input-full</i>
o/b	Число буферов типа <i>output-full</i>

AddBuf выглядит следующим образом:

```

procedure AddBuf(type, number);
value type, number; integer type, number;
begin
  integer oldlast;
  oldlast := L[type];
  F[oldlast] := number;
  F[number] := type;
  L[type] := number
end AddBuf

```

Пусть $|x|$ — число буферов в списке типа x ; пусть $|em| = eb$ (пустые буферы), $|in| = i/b$ (буферы, заполняемые при вводе) и $|out| = o/b$ (буферы, заполненные для вывода). Сначала $eb = n$ и $i/b = o/b = 0$. Это завершает спецификацию всех переменных, которые мы будем использовать; в табл. 2.1 перечисляются все переменные и их значения.

Процедуры, вызываемые основной программой:

```

procedure Gbufin;
begin
  while ifb = 0 do
    if  $\neg$  busy[ch1] then startin; 1)
    ifb := ifb - 1;
    curin := TakeBuf(in)
  end Gbufin

procedure Rbufin;
begin
  eb := eb + 1;
  AddBuf(em, curin);
  if  $\neg$  busy[ch] then startin
end Rbufin

procedure Gbufout;
begin
  while eb = 0 do
    if  $\neg$  busy[ch2] then startout;
    eb := eb - 1;
    curout := TakeBuf(em)
  end Gbufout

procedure Rbufout;
begin
  ofb := ofb + 1;
  AddBuf(out, curout);
  if  $\neg$  busy[ch2] then startout
end Rbufout

```

Дополнительные процедуры *startin* и *startout* инициируют операции ввода-вывода:

```

procedure starin;
if eb  $\neq$  0 then
begin
  eb := eb - 1;
  inio := TakeBuf(em);

```

¹⁾ См. упражнение 3 в конце разд. 2.4.2 для определения этой формы оператора *while*,

```

Read(ch1, Buf[inio])
end startin

procedure startout;
if ofb ≠ 0 then
begin
    ofb := ofb - 1;
    outio := TakeBuf(out);
    Write(ch2, Buf[outio])
end startout

```

По мере возникновения прерываний при завершении ввода и вывода управление передается программам

ch1 Interrupt(input) (IR1)

```

ifb := ifb + 1;
AddBuf(in, inio);
startin;
Restore state;

```

ch2 Interrupt(output) (IR2)

```

eb := eb + 1;
AddBuf(em, outio);
startout;
if  $\neg busy[ch1]$  then startin;
Restore state;

```

Проблема критической секции, обсуждавшаяся в предыдущем разделе, здесь более серьезна, но она до сих пор игнорировалась для простоты. При тщательном изучении рассмотренных программ обнаруживается еще одна проблема, которая часто возникает в системах разделения ресурсов. Что происходит, если все буфера (разделяемый ресурс) заполнены при вводе и выдан вызов *Gbufout?* Тогда $eb = ofb = 0$, а первое предложение *Gbufout* навсегда застывает! Чтобы продолжаться далее, *Gbufout* должен иметь пустой буфер, но создатели пустого буфера, т. е. *Rbufin* и *IR2*, не могут быть вызваны, если не продолжается *Gbufout*; мы находимся в безвыходном положении. Такого рода ситуацию обычно называют *тупиком* или, более выразительно, «смертельным объятием». Эта проблема в общем виде будет обсуждаться в гл. 8 (однако рассмотрите упражнение 3).

Должны ли быть запрещены прерывания ввода (вывода) во время выполнения *IR2*(*IR1*)? До тех пор пока мы будем заботиться о состояниях критических секций и стеков, с тем чтобы

„Restore state” восстанавливала правильное состояние машины, прерывания в пределах программ обработки прерываний допустимы. Однако общепринятый подход, особенно для коротких программ, состоит в том, чтобы маскировать другие прерывания и таким образом устраниить эти сложности.

Мы заканчиваем раздел некоторым негативным замечанием, касающимся ясности, целенаправленности и общности программных конструкций, представленных в этих двух последних разделах. Исключительно трудно проследить логику этих программ и убедиться в их правильности из-за параллельного развития процессов в ЦОУ и канале и асинхронной природы прерываний ввода-вывода; например, не ясно, необходима ли проверка занятости ввода-вывода в *Gbufout?* (Как вы полагаете?)

Обсуждение

Читателю легко увязнуть в деталях наших программ и упустить из вида основные моменты. Кроме иллюстрации некоторых устоявшихся методов управления вводом-выводом и буферами, предыдущие примеры представляют конкретные случаи нескольких более общих концепций и проблем, встречающихся в операционных системах.

Логический и физический параллелизм среди процессов является основной характеристикой современных систем. Фактически все системы имеют аппаратурные процессоры, которые могут работать параллельно; наиболее общим примером являются ЦОУ и каналы. Параллельная работа программ на этих процессорах может обеспечиваться или методами опроса, например, когда программы на ЦОУ опрашивают канал, или в более общем случае через прерывания. Часто также бывает полезным рассматривать выполнение нескольких программ как логически параллельные виды деятельности, даже несмотря на то, что они могут в действительности выполняться последовательно на единственном процессоре по способу мультиплексирования процессора во времени. Этот тип параллелизма может быть в общем случае смоделирован программно с использованием сопрограмм, как показано в разд. 2.4.2, или же могут быть использованы аппаратурные прерывания для инициирования или возобновления логически параллельных процессов, таких, как программа *IR* в разд. 2.4.3.

Параллельные процессы часто нуждаются в разделении одного и того же набора ресурсов, такого, как буфера в наших примерах. И тогда становится необходимым тщательно регулировать и синхронизировать это разделение для правильной работы; проблемы критической секции и тупика, введенные в последних двух примерах, иллюстрируют некоторые осложнения

и ловушки при использовании общих ресурсов. Программы объединения буферов в пулы показывают, в частности, необходимость лучшего способа описания параллельных работ и их взаимодействий.

Мы будем изучать все эти вопросы более подробно. Алгоритмы буферизации ввода-вывода этой главы будут при случае использованы как примеры в последующих главах.

Упражнения

1. Напишите алголиную процедуру целого типа *TakeBuf(type)*, которая в результате работы возвращает индекс первого буфера в список *type* и обновляет структуру данных в буфере, чтобы отразить это перемещение.

2. Измените алгоритмы объединения буферов в пул, чтобы обеспечить правильное управление всеми критическими секциями. В этом контексте должна быть также рассмотрена одновременная манипуляция связанными списками со стороны нескольких процессов.

3. Измените алгоритмы объединения буферов в пул, чтобы предотвратить туникую ситуацию по отношению к ресурсу пустого буфера. Может ли она быть предотвращена, если число буферов $n = 1$?

4. *Буферизация ввода-вывода при форматировании текста*. Целью этого упражнения по программированию является применение техники сопрограмм и объединения буферов в пул к ситуации типа «ввод — вычисление — вывод», в которой ЦОУ опрашивает каналы. Составьте и реализуйте программу, которая принимает с перфокарт текст в свободном формате, снабженный редактирующими символами, и вырабатывает соответствующий форматированный текст на устройстве построчной печати. Расширенные версии этого типа программ могут часто упростить написание книг, статей, писем.

Программа должна состоять из трех сопрограмм: сопрограммы ввода, которая заполняет буферы неформатированным текстом; форматирующей сопрограммы, которая создает выходные буфера с форматированным текстом, и программы вывода, которая управляет печатью.

Общий пул памяти из n ($n \geq 2$) буферов, каждый размером 80 байтов, используется всеми программами для трех различных типов буфера (рис. 2.10(а)).

Входным потоком в систему является *набор заданий*, организованных, как показано на рис. 2.10(б):

Карта задания находится в начале каждого задания и состоит из элементов:

<i>/FORMAT</i>	колонки 1—7
<i><job id></i>	колоики 10—20, идентификатор задания

Последняя карта каждого задания есть индикатор конца задания:

<i>/FINISHED</i>	колонки 1—9
------------------	-------------

Часть колоды с данными в каждом задании состоит из текстовых карт, содержащих символы из набора $S \cup \{b\} \cup \{\#, \$, *\}$, где $S = \{A, B, \dots, Z, ., ;, ?, .., "\}$, а b обозначает пробел.

Форматирующая программа должна редактировать ввод в соответствии со следующими правилами.

Пусть x — текущий входной символ.

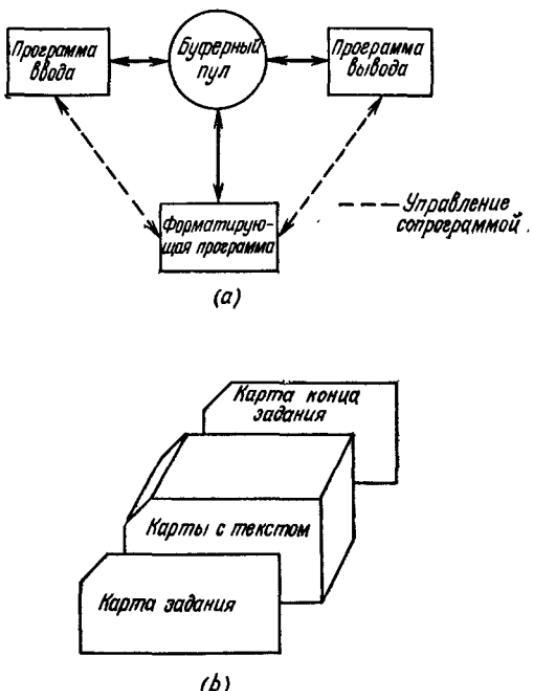


Рис. 2.10. Упражнение 4: (a) — пути программ и данных; (b) — формат колоды задания.

- 1) Если $x \in S$, то x есть очередной выходной символ (за исключением случая 6 ниже).
- 2) Если $x = \#$, то начать новую строку вывода.
- 3) Если $x = \$$, то начать новый параграф вывода.
- 4) Если $x = *$, то следующая строка пробелов не сокращается.
- 5) Строки вида $x_1 b b^n x_2$, где $x_1 \notin \{b, *, \#, \$\}$, $x_2 \neq b$, и $n \geq 1$ сокращаются до $x_1 x_2$, т. е. лишние пробелы удаляются. Если $x_1 \in \{\#, \$\}$, пробелы устраняются после того, как выполнены действия, заданные символом x_1 .
- 6) Любая входная строка вида $b x_1 x_2 \dots x_n b$, $x_i \in S$ должна быть отредактирована так, чтобы $x_1 x_2 \dots x_n$ появлялись только в одной выходной строке; она не может пересекать границы строк.

Обработка карты задания состоит из вывода карты на печать и инициализации таймера задания. Текст карты конца задания печатается непосредственно и сопровождается указанием, сколько времени выполнялось задание. При завершении работы ваша программа должна напечатать суммарное время выполнения (время ЦОУ), ввода и вывода. Изменяя число буферов n от 2 до 10, определите, как это скажется на общем времени выполнения. Программа должна проверяться на реальном объеме данных (скажем, 50 заданий более чем с 400 картами текста).

Если нет вычислительной системы, которая позволяет пользователю легко запрашивать карту и таймер, то эти средства могут имитироваться с помощью следующих спецификаций. Пусть операции ввода и вывода производятся на

отдельных специально предназначенных для этого каналах; каждая операция имеет дело с записями фиксированной длины по 80 байтов. Предположим, что время (моделируемое) для каждой задачи есть:

Задача	Единицы времени
Читать карту	2 (ввод-вывод)
Печатать строку	2 (ввод-вывод)
Обработать карту задания	1 (ЦОУ)
Обработать карту конца задания	1 (ЦОУ)
Интерпретировать \$ в карте текста	1 (ЦОУ)
Интерпретировать # в карте текста	1 (ЦОУ)
Интерпретировать * в карте текста	1 (ЦОУ)
Обработать карту текста (в дополнение к приведенным выше)	1 (ЦОУ)
Обработка конца строки	1 (ЦОУ)

Проверки занятости ввода-вывода в вычислительной программе должны быть размещены в конце всех задач, за исключением первых двух, в дополнение к тем местам, где они логически необходимы. Чтобы отразить завершения задач при этих проверках занятости ввода-вывода, инкрементируйте моделируемый аппаратный таймер.

2.5. Супервизор ввода-вывода

Наибольшая часть системы пакетной обработки, выполняющей по одному заданию в каждый момент времени, предназначена для организации производительной работы ввода-вывода. Этот компонент назван *системой управления вводом-выводом* (IOCS), или *супервизором ввода-вывода*. Важным побочным следствием этого центрального управления было введение *символических* устройств ввода-вывода. Вместо определения абсолютного адреса физического устройства в операции ввода-вывода пользователь ссылается на устройство ввода-вывода символически; операция ввода-вывода могла бы иметь вид

<имя операции> (<символическое имя устройства>, <другая информация>)

где *<имя операции>* может быть, скажем, *READ*, *WRITE*, *TEST* или *CONTROL*; например, *READ(CARDREADER, INPUTAREA)*. Таблица соответствия символьических имен физическим устройствам ввода-вывода содержится в памяти. Система может теперь создавать и изменять назначения физических устройств, чтобы отражать конкретную конфигурацию оборудования, проводить восстановление при временных отказах аппаратуры или повышать эффективность. Например, описанный выше

CARDREADER мог бы быть устройством чтения с перфокарт в одной конфигурации, ленточным устройством — в другой (когда используется преобразование «карта — лента» в автономном режиме) или дисковым устройством (если все данные периферийного ввода-вывода проходят через дисковую память).

Система IOCS будет выполнять следующие функции:

1. *Управление буфером.* Система IOCS будет обеспечивать общую и личную буферную память и динамически регистрировать все назначения. Сюда входят в общем случае стандартные программы работы с буферами, описанные ранее (например, различные версии *Rbuf* и *Gbuf*).

2. *Трансляция запросов на ввод-вывод.* Символические имена устройств сопоставляются действительным адресам, и создается соответствующая последовательность команд программы канала.

3. *Планирование ввода-вывода.* Могут возникать одновременно несколько запросов на канал или устройство. Это может случиться, если канал назначен единственному устройству, если один канал разделяется несколькими устройствами, если несколько каналов могут разделять устройства или если взаимное влияние канала и ЦОУ при доступе к основной памяти накладывает ограничения на число одновременных операций ввода-вывода. Запросы должны быть поставлены в очередь, а программа планирования канала определяет очередную операцию ввода-вывода, которую нужно инициировать.

4. *Управление прерыванием.* Обслуживаются прерывания, вызываемые завершениями ввода-вывода и ошибками ввода-вывода. Это обслуживание обычно заключается в обновлении структур буферных данных и в вызове планировщика канала для инициирования следующей операции ввода-вывода, как было показано в упрощенном виде в наших предыдущих примерах.

3. Взаимодействующие процессы

Канальный процесс, выполняющий операцию ввода-вывода параллельно с развитием¹⁾ «чисто» вычислительного процесса в ЦОУ, является реальным примером параллельной работы в вычислительной системе. Часто и естественно встречаются как *аппаратный параллелизм* — параллельная работа нескольких аппаратных обрабатывающих устройств, так и *логический параллелизм* — концептуальный параллелизм, введенный человеком независимо от того, происходит ли обработка последовательно или параллельно. Мультипрограммная операционная система вместе с выполняющимися в ней заданиями пользователя может быть логически описана как набор *последовательных* процессов, которые действуют почти независимо один от другого, *кооперируются*²⁾ друг с другом путем пересылки сообщений и синхронизирующих сигналов и *состязается* за использование ресурсов. В этой главе мы рассмотрим механизмы описания параллелизма, взаимосвязи и состязания последовательных процессов и приведем примеры, иллюстрирующие их применение. Основные концепции взаимодействия процессов были впервые разработаны Дейкстрой и появились в его основополагающей монографии „Cooperating Sequential Processes” (Дейкстра, 1965а; 1968б).

3.1. Параллельное программирование

3.1.1. Применения

Увеличение эффективных скоростей ЭВМ может быть достигнуто путем усовершенствования технологии изготовления ее компонентов, а также архитектуры самой машины. При определенной технологии параллельная работа аппаратных устройств в принципе может резко улучшить производительность системы по сравнению с последовательной организацией

¹⁾ В оригинале „execution”, т. е. буквально «выполнение». В данном случае термин «развитие» более точно соответствует сущности понятия «процесс». — *Прим. перев.*

²⁾ В смысле образуют систему посредством установления всевозможного рода связей между процессами. — *Прим. перев.*

действий. Несколько независимых процессоров часто бывают связаны с общей памятью и управляющими устройствами. Это центральные процессоры, процессоры ввода-вывода, такие, как каналы данных, и процессоры специального назначения, такие, как арифметические устройства. В таком случае выполняются параллельно и посыпать сообщения друг другу могут несколько программ и частей отдельной программы. Если аппаратный параллелизм минимален, то часто бывает выгодно разделять время единственного обрабатывающего устройства между несколькими процессами; полезной абстракцией является рассмотрение этих процессов как процессов, протекающих параллельно. Сначала мы будем исследовать «естественный» параллелизм, который существует в нескольких очень простых алгоритмах, и будем игнорировать проблему коммуникаций между процессами. Нашей основной целью является введение посредством простых и знакомых примеров некоторых элементарных понятий параллельного программирования.

Рис. 3.1 показывает типы отношения предшествования, которые возможны между процессами, если мы полагаем, что система имеет одну начальную и одну конечную точку. Развитие процесса p_i представляется направленной дугой графа. Каждый граф на рисунке обозначает трассу развития во времени набора процессов и связно описывает отношения предшествования процессов. Для удобства эти графы будут называться *графами развития процесса*. Все компоненты в последовательном/параллельном примере являются правильно вложенными. Пусть $S(a, b)$ обозначает последовательную связь процесса a с процессом b , и пусть $P(a, b)$ обозначает параллельную связь процессов a и b . Тогда граф развития процесса является *правильно вложенным*, если он может быть описан функциями S и P или только композицией этих функций¹⁾.

Примеры

Первые три графа на рисунке могут быть описаны так:

$$S(p_1, S(p_2, S(p_3, p_4))),$$

$$P(p_1, P(p_2, P(p_3, p_4))),$$

$$S(p_1, S(P(p_2, P(S(p_3, P(p_4, p_5)), p_6)), P(p_7, p_8)))$$

Общий граф предшествования (d) на рисунке не является правильно вложенным. Мы доказываем это, прежде всего указывая, что любое описание, сделанное с помощью функциональ-

¹⁾ Это свойство очень похоже на свойство «правильной вложенности» блочной структуры в языках программирования и скобок в выражениях.

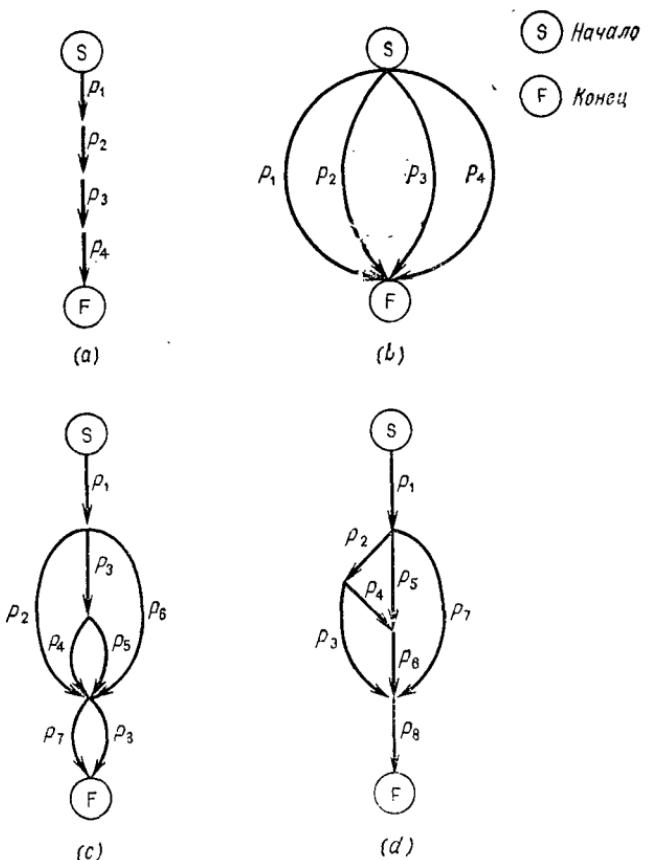


Рис. 3.1. Отношения предшествования между процессами: (а) — последовательное; (б) — параллельное; (с) — последовательно-параллельное; (д) — ощий случай предшествования.

ной композиции, должно включать на самом внутреннем уровне выражение или в форме $S(p_i, p_j)$ или $P(p_i, p_j)$ для $p_i, p_j \in \{p_k | k = 1, \dots, 8\}$. Связь $P(p_i, p_j)$ не может появиться, поскольку рис. 3.1(д) не содержит ни одного подграфа в этой форме. Все последовательно связанные p_i и p_j имеют по крайней мере один процесс p_k , который начинается или заканчивается в вершине ij , скажем между p_i и p_j ; но ij становится недоступной для дальнейшего использования, если появляется $S(p_i, p_j)$, так как тогда связь p_k не может быть описана. Следовательно, $S(p_i, p_j)$ также не может быть использована, и описание правильно вложенного графа невозможно.

Следующие три примера параллелизма приводят к правильному вложенным графикам развития процесса:

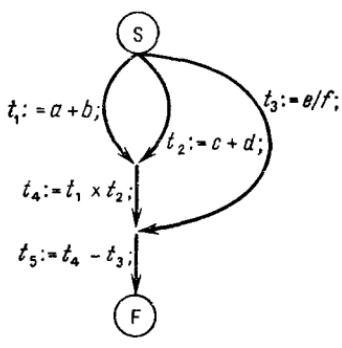
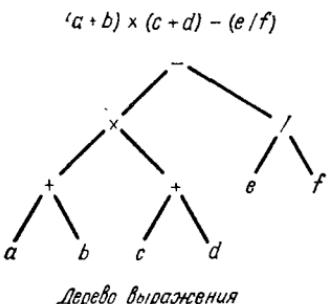


Рис. 3.2. Вычисление арифметического выражения.

1. Вычисление арифметических выражений. Если побочные эффекты или числовая точность не диктуют фиксированную последовательность вычисления, то подвыражения арифметических выражений могут вычисляться параллельно; степень параллелизма, который может иметь место, ограничена глубиной дерева выражения (см. рис. 3.2, где ребра помечены кодом программы, выполняемой соответствующим процессом). Многие проблемы, в которых основная структура данных является деревом, могут быть логически описаны в терминах параллельных вычислений.

2. Сортировка. Во время i -го прохода в стандартной сортировке слиянием второго порядка пары сортируемых списков длины 2^{i-1} сливаются в списки длины 2^i ; причем все слияния в пределах одного прохода могут быть выполнены параллельно (рис. 3.3).

3. Умножение матриц. При выполнении умножения матриц, $A = B \times C$, все элементы A могут быть вычислены одновременно.

Упражнение

Рассмотрите арифметические выражения E , содержащие только простые переменные и бинарный оператор $+$, т. е.

$$E ::= v \mid E + v$$

$$v ::= a_1 \mid a_2 \mid a_3 \mid \dots$$

Предположим, что может быть выполнено параллельно любое число операций сложения и что каждая такая операция занимает одну единицу времени, включая выборку двух операндов и запись результата. Каково минимальное время, требуемое для вычисления выражения $a_1 + a_2 + \dots + a_n$, $n \geq 1$, как функции n ? Докажите ваше решение.

3.1.2. Некоторые программные конструкции для параллелизма

Нотация „and“

Вирт (1966b) предложил простое дополнение к АЛГОЛу, которое позволяет программисту указывать, что предложения могут выполняться параллельно. Параллельное выполнение опре-

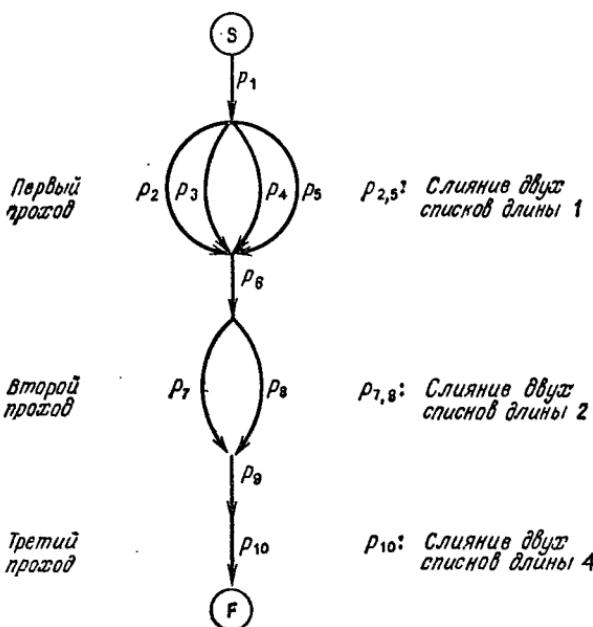


Рис. 3.3. Сортировка слиянием списка из восьми элементов.

деляется заменой точки с запятой, отделяющей предложение, на символ **and**. Например, предложениями программы для вычисления выражения по рис. 3.2 являются

```

begin
  t1 := a + b and t2 := c + d;
  t4 := t1 × t2
end
and t3 := e/f; t5 := t4 - t3;

```

Параллельные вычисления могут быть созданы условно и динамически, как показано в следующей программе, которая вычисляет параллельно каждый элемент матричного произведения $A = B \times C$ (Вирт, 1966 б):

```

integer array A[1:m, 1:n], B[1:m, 1:p], C[1:p, 1:n];
procedure product(i, j);
value i, j; integer i, j;
begin
  integer k; real s;
  s := 0;

```

```

for  $k := 1$  step 1 until  $p$  do
     $s := s + B[i, k] \times C[k, j];$ 
     $A[i, j] := s$ 
end product;

procedure column( $i, j$ );
    value  $i, j$ ; integer  $i, j$ ;
    product( $i, j$ ) and
        if  $j > 1$  then column( $i, j - 1$ );
procedure row( $i$ );
    value  $i$ ; integer  $i$ ;
    column( $i, n$ ) and
        if  $i > 1$  then row( $i - 1$ );
    row( $m$ )

```

Создание параллельных процессов графически показано на рис. 3.4 для случая $m = n = p = 2$.

Пусть процесс p_i в граfe развития процесса описывается последовательностью программных предложений S_i . Тогда любой правильно вложенный граfe развития может быть описан

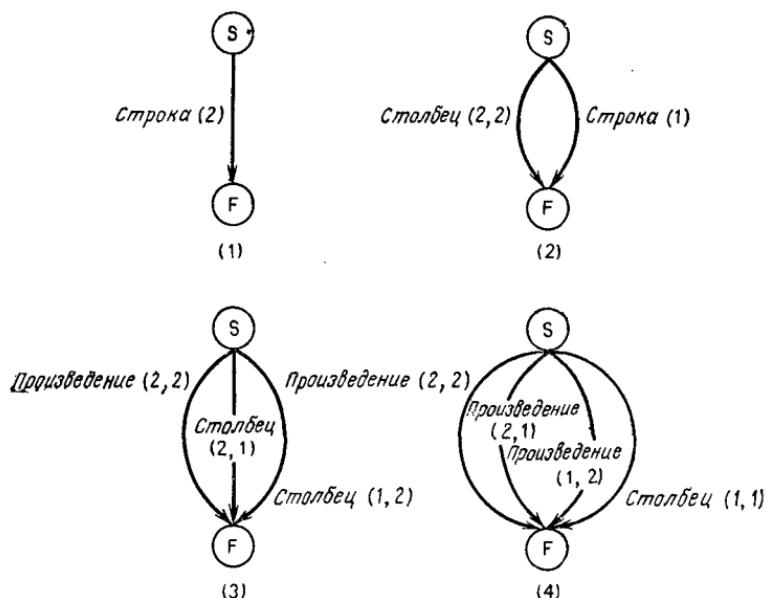


Рис. 3.4. Возникновение параллельных процессов при умножении матриц ($m = n = p = 2$).

линейно с помощью нотации **and**: связь $S(p_i, p_j)$ преобразует в „**begin** S_i ; S_j **end**”, связь $P(p_i, p_j)$ преобразуется в „**begin and** S_i **end**”. Однако нотации **and** недостаточно для описан произвольного графа развития процесса.

Примитивы Fork, Join и Quit

Примитивы **fork**, **join** и **quit** (Конвей, 1963; Деннис и В Хори, 1966) обеспечивают более общие средства для линейно описания параллельной работы в программе.

Выполнение процессом p команды „**fork** w ” вызывает нача развития нового процесса q с команды, помеченной w ; затем и q развиваются одновременно. Если процесс p выполня комманду „**quit**”, то он заканчивается. Команда „**join** t, w ” вызывает следующее действие:

```
t := t - 1;
if t = 0 then go to w;
```

Область, ограниченная пунктирной линией, рассматривает как *неделимое* действие.

Программный сегмент для вычисления выражения рис. 3.

```
n := 2;
fork p3;
m := 2;
fork p2;
t1 := a + b; join m, p4; quit;
p2: t2 := c + d; join m, p4; quit;
p4: t4 := t1 × t2; join n, p5; quit;
p3: t3 := e/f; join n, p5; quit;
p5: t5 := t4 - t3;
```

Это решение, конечно, менее ясно, чем программа последне раздела. Однако при использовании примитивов могут быть б лее четко описаны итерации, как показано на следующем пр мере из области обработки изображений.

Нам дан массив $A[0:n+1, 0:n+1]$, состоящий из нуле и единиц, представляющих преобразованное в цифровую форм изображение. Требуется «сгладить» изображение заменой как дой внутренней точки $A[i, j]$ единицей, если большинство зна чений из $A[i, j]$ и ее 8 ближайших соседей являются 1, или нулем в противном случае. Этот процесс называется *локальны усреднением* и логически является параллельным вычислением

Мы полагаем, что сглаженное изображение запоминается в новом массиве.

```

procedure Localaverage(A, B, n);
value n; integer array A, B; integer n;
begin
  integer t, l, j; private i, j;
  t := n ↑ 2;
  for i := 1 step 1 until n do
    for j := 1 step 1 until n do
      fork e;
      quit;
e: B[i, j] := if A[i - 1, j - 1] + A[i - 1, j] + A[i - 1, j + 1]
               + A[i, j - 1] + A[i, j] + A[i, j + 1] + A[i + 1, j - 1]
               + A[i + 1, j] + A[i + 1, j + 1] ≥ 5 then 1 else 0;
      join t, r;
      quit;
r: end Localaverage

```

Чтобы создать личные копии переменных внутри параллельных процессов, переменные могут быть объявлены «личными» для процесса с помощью декларации „**private** x_1, x_2, \dots, x_n ”. Тогда переменные x_1, x_2, \dots, x_n существуют только для процесса, выполняющего декларации **private**; кроме того, любой новый процесс, созданный процессом с декларацией **private** (с использованием **fork**), будет получать личную копию личных переменных процесса-отца. Еще один пример: три предложения, начинающиеся с метки *e* в программе *Localaverage*, могут быть заменены на

```

e: begin private integer u, v, x;
      x := 0;
      for u := -1 step 1 until 1 do
        for v := -1 step 1 until 1 do
          x := x + A[i + u, j + v];
        B[l, j] := if x ≥ 5 then 1 else 0;
      join t, r;
      quit;
end;

```

Предложений **fork**, **join** и **quit** достаточно для описания графа развития любого процесса. Ниже дана программа для графа рис. 3.1(d), где S_i обозначает программные предложения для

процесса p_i :

```
t6 := 2; t8 := 3;
S1; fork w2; fork w5; fork w7; quit;
w2: S2; fork w3; fork w4; quit;
w5: S5; join t6, w6; quit;
w7: S7; join t8, w8; quit;
w3: S3; join t8, w8; quit;
w4: S4; join t6, w6; quit;
w6: S6; join t8, w8; quit;
w8: S8; quit;
```

Параллельные вычисления в примерах этого раздела слишком малы, чтобы иметь практическое значение для операционных систем. Однако введенные конструкции программирования полезны для описания параллельных работ в ОС и будут использованы в последующих разделах.

Упражнения

- Покажите, что основной граф предшествования на рис. 3.1 не может быть описан с помощью нотации **and**.
 - Напишите алгольную процедуру, использующую нотацию **and** для параллельного вычисления $\sum_{i=1}^n a_i b_i$ двух векторов.
 - Пусть набор T деревьев простых арифметических выражений определяется следующим образом:
 - $x \in T$ для любой простой переменной или константы x ,
 - если $t_1 \in T$ и $t_2 \in T$, то $\theta \in T$ для любого бинарного оператора θ
- $$\begin{array}{c} \theta \\ / \quad \backslash \\ t_1 \quad t_2 \end{array}$$
- Операторы имеют обычную интерпретацию. Напишите алгольную процедуру, используя **and** для вычисления выражения, представленного любым $t \in T$. (Выберите удобное представление в ЭВМ для t .)
- Почему большая часть операции **join** должна быть неделимой?
 - Напишите программы для параллельного матричного умножения, используя **fork**, **join**, **quit** и **private**.
 - Используйте **fork**, **join**, **quit** и **private** для программирования слияния второго порядка с параллельным слиянием при каждом проходе.

3.2. Концепция процесса

Мы использовали термин «процесс» достаточно произвольно, полагая, что читатель интуитивно понимает его значение. Общепринятое, но неформальное определение процесса таково:

Последовательный процесс (иногда называемый «задача») есть работа, производимая последовательным процессором при выполнении программы с ее данными.

С логической точки зрения каждый процесс имеет свой собственный процессор и программу. В действительности, два различных процесса могут разделять одну и ту же программу или один и тот же процессор. Например, в программе умножения матриц в разд. 3.1.2 n^2 процессов используют одну и ту же программу $product(i, j)$. Таким образом, процесс не эквивалентен программе, а также это не то же самое, что процессор; это пара \langle процессор, программа \rangle при выполнении.

Развитие процесса может быть описано как последовательность векторов состояния s_0, s_1, \dots, s_i , где каждый вектор состояния s_i содержит указатель на следующую программную команду, которую нужно выполнить, а также значения всех промежуточных и определяемых в программе переменных. С другой точки зрения вектор состояния процесса p есть та информация, которая требуется процессору, чтобы направлять развитие процесса p . Вектор состояния процесса p может быть изменен или при развитии p или при развитии других процессов, которые *разделяют* некоторые компоненты вектора состояния с p .

Взаимосвязи между процессами и управление их работой будут происходить с помощью установки общих переменных и специальных базовых операций процессов (примитивов), которые будут определены в дальнейшем. Если мы рассмотрим процесс в любой момент времени, то он будет или *продолжающимся* или *блокированным*. Процесс p является (логически) продолжающимся, если он или выполняется процессором или мог бы выполняться процессором, если бы процессор был доступен; в последнем случае мы часто будем говорить, что процесс находится в состоянии *готовности*. Процесс p блокирован, если он не может протекать, пока не получит сигнал, сообщение или ресурс от некоторого другого процесса.

[Иногда используются более точные и формальные определения понятия процесса. В одном из крайних случаев делается попытка расширить область определения, чтобы охватить разнообразие общих процессов в аппаратуре, такой, как таймер, ЦОУ или механическое читающее устройство с перфокарт, а также, возможно, даже человеческие процессы, например работу оператора ЭВМ, следующего набору предписанных правил для управления системой. Другой подход — формализовать определение, чтобы упростить теоретический анализ какого-нибудь ограниченного аспекта взаимодействующих процессов. Примерами являются общие определения Хорнинга и Рэнделла (1973) и модель Холта (1971а), описанная в гл. 8.]

За счет рассмотрения только логики процессов и игнорирования числа доступных физических процессоров мы можем обеспечить процессорно-независимые решения для ряда системных

проблем; т. е. эти решения будут гарантировать нам, что система процессов составляется правильно вне зависимости о том, разделяют процессы физические процессоры или нет. Концепция процесса имеет несколько других важных применений в операционных системах (ОС). Она допустила выделение и определение многих базовых задач ОС, упростила изучение их организации и динамики и привела к развитию полезных методологий проектирования. Все предыдущие пункты обсуждаются далее в последующих разделах и главах, а одной из главных тем является взаимодействие процессов. Процессы будут определяться их программами, а нотация *and* будет использована для описания параллелизма.

3.3. Проблема критической секции

3.3.1. Проблема

Когда несколько процессов могут асинхронно изменять содержимое общей области данных, необходимо защитить данные от одновременного доступа и изменения двумя или более процессами. Обновляемая область в общем случае может и не содержать предполагаемых изменений, если защита не обеспечена. Эта ситуация была показана на некоторых примерах буферизации ввода-вывода в последней главе. Общие данные, разделяемые несколькими процессами, наиболее часто описываются как *ресурс*; обновление данных соответствует распределению или освобождению элементов ресурса.

Рассмотрим два процесса p_1 и p_2 , увеличивающих асинхронно значение общей переменной x , представляющей число единиц ресурса:

$p_1: \dots x := x + 1; \dots$

$p_2: \dots x := x + 1; \dots$

Пусть C_1 и C_2 — центральные процессоры с внутренними регистрами $R1$ и $R2$ соответственно и разделяемой основной памятью. Если бы p_1 выполнялся на C_1 и p_2 — на C_2 , то могла бы возникнуть любая из следующих двух последовательностей исполнения во времени:

(1) $p_1: R1 := x; R1 := R1 + 1; x := R1; \dots \dots \dots$

$p_2: \dots \dots \dots R2 := x; R2 := R2 + 1; x := R2;$

$|t_0 \rightarrow \text{time}|$

$|t_n|$

(2) $p_1: R1 := x; R1 := R1 + 1; x := R1; \dots \dots \dots$

$p_2: \dots \dots \dots R2 := x; R2 := R2 + 1; x = R2;$

Пусть x содержит значение v в момент t_0 . В момент t_n переменная x содержала бы $v + 1$, если бы выполнение на процессорах C_1 и C_2 следовало согласно (1), содержала бы $v + 2$, если бы выполнение следовало согласно (2). Оба значения x могли бы также быть реализованы, если бы p_1 и p_2 разделяли во времени единственный процессор с переключением управления между процессами посредством прерываний. Если бы процессы p_1 и p_2 вызывались как часть системы резервирования билетов на авиалиниях, а значение x представляло число доступных мест на конкретном рейсе, то пассажиры и администрация были бы очень огорчены, если бы переменная x имела бы значение $v + 1$ вместо $v + 2$. Аналогично, если бы переменная x представляла значение числа блоков памяти, распределяемой для пользовательских заданий в МС, то описанные выше возможности оказались бы нетерпимыми. Ясно, что должно учитываться *каждое* приращение x . Решение заключается в разрешении входить в любой момент времени в *критическую секцию* (critical section — CS), $x := x + 1;$ только одному процессу. В общем случае критическая секция могла бы состоять из любого числа предложений, например манипуляций с буфером в разд. 2.4.4.

Теперь может быть определена более точно проблема и вопросы, имеющие к ней отношение. Нам дано несколько последовательных процессоров, которые могут связываться друг с другом через общую память для хранения данных. Каждая из программ, выполняемых процессорами, содержит критическую секцию, в которой организован доступ к общим данным; эти программы предполагаются циклическими. Проблема заключается в том, чтобы запрограммировать процессоры так, чтобы в любой момент только один из процессоров находился в своей критической секции; если, скажем, процессор C входит в свою CS, то никакой другой процессор не может сделать то же самое до тех пор, пока C не покинет свою CS. Относительно системы сделаны следующие предположения:

1. Считывание из общей памяти и запись в нее есть неделимые операции; одновременные обращения (на запись или на считывание) к одной и той же ячейке памяти более чем одного процессора приведут к последовательным обращениям в неизвестном порядке.
2. Критические секции не могут иметь связанных с ними приоритетов.
3. Относительные скорости процессоров неизвестны.
4. Программа может останавливаться вне ее CS.

Проблема может также быть сформулирована в терминах нескольких процессов, асинхронно разделяющих во времени единственный процессор. Удобно забыть о числе физических

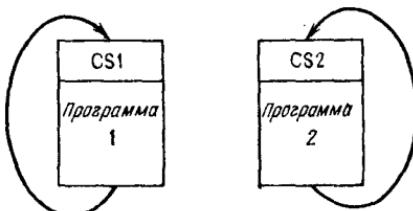


Рис. 3.5. Два процесса с критически-ми секциями.

процессоров и думать только о наборе (почти) независимых *процессов*, каждый из которых имеет некоторую CS.

Предполагается, что система циклических процессов для проблемы критической области имеет следующие программные формы:

```

begin
P1: begin L1: CS1; program1; go to L1 end
      and
P2: begin L2: CS2; program2; go to L2 end
      and
      :
      :
      and
Pn: begin Ln: CSn; programn; go to Ln end
end;

```

где CS_i есть критическая секция для процесса i , предложение в каждой строке P_i есть программа для процесса P_i и $n \geq 2$.

3.3.2. Программное решение (Дейкстра, 1965а, 1968б)

Проблема будет сначала ограничена двумя процессами (рис. 3.5). Нашей начальной целью является предохранение процессов P_1 и P_2 от одновременного нахождения в соответствующих им CS (*взаимное исключение*). В то же время должны быть устранены два возможных типа блокировки:

1. Процесс, нормально работающий вне своей CS, не может блокировать другой процесс при входении последнего в свою CS.
2. Два процесса, готовые войти в свои критические секции, не могут откладывать неопределенно долго решение о том, который из них действительно войдет в CS первым, руководствуясь принципом: «после вас — после вас»¹⁾.

¹⁾ Принцип «сверхежливости», который в равной степени можно назвать и принципом «Манилова — Чичикова». — Прим. ред.

Теперь мы попытаемся разработать решения проблемы и показать некоторые существующие ловушки.

Проблема легко решается, если мы потребуем, чтобы процессы P_1 и P_2 входили в свои CS попеременно; одна общая переменная может хранить указатель того, чья очередь войти в CS:

```
begin integer turn; turn:=2;
  P1:   begin L1: if turn=2 then go to L1;
        CS1; turn:=2;
        program1; go to L1;
        end and
  P2:   begin L2: if turn=1 then go to L2;
        CS2; turn:=1;
        program2; go to L2
        end
end;
```

Однако если бы P_1 начал развитие на более быстром процессоре, чем P_2 , или если бы *program1* была много длиннее, чем *program2*, или если бы P_1 остановился в *program1*, то это решение едва ли было бы удовлетворительным. Один процесс вне своей CS может помешать другому при его вхождении в свою CS.

Делается попытка устранить это возможное блокирование с помощью двух общих переменных C_1 и C_2 , как флагов для указания, находится ли процесс внутри или вне своей CS.

```
begin Boolean C1, C2; C1:=C2:=true;
  P1:   begin L1: if  $\neg C_2$  then go to L1;
        C1:=false; CS1;
        C1:=true; program1;
        go to L1
        end and
  P2:   begin L2: if  $\neg C_1$  then go to L2;
        C2:=false; CS2;
        C2:=true; program2;
        go to L2
        end
end;
```

Когда значение C_1 или C_2 равно **false** (**true**), то соответствующий процесс находится внутри (вне) своей критической секции. Взаимное блокирование теперь невозможно, но оба про-

цесса могут войти в свои CS вместе; последнее может случиться, так как обе программы могут вместе достичь точек L1 и L2 при $C1 = C2 = \text{true}$.

Взаимное выполнение¹⁾ последнего примера устраняется установкой значения C1 и C2, равного false, в L1 и L2 соответственно:

```
begin Boolean C1, C2; C1:=C2:=true;
P1: begin A1: C1:=false;
        L1: if  $\neg C2$  then go to L1;
        CS1; C1:=true;
        program1; go to A1
    end and
P2: .. (подобно P1)
end;
```

Последняя трудность разрешена, но теперь снова стало возможным взаимное блокирование. Переменной C1 может быть присвоено значение false в A1 в то же самое время, когда C2 будет присвоено значение false в A2; в этом случае и P1 и P2 будут циклиться неопределенно долго в предложениях с метками L1 и L2. Очевидным способом устранения этого является присваивание значения true переменным C1 и C2 после проверки, имеют ли они значение false в L1 и L2.

```
begin Boolean C1, C2; C1:=C2:=true;
P1: begin L1: C1:=false;
        if  $\neg C2$  then begin C1:=true; go to L1
            end;
        CS1; C1:=true;
        program1; go to L1
    end and
P2: ...
end;
```

К сожалению, это решение все еще может привести к такому же типу блокировки, как и в последнем примере; если оба процесса находятся точно на шаге с метками L1 и L2 и их скорости в точности одинаковы для каждой выполняющейся команды, то будет развиваться такой же цикл, как и прежде относительно L1 и L2.

¹⁾ Одновременное выполнение действий различными процессами, относящимися к критическим областям каждого из процессов. — Прим. перев.

Описанные выше попытки иллюстрируют некоторые тонкости, лежащие в основе этой проблемы. Следующее решение было впервые предложено математиком Т. Деккером:

```

begin integer turn; Boolean C1, C2;
C1 := C2 := true; turn := 1;
P1: begin A1: C1 := false;
        L1: if  $\neg C2$  then
            begin if turn = 1 then go to L1;
                  C1 := true;
                  B1: if turn = 2 then go to B1;
                  go to A1
            end
        CS1; turn := 2;
        C1 := true; program1;
        go to A1
    end and
P2: begin A2: C2 := false;
        L2: if  $\neg C1$  then
            begin if turn = 2 then go to L2;
                  C2 := true;
                  B2: if turn = 1 then go to B2;
                  go to A2
            end
        CS2; turn := 1;
        C2 := true; program2;
        go to A2
    end
end;

```

Переменные $C1$ и $C2$ гарантируют, что взаимное выполнение не может иметь места; переменная $turn$ гарантирует от взаимного блокирования. Взаимное выполнение невозможно, так как значение любого C_i ($i = 1$ или 2) есть всегда `false`, когда P_i запрашивает разрешения вступить в свою CS под меткой L_i . Взаимное блокирование невозможно, так как переменная $turn$ не изменяет значения во время выполнения программы принятия решения, предшествующей CS. Предположим, например, что $turn = 1$ и оба процесса пытаются войти в свою CS. Тогда P_1 может только циклиться на двух предложениях под меткой L_1 , причем значение $C1$ остается постоянным и равным `false`;

аналогично, P_2 может только циклиться в B_2 с постоянным значением C_2 , равным `true`. Но последнее подразумевает, что процесс P_1 получит разрешение войти в L_1 . Так как никакое другое зацикливание в этом случае невозможно, то такое решение предотвращает взаимное блокирование. Решение остается правильным, если P_1 и P_2 продолжались бы на одном и том же процессоре, и процессор переключается между P_1 и P_2 через конечный промежуток времени.

Упражнения

1. В качестве решения проблемы критической секции была предложена следующая программа. Предотвращает ли она взаимное выполнение и блокирование?

```

begin integer ask, inuse;
ask := inuse := 0;
P1: begin L1: if ask ≠ 0 then go to L1;
      ask := 1;
      if ask ≠ 1 then go to L1;
      if inuse ≠ 0 then go to L1;
      inuse := 1;
      if ask ≠ 1 then go to L1;
      CS1; ask := inuse := 0;
      program1; go to L1
end
P2: begin L2: if ask ≠ 0 then go to L2;
      ask := 2;
      if ask ≠ 2 then go to L2;
      if inuse ≠ 0 then go to L2;
      inuse := 1;
      if ask ≠ 2 then go to L2;
      CS2; ask := inuse := 0;
      program2; go to L2
end
end;

```

2. Решение Деккером проблемы CS предохраняет от взаимного выполнения и взаимного блокирования. Но возможно ли, что один из процессов мог быть навсегда блокирован, зациклившись через A_i , L_i и B_i , в то время как другой процесс непрерывно развивается в своей CS? (Вспомните, что относительные скорости обработки P_1 и P_2 известны: следовательно, возможна любая скорость для каждого процесса.)

3. Выведите программное решение для проблемы критической секции для n процессов ($n \geq 2$), работающих параллельно (Дейкстра, 1965b; Эйзенберг и Мак-Гайер, 1972).

3.4. Семафорные примитивы

3.4.1. P- и V-операции

Программное решение проблемы критической секции имеет два отрицательных качества:

1. Решение мистифицированное и неясное; простое концептуальное требование на выполнение взаимного исключения в критических секциях приводит к сложным и неудобным дополнениям программы.

2. В течение времени, когда один процесс находится в своей CS, другой процесс может непрерывно циклиться и при этом иметь доступ и проверять общие переменные. Чтобы сделать это, ожидающий процессор должен «красть» циклы памяти у активного процессора; если процессы разделяют единственный процессор, то ожидающий процесс тратит ценное время ЦОУ, на самом деле ничего реального не выполняя. Результатом является общее замедление системы процессами, которые не выполняют никакой полезной работы.

Дейкстра (1965а) ввел два новых примитива, которые значительно упростили взаимосвязь и синхронизацию процессов. В абстрактной форме эти примитивы, обозначаемые P и V , оперируют над *неотрицательными целыми переменными*, называемыми *семафорами*. Пусть S — такой семафор. Операции определяются следующим образом:

1. Операции $V(S)$: переменная S увеличивается на 1 (инкремент) одним *неделимым* действием; выборка, инкремент и запоминание не могут быть прерваны, и к S нет доступа другим процессам во время операции.

2. Операция $P(S)$: уменьшение S на 1, если возможно. Если $S = 0$, то невозможно уменьшить S и остаться в области целых неотрицательных значений; тогда процесс, вызывающий P -операцию, *ждет*, пока это уменьшение не станет возможным. Успешная проверка и уменьшение S — также неделимая операция. (Мы оставляем до разд. 3.5 обсуждение того, как происходит ожидание процесса, когда $S = 0$.)

Если несколько процессов одновременно запрашивают P - или V -операции над одним и тем же семафором, то эти операции будут выполняться последовательно в произвольном порядке; аналогично, если более одного процесса ожидает при выполнении P -операции и изменяемый семафор становится положительным, то конкретный ожидающий процесс, который выбирался для завершения операции, произволен и неизвестен.

Именно семафорные переменные используются для синхро-

низации процессов. P -примитив заключает в себе потенциальное ожидание вызывающих процессов, в то время как V -примитив может, возможно, активизировать некоторый ожидающий процесс. Неделимость P и V гарантирует целостность значений семафоров.

3.4.2. Взаимное исключение с помощью семафорных операций

Семафорные операции дают простое и непосредственное решение проблемы критической секции. Пусть $mutex$ — семафор; $mutex$ используется ниже для защиты CS. Программным решением проблемы для n процессов, работающих параллельно, является:

```

begin semaphore mutex;
    mutex := 1;
P1: begin ... end and
P2: ...
.
.
.
Pi: begin Li: P(mutex); CSi; V(mutex);
    programi; go to Li
end and
.
.
.
Pn: ...
end;

```

Значение $mutex$ равно 0, когда некоторый процесс находится в своей CS; иначе $mutex = 1$. Семафор выполняет функцию простого замка. Взаимное исключение гарантировано, так как только один процесс может уменьшить $mutex$ до нуля с помощью P -операции; все другие процессы, пытающиеся войти в свои критические секции, когда $mutex = 0$, будут вынуждены ожидать по $P(mutex)$. Взаимное блокирование невозможно, так как одновременные попытки войти в CS, когда $mutex = 1$, должны по нашему определению в разд. 3.4.1 преобразоваться в *последовательные* P -операции. Когда семафор может принимать только значение 0 или 1, он будет называться *двоичным* семафором.

3.4.3. Семафоры как счетчики ресурсов и синхронизаторы в проблемах производителя¹⁾ и потребителя

Каждый процесс в вычислительной системе может быть охарактеризован числом и типом ресурсов, которые он *потребляет* (использует) и *производит* (освобождает). Это могут быть «твёрдые» ресурсы, такие, как основная память, лентопротяжные механизмы или процессоры, или «мягкие» ресурсы, такие, как заполненные буферы, критические секции, сообщения или задания. Семафоры могут быть использованы для учета ресурсов и для синхронизации процессов, а также для запирания критических секций. Например, процесс может блокироваться *P*-операцией на семафоре *S* и может быть разблокирован другим процессом, выполняющим *V(S)*:

```

begin semaphore S; S:=0;
P1: begin ...
        comment Ждать сигнала от P2;
        P(S); ...
        end and
P2: begin
        comment Послать сигнал побудки P1;
        V(S); ...
        end
end;

```

Процесс *P1* можно также рассматривать как потребляющий ресурс, обозначенный *S*, посредством команды *P(S)*, в то время как *P2* производит единицы ресурса *S* посредством команды *V(S)*. В этом разделе мы иллюстрируем использование семафоров в типичных применениях производителя — потребителя ресурсов.

Пример 1: Два процесса, связанные через буферную память (Дейкстра, 1965a, 1968)

Процесс-производитель вырабатывает информацию и затем добавляет ее в буферную память; параллельно с этим процесс-потребитель удаляет информацию из буферной памяти и затем перерабатывает ее. Это обобщение ситуации, где, например, основной процесс будет вырабатывать выходную запись и затем передавать ее в очереди доступный буфер, в то время как

¹⁾ В литературе употребляется также термин «поставщик». Например, в кн. Д. Цикритзис, Ф. Бернстайн. Операционные системы. — М.: «Мир», 1977. — Прим. перев.

процесс вывода удаляет запись асинхронно из буферной памяти и затем печатает ее. Пусть буферная память состоит из N буферов одинакового размера, причем каждый буфер может хранить одну запись.

Будем использовать два семафора в качестве счетчиков ресурсов:

e — число пустых буферов

и

f — число заполненных буферов.

Предположим, что добавление к буферу и изъятие из него образуют критические секции; пусть b — двоичный семафор, используемый для взаимного исключения. Тогда процессы могут быть описаны следующим образом:

```

begin semaphore e, f, b; e:=N; f:=0; b:=1;
producer: begin Lp: produce next record; P(e);
               P(b); add to buffer; V(b); V(f); go to Lp
           end and
consumer: begin Lc: P(f); P(b); take from buffer; V(b);
               V(e); process record; go to Lc
           end
end;

```

Операции увеличения e и f должны быть неделимыми, иначе значения этих переменных могут стать неправильными; можно было бы дополнительно обращаться с изменениями e и f как с критическими секциями, но должна быть включена некоторая добавочная логика, с тем чтобы позаботиться о возможных ожиданиях при P -операциях (см. конец этого раздела для получения более детальных сведений). Описанное взаимное исключение при манипуляции с буфером, возможно, и не было необходимым; однако, если применены связанные списки буферов или программа обобщается на m производителей и n потребителей ($m, n \geq 1$), взаимное исключение необходимо.

Пример 2: Описание входного спулера

Вместо распределения устройства чтения с перфокарт и печатающего устройства каждому интерактивному заданию, выполняющемуся в мультипрограммной системе, обычно на основе вспомогательной памяти обеспечиваются «виртуальные» читающие и печатающие устройства. Задания собираются во вспомо-

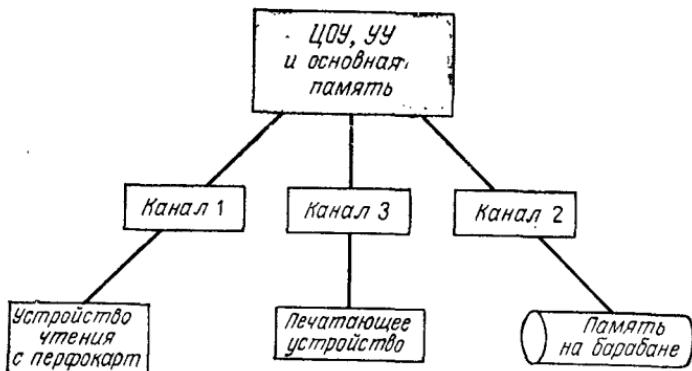


Рис. 3.6. Конфигурация простейшей ЭВМ.

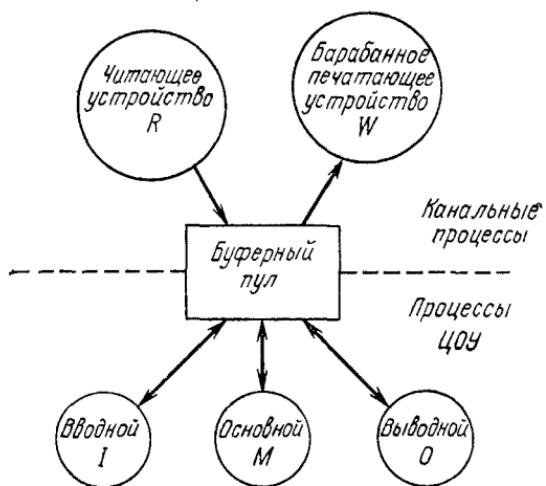


Рис. 3.7. Спулер ввода.

гательной памяти до их выполнения, а их выходные данные записываются во вспомогательную память во время обработки. Мультипрограммирование в таком случае может выполняться гораздо более эффективно. Планировщик заданий становится более свободен в выборе заданий; (виртуальные) операции ввода-вывода выполняются быстрее и, таким образом, уменьшается время, в течение которого задание занимает основную память; нагрузка на устройства ввода и печати может быть распределена по заданиям равномерно. Та часть системы, которая (a) считывает задания во вспомогательную память и (b) печатает выходные данные задания из вспомогательной памяти, назы-

вается *спулером* (*spooler*)¹⁾. Рассмотрим схему части для организации ввода гипотетического спулера [определенная выше задача (а)] для простой конфигурации ЭВМ на рис. 3.6. (Каналы 1 и 3 обычно являются подканалами мультиплексного канала, но могут рассматриваться для приведенного примера как независимые.)

Спuler ввода может быть организован в соответствии с рис. 3.7, где предполагается, что пять процессов работают па-

Таблица 3.1. Переменные, используемые в программах спулера

Переменные	Значение
<i>n</i>	Число буферов $n \geq 2$
<i>Bu</i> [<i>i</i>]	<i>i</i> -й буфер в пуле
<i>curin</i>	Текущий входной буфер есть <i>Bu</i> [<i>curin</i>]
<i>curin</i>	Текущий выходной буфер есть <i>Bu</i> [<i>curout</i>]
<i>inio</i>	Буфер, используемый <i>R</i> , есть <i>Bu</i> [<i>inio</i>]
<i>outio</i>	Буфер, используемый <i>W</i> , есть <i>Bu</i> [<i>outio</i>]
<i>em</i>	Обозначение класса пустых буферов
<i>in</i>	Обозначение класса <i>input-full</i> буферов
<i>out</i>	Обозначение класса <i>output-full</i> буферов
<i>F(t)</i>	Указатель на первый буфер класса <i>t</i> , $t \in \{em, in, out\}$
<i>L(t)</i>	Указатель на последний буфер класса <i>t</i> , $t \in \{em, in, out\}$
Семафоры счета:	
<i>cem</i>	пустых буферов
<i>cin</i>	<i>input-full</i> буферов
<i>cout</i>	<i>output-full</i> буферов
Семафоры взаимного исключения:	
<i>mem</i>	пустых буферов
<i>min</i>	<i>input-full</i> буферов
<i>mout</i>	<i>output-full</i> буферов
Семафоры <i>startio</i> :	
<i>sR</i>	устройства чтения с перфокарт
<i>sW</i>	барабана
Семафоры прерывания:	
<i>xR</i>	устройства чтения с перфокарт
<i>xW</i>	барабана

¹⁾ Сокращение „SPOOL” означает Simultaneous Peripheral Operations On Line (одновременные операции, выполняемые в реальном масштабе времени на периферийном оборудовании); впервые оно было использовано в вычислительной системе IBM 7070.

ралльно. Буферный пул из n буферов ($n \geq 2$), каждый длиной 80 байтов (образ одной карты) будет содержать *пустые* — *empty*(*em*), *заполненные при вводе* — *input-full*(*in*) и *заполненные для вывода* — *output-full*(*out*) буфера. Структура данных буферного пула, стандартные программы манипуляции с буфером и некоторые из имен переменных взяты из разд. 2.4.4. Для удобства ссылок программные переменные и их значения сведены в табл. 3.1; они будут объяснены в дальнейшем, когда мы разработаем программу для пяти процессов.

Взаимосвязь типа «производитель — потребитель» между тремя процессами в ЦОУ, а именно: *I*, *M* и *O*, представлена в табл. 3.2 по отношению к ресурсам буферов. Основной процесс *M* в ЦОУ будет выбирать заполненные при вводе буфера,

Таблица 3.2. Производители — потребители буферных ресурсов

Ресурс	Процессы	
	Производитель	Потребитель
<i>em</i>	<i>M, O</i>	<i>M, I</i>
<i>in</i>	<i>I</i>	<i>M</i>
<i>out</i>	<i>M</i>	<i>O</i>

производить их обработку (например, преобразовывать формат входных данных), создавать таблицы заданий и добавлять задания в очереди, создавать буфера, заполненные для вывода на барабан. Для каждой входной записи может быть создано 0 или более выходных записей. Пусть процесс *M* имеет следующую форму:

```

M: begin LM: GetNextInputFullBuffer; Compute;
    (GetEmptyBuffer; Compute;
    ReleaseOutputFullBuffer;) *
    ReleaseEmptyBuffer; go to LM
end;

```

где (*s*)* обозначает 0 или более появлений строки *s* за время выполнения. Пусть *sem*, *cin* и *cout* — семафоры, которые подсчитывают доступные буферные ресурсы типа *em*, *in* и *out*. Чтобы гарантировать целостность структуры данных в буфере, мы используем три двоичных семафора, *tsem*, *tin* и *tout*, которые обеспечивают взаимное исключение манипуляций с буфером.

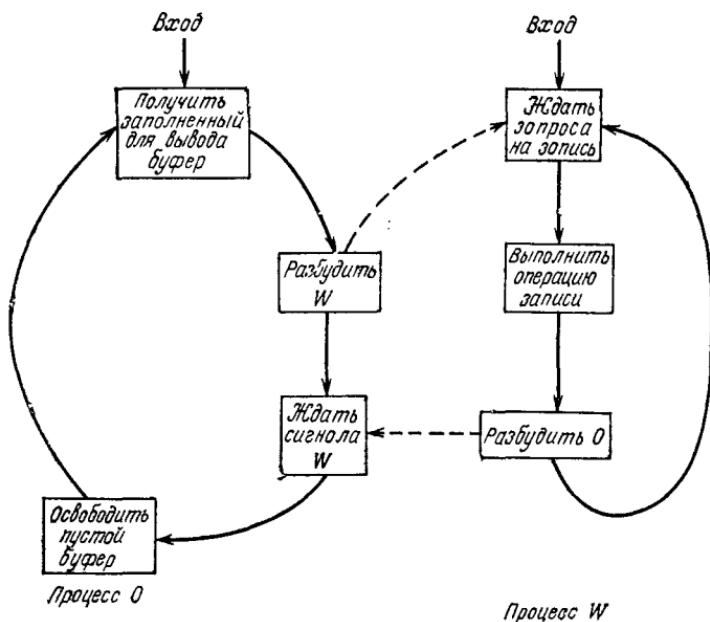
Тогда буферные программы в M есть:

GetNextInputFullBuffer: $P(cin); P(min); curin := TakeBuf(in);$
 $V(min);$
ReleaseEmptyBuffer: $P(mem); AddBuf(em, curin); V(mem);$
 $V(cem);$
GetEmptyBuffer: $P(cem); P(mem); curout := TakeBuf(em);$
 $V(mem);$
ReleaseOutputFullBuffer: $P(mout); AddBuf(out, curout); V(mout);$
 $V(cout);$

Назначением процесса I является обеспечение читающего устройства (R) пустыми буферами, инициация R и добавление заполненных при вводе буферов в пул, когда завершается работа R . С каждым канальным процессом i свяжем два двоичных семафора si и xi , которые будут моделировать аппаратные сигналы *startio* и сигналы *прерывания*. Процесс в ЦОУ будет выдавать $V(si)$, чтобы инициировать канальный процесс i , когда канальный процесс будет прерывать ЦОУ при завершении ввода-вывода по $V(xi)$. Обратно, канальный процесс будет ждать сигнала «огонь» по операции $P(si)$, а обработчик прерываний ввода-вывода будет ждать активизации по операции $P(xi)$. Тогда процессы I и R имеют следующий вид:

```
I: begin LI: P(cem); P(mem); apd; inio := TakeBuf(em);
   V(mem); V(sR); P(xR);
   P(min); AddBuf(in, inio); V(min);
   V(cin); go to LI
end;
R: begin LR: P(sR); RealRead(ch1, Buf[inio]);
   V(xR); go to LR
end;
```

Операция *RealRead* показывает, что R будет выполнять операцию чтения полностью перед переходом к выполнению следующей команды. Предложение *apd* в I означает «избежать возможного тупика» и относится к ситуации, описанной ранее в разд. 2.4.4; возможно, что запрос на пустой буфер из процесса M может никогда не быть удовлетворен, потому что процесс R их все израсходовал и не существует буферов, заполнен-

Рис. 3.8. Синхронизация O и W .

ных для вывода, которые могут быть очищены. Эта проблема может быть устранена включением для *apd* команд:

comment вернуться к началу I , если это последний пустой буфер;
if $F[em] = L[em]$ **then**
begin $V(cen)$; $V(mem)$; **go to** LI **end**;

Это гарантирует, что по крайней мере один пустой буфер всегда доступен в *GetEmptyBuffer* процесса M . (Более полная проверка потребовала бы пустой буфер, только если не было буферов, заполненных при выводе, или не было работы вывода.)

Подобные стандартные программы могут быть легко построены для процессов O и W . В них мы игнорируем важную проблему распределения памяти барабана и полагаем, что начальная ячейка d на барабане каким-то образом была предоставлена процессу W при каждой операции записи на барабан. Команда *RealWrite(ch2, (Buf[outio], d))* может быть использована процессом W . Рис. 3.8 изображает взаимодействия процессов O и W .

Тогда система процессов описывается следующим образом:

```

begin semaphore mem, min, mout, cem, cin, cout, sR, xR, sW, xW;
  mem := min := mout := 1; cem := n;
  cin := cout := sR := xR := sW := xW := 0;
  M and I and O and R and W
end;

```

Каждый из пяти процессов обычно непрерывно активен в системе (или развивается или ждет сигнала «разбудить»), так что естественно сделать их циклическими процессами.

В описании гипотетического вводного спулера опущены многие детали. Однако оно иллюстрирует, как может быть логически описан набор кооперирующихся последовательных процессов с использованием семафоров для взаимного исключения внутри стандартных программ распределения ресурсов, для учета ресурсов и для синхронизации процессов. «Разрешение войти в критическую секцию» и синхронизирующие сигналы могут также рассматриваться как ресурсы в среде производитель — потребитель. В этом случае все семафоры могут рассматриваться как «мягкие» ресурсы.

Пример 3: Взаимообмен сообщениями между внутренними процессами и терминалом

Рассмотрим систему, где ряд внутренних процессов P_1, P_2, \dots, P_n связан с оператором-человеком или интерактивным пользователем U за телетайпной консолью. Мы хотим разрешить как каждому процессу P_i , так и пользователю за терминалом порождать односторонние и двусторонние сообщения. Приведем четыре возможных типа обмена и примеры их применения:

1. *Процесс — Пользователь (PU)*. Процесс P_i посылает одностороннее сообщение (никакого ответа не требуется) процессу пользователя U . Типичное сообщение может содержать некоторую статистику о системе или процессе, например о числе заданий, выполненных в течение последнего часа, или предупреждение о том, что только что произошло переполнение при делении.

2. *Процесс — Пользователь — Процесс (PUP)*. Процесс P_i инициирует двустороннее сообщение к пользователю консоли; последний должен ответить порождающему процессу P_i . Мы разрешаем пользователю задерживать его ответ, он уведомляет систему о задержке немедленной отсылкой фиктивного ответа, скажем „WILCO“. Спустя некоторое время будет послан соответствующий ответ. Немедленный отклик мог бы быть сделан на сообщение, запрашивающее текущую дату; типичной ситуа-

цией, вызывающей задержанный ответ, является запрос оператору от процесса P_i на установку ленты. В любом случае требуется какой-то немедленный ответ, фиктивный или действительный.

3. Пользователь — Процесс (*UP*). Это одностороннее сообщение от U некоторому P_i , например, чтобы объявить доступность некоторого периферийного устройства, которое предварительно было переведено в автономный режим (*off-line*) для ремонта, или чтобы включить переключатель отладки в программе.

4. Пользователь — Процесс — Пользователь (*UPU*). Пользователь запрашивает процесс и ждет ответа. Примерными вопросами могут быть: «Сколько вспомогательной памяти используется для задания спулера?» или «Напечатайте имена программ из моей личной библиотеки программ».

Передача от ЭВМ на консоль будет происходить по полудуплексной линии — данные могут посыпаться в любом направлении, но одновременно возможна только одна передача. Линия могла бы быть стандартным каналом или линией связи с удаленным терминалом; мы не будем касаться конкретного типа. Предполагается, что U имеет наивысший приоритет относительно P_i при инициировании сообщений и что пользователь указывает на это требование нажатием кнопки «внимание», которая устанавливает внутренний флаг a . Флаг a может быть установлен в любой момент времени, даже во время передачи выводных данных через линию.

Внутренние процессы для простоты будут разделены на два несвязанных набора

$\mathcal{P}_1 = \{P_i, \text{ который включает только передачи } PU \text{ и } PUP\}$

$\mathcal{P}_2 = \{P_i, \text{ который управляет только сообщениями } UP$
или *UPU*}

Процессы в \mathcal{P}_1 инициируют переговоры с пользователем консоли, в то время как процессы в \mathcal{P}_2 управляют переговорами, инициированными пользователем. Прямая связь между терминалом и внутренними процессами затруднена из-за необходимости направлять терминалные сообщения в соответствующий P_i . Поэтому мы используем процесс интерпретатора сообщений *MI* для этой задачи, а также для контроля и инициирования физических операций ввода-вывода. Система изображена на рис. 3.9. Этот пример показывает набор соответствующих структур данных и программу, требуемую для синхронизации и выполнения функций отсылки и приема сообщений.

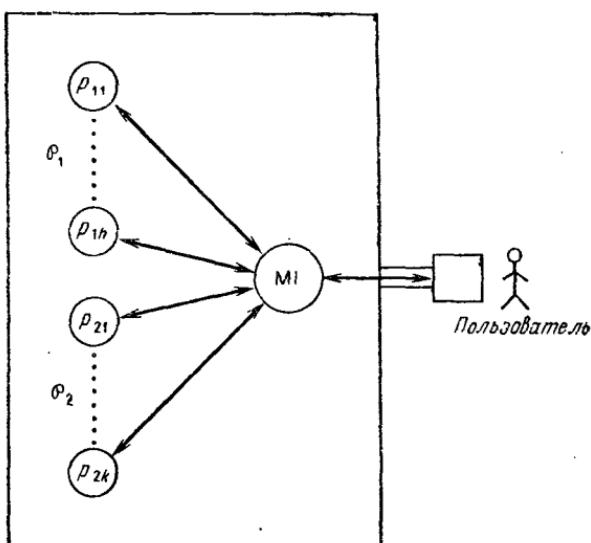


Рис. 3.9. Связь внутренних процессов с терминалом.

Пусть каждое сообщение состоит из триады (p, t, m) , где p — имя внутреннего процесса, $t \in \{up, upi, ru, rup\}$ обозначает тип сообщения и m содержит текст сообщения; первые две записи позволяют оператору и MI идентифицировать сообщение. Для хранения каждого сообщения используется буферный пул; каждый буфер b описывается таким образом:

$$Buf[b] = (pr[b], ty[b], texl[b]),$$

где три компонента соответствуют упомянутым выше элементам. Относительно пула поддерживаются две очереди:

1. Список пустых буферов.
2. Список заполненных буферов типа ru или rup .

Поиск и освобождение буфера происходит через знакомые нам программы $TakeBuf(x)$ и $AddBuf(index, x)$, где $x \in \{empty, full\}$ (см. разд. 2.4.4). С каждым $P_i \in \mathcal{P}_2$ связана переменная $B[i]$, которая используется для хранения индекса буфера во время обработки up или upi . Наконец, применяются несколько семафоров для взаимного исключения, синхронизации и учета ресурсов. Таблица 3.3 содержит значения каждой переменной, употребляющейся в программах.

Таблица 3.3. Переменные, используемые в программах обмена сообщениями

Переменная	Значение
<i>a</i>	Флаг внимания
	Обозначение класса сообщения:
<i>ip</i>	пользователь — процесс
<i>iri</i>	пользователь — процесс — пользователь
<i>ri</i>	процесс — пользователь
<i>rip</i>	процесс — пользователь — процесс
<i>Bu_i[b] =</i>	<i>b</i> -й буфер в пule: (<i>pr[b]</i> , <i>ty[b]</i> , <i>text[b]</i>)
	имя процесса тип сообщения текст сообщения
	Обозначение класса:
<i>empty</i>	пустых буферов
<i>full</i>	заполненных буферов
	Семафоры учета буферов:
<i>ce</i>	пустые буферы
<i>cf</i>	полные буферы
	Семафоры взаимного исключения:
<i>me</i>	пустые буферы
<i>mf</i>	полные буферы
<i>s[i]</i>	Семафоры синхронизации для процесса <i>i</i>
<i>ans</i>	Семафор синхронизации для сообщений типа <i>ip/iri</i>

Процесс $P_i \in \mathcal{P}_1$ имеет такую схему программы:

P_i : begin *L_i*:

⋮

comment Послать сообщение *PU*;

PU: *b* := Dequeue(*ce, me, empty*);

Buf[b] := (i, pu, m);

Queue(cf, mf, b, full);

⋮

⋮

comment Начать передачу *PUP*;

PUP: *b* := Dequeue(*ce, me, empty*);

Buf[b] := (i, pup, m);

Queue(cf, mf, b, full);

comment Ждать ответа;

```

    P(s[i]);
    m := text[b];
    Queue(ce, me, b, empty);
    .
    .
    go to Li
end;

```

Вспомогательные процедуры процесса P_i определяются следующим образом:

```

integer procedure Dequeue(s1, m1, t);
semaphore s1, m1;
begin P(s1); P(m1); Dequeue := TakeBuf(t); V(m1) end
procedure Queue(s1, m1, index, t);
semaphore s1, m1;
begin P(m1); AddBuf(t, index); V(m1); V(s1) end

```

Процесс $P_j \in \mathcal{P}_2$ имеет вид:

```

Pj: begin Lj:
        UP/UPU: P(s[j]); comment Ждать сообщения;
        b := B[j]; m := text[b]; t := ty[b];
        if t = upu then
            begin
                Find the appropriate reply;
                text[b] := reply
            end;
            comment Разбудить MI;
            V(ans);
        go to Lj
end;

```

Интерпретатор сообщений более сложен:

```

MI: begin LMI:
        if a then
            begin
                comment U требует внимания. Прочитать его
                сообщение, когда оно будет
                напечатано;
                Receive(p, t, m);
                a := false; comment Сбросить флаг внимания

```

```

if  $t = up \vee t = upi$  then
  begin comment Ввести сообщение в буфер;
     $b := B[p] := Dequeue(ce, me, empty);$ 
     $Buf[b] := (p, t, m);$ 
    comment Разбудить  $p$ ;  $V(s[p]);$ 
    comment Ждать, пока не будет принято
      сообщение или возвращен ответ;
       $P(ans);$ 
    if  $t = upi$  then  $Send(Buf[b]);$ 
    comment Ответ будет напечатан, если  $upi$ ;
    comment Теперь вернуть буфер в пул;
     $Queue(ce, me, b, empty)$ 
  end
  else if  $t = pup$  then
    begin comment Это задержанный ответ  $pup$ ;
       $text[B[p]] := m; V(s[p])$ 
    end
  end
  else
    begin comment Процесс  $ri$  или  $rui$ , если
       $full$ -очередь имеет записи;
    if  $F[full] \neq full$  then
      begin
         $b := Dequeue(cf, mf, full);$ 
         $Send(Buf[b]);$ 
        comment Проверить для  $pup$ ;
        if  $ty[b] = pup$  then
          begin  $Receive(p, t, m);$ 
            comment Это фиктивный ответ;
            if  $m = "WILCO"$  then  $B[p] := b$ 
            else
              begin  $text[b] := m; V(s[p])$  end
            end
          end
        end;
        go to  $LMI$ 
      end;

```

Семафоры $s[i]$, $i = 1, \dots, n$, используются для получения гарантии, что P_i синхронизирован с MI и U для типов обмена сообщениями: PUP , UP , UPU . Семафор ans обеспечивает синхронизацию MI с $P_i \in \mathcal{P}_2$. Остающиеся семафоры служат для тех же целей, что и в примере 2. Альтернативным подходом для процессов \mathcal{P}_2 могло бы быть построение очереди сообщений UP с требованием того, чтобы MI ждал только ответов UPU . Заметьте, что в MI не существует никаких проверок на ошибки; эти проверки должны, конечно, быть включены в любую реальную систему. Пример является еще одной иллюстрацией применения (и необходимости) синхронизующих примитивов в среде, где несколько процессов могут сосуществовать, связываться и разделять ресурсы.

Двоичные и общие семафоры

Поведение общего семафора всегда может быть смоделировано использованием только двоичных семафоров. Каждый общий семафор S может быть заменен целой переменной N_S и двумя двоичными семафорами, $mutex_S$ и $delays$. Для каждой операции $P(S)$ мы используем

```
 $P(mutex_S); N_S := N_S - 1; \text{ if } N_S \leq -1 \text{ then}$ 
    begin  $V(mutex_S); P(delays) \text{ end}$ 
    else  $V(mutex_S);$ 
```

Каждая операция $V(S)$ заменяется на

 $(2) \quad P(mutex_S); N_S := N_S + 1; \text{ if } N_S \leq 0 \text{ then } V(delays);$
 $V(mutex_S);$

Первоначально $mutex_S = 1$, $delays = 0$ и N_S присвоено начальное значение S . Процесс, который был бы блокирован при выполнении $P(S)$, также блокируется при выполнении (1), так как $N_S \leq -1$ и была бы вызвана операция $P(delays)$; аналогично, если операция $V(S)$ могла вызвать продолжение процесса, то в (2) это также делается носредством $V(delays)$.

Упражнения

1. Каков эффект взаимоизменения
 - a) $P(b)$ и $P(e)$ или
 - b) $V(b)$ и $V(f)$ в процессе производителя в примере 1?
2. Напишите программы для процессов O и W во входном спулере (пример 2).
3. Предположим, что наилучший размер записи барабана 32 байта (т. е. 1 барабанская запись = 4 буферные записи). Пусть операция записи на барабан обозначается

$RealWrite(ch2, (b_1, b_2, b_3, b_4), d));$

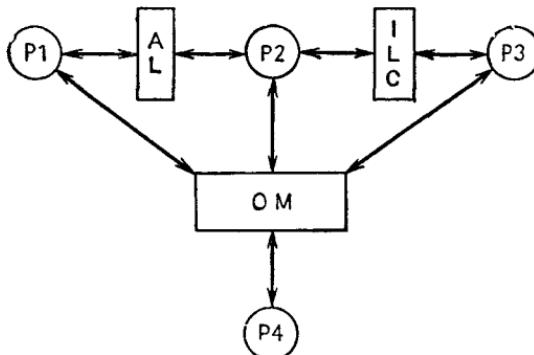


Рис. 3.10. Параллельные процессы в компиляторе.

Данные:

AL: список атомов

ILC: программа на промежуточном языке

OM: выводимые сообщения

Процессы:

P1: сканирование

P2: синтаксис и семантика

P3: генерация программы

P4: вывод

где b_i — буферные указатели и d — адрес записи на барабане. Измените процессы M , O и W и любые семафоры, которые вы пожелаете в примере 2, чтобы отразить длину этой новой выходной записи. Положите число буферов равным $n \geq 5$.

4. Рассмотрите четыре последовательных процесса, $prod1$, $con1$, $con2$ и $prod2$, которые обращаются к общему буферному пулу, состоящему из n буферов ($n \geq 1$). Процесс $prod1$ — это вычислительный процесс, который наполняет буфера; $con1$ — выводной процесс, который выводит (опустошает) содержание буферов, заполняемых $prod1$; $prod2$ — это вводной процесс, который считывает данные в буфера; $con2$ — вычислительный процесс, который производит вычисления с данными, вырабатываемыми $prod2$. Каждый процесс организован следующим образом:

- 1) получить соответствующий буфер, скажем i , из общего пула;
- 2) вычислить (*Compute(Buf[i])*) или выполнить операцию ввода-вывода (*Read(Buf[i])*) или (*Write(Buf[i])*) с выбранным буфером;
- 3) освободить *Buf[i]* и ввести его в пул,
- 4) перейти к шагу 1.

Допустим, что операции *Real* и *Write* являются непосредственно командами ввода-вывода. Напишите программы на АЛГОЛе с операциями над семафорами для описания этих четырех процессов. Система процессов организована следующим образом:

```

begin semaphore ...
  prod1 and con1 and prod2 and con2
end;
  
```

5. Рассмотрите возможные параллельные работы в компиляторе, показанном на рис. 3.10. Каждый из четырех процессов $P1$, $P2$, $P3$ и $P4$ является циклическим и взаимодействует с одним или более другими процессами через общие области данных AL , ILC и OM . Процесс $P1$ будет добавлять указатели атомов к AL , которые используются $P2$. Процесс $P2$ в свою очередь

производит программу на промежуточном языке для *ILC*, которая используется *P3*.

Процессы *P1*, *P2* и *P3* вырабатывают выходные сообщения в *OM*, которые используются *P4*. Пусть общие области данных есть массивы

$AL[0 : al-1]$, $ILC[0 : ilc-1]$, $OM[0 : om-1]$ ($al, ilc, om \geq 1$)

которые организованы как циклические буферы. Предположим, что каждая область данных запроса или освобождения включает в себя только один элемент массива. Напишите соответствующую программу для каждого процесса для добавления и удаления элементов из общей памяти и для гарантирования их правильной синхронизации. Общая форма следующая:

begin semaphore ...

P1 and P2 and P3 and P4

end;

6. Вставьте соответствующую программу в процесс *M1* в примере 3 для тестирования и соответствующей реакции на ошибки оператора.

7. Что нужно сделать, чтобы устранить возможный тупик при использовании ресурса буферного пула в примере 3?

8. Спроектируйте интерпретатор сообщений и внутренние процессы так, чтобы каждый процесс *P_i* в примере 3 управлял обменами всех четырех типов (*PUP*, *PUP*, *UP*, *UPU*).

9. Переделайте пример 1, используя только двоичные семафоры.

10. Останутся ли правильными присутствующие в конце этого раздела модели двоичного семафора, если поставить *V(mutex)* перед проверкой *N_s* в случаях (1) или (2)?

11. Используйте семафоры для описания синхронизации восьми процессов в общем случае предшествования на рис. 3.1(d).

3.5. Реализация семафорных операций

Ни одна из имеющихся в настоящее время ЭВМ не имеет прямых аппаратных команд, соответствующих *P*- и *V*-семафорным операциям¹⁾. Однако программные версии *P* и *V* используются как синхронизирующие примитивы по крайней мере в двух операционных системах (Дейкстра, 1968; Бетурн и др., 1969). Нетрудно запрограммировать логические эквиваленты этих операций на ЭВМ, которые могут как проверять, так и устанавливать ячейку памяти за *одну* (*неделимую*) операцию. Пусть наша версия такой команды обозначена *TS(X)*, где ячейка памяти *X* может содержать или «0» (*false*) или «1» (*true*); мы будем рассматривать *X* как булеву переменную. Тогда *TS(X)* выполняет следующие действия:

проверка (*test*): *R := X*;

установка (*set*): *X := true*;

где *R* есть программно-адресуемый регистр машины, выполняющей *TS*. (Допустим, что *R* может содержать любое из значений *true* или *false*.)

¹⁾ В момент перевода данной книги такие ЭВМ уже имеются. Например, мультипроцессорная система «Эльбрус». — Прим. ред.

Чтобы упростить алгоритмы в следующих разделах, мы будем игнорировать регистр R и трактовать операцию $TS(X)$ как неделимую булеву процедуру, которая возвращает значение, предварительно присвоенное R :

```
Boolean procedure  $TS(X);$ 
Boolean  $X;$ 
begin
     $TS := X;$ 
     $X := \text{true}$ 
end  $TS$ 
```

3.5.1. Реализация с «занятым» ожиданием

Критическая секция CS_i может быть защищена в процессе P_i при использовании программы:

```
 $Li:$  if  $TS(mutex)$  then go to  $Li;$ 
       $CS_i; mutex := \text{false};$ 
```

Предполагается, что одна команда, например *store immediate*, достаточна для установки $mutex$ в **false** в конце CS . Значение $mutex$ есть **true**, когда процесс находится в своей CS , и **false** в противном случае. Если P_i попытается войти в CS_i , когда $mutex$ равно **true**, он будет осуществлять занятое ожидание, зациклившись на Li и затрачивая циклы памяти и время процессора. Действие двоичного семафора S может тогда быть достигнуто:

PS эквивалентно	$L:$ if $TS(S)$ then go to $L;$
$V(S)$ эквивалентно	$S := \text{false};$

Логическая эквивалентность общего и двоичного семафоров, представленных в конце последнего раздела, может быть применена для моделирования общего семафора S ; мы используем простую переменную N_s для подсчета, двоичный семафор ms — для взаимного исключения и двоичный семафор ds — для задержки:

1) Операция $P(S)$ может быть смоделирована программой

```
 $L1:$  if  $TS(ms)$  then go to  $L1;$   $N_s := N_s - 1;$ 
```

```
if  $N_s \leq -1$  then
```

```
begin  $ms := \text{false};$ 
```

```
 $L2:$  if  $TS(ds)$  then go to  $L2$ 
```

```
end else  $ms := \text{false};$ 
```

2) Операция $V(S)$ вводится аналогично.

Занятые ожидания для CS могут и не быть бесполезными, если они являются «короткими» ожиданиями. Часто CS могут быть выполнены относительно быстро. (Они должны быть запланированы, чтобы обладать таким свойством, если оно вообще возможно.) Тогда может появиться возможность запрещать прерывания во время выполнения CS так, что кому-нибудь в течение малого интервала времени будет дано разрешение, гарантирующее вход в CS. В среде, включающей более общие CS и способы синхронизации, где, например, процесс может быть заблокированным, ожидая завершения операции ввода-вывода или ожидая, когда станет доступным аппаратный ресурс, или ожидая сообщение от другого процесса, которое может поступить в любой произвольный момент времени, длительные занятые ожидания могут быть неудовлетворительными. При этом мультипрограммная система может «выродиться» в однопрограммную, а время ответа на события реального времени увеличится до недопустимых уровней.

Упражнения

1. Составьте программу для реализации $V(S)$ для общего семафора S .
2. Переделайте пример 1 последнего раздела (разд. 3.4.2), используя TS и занятое ожидание.
3. Рассмотрите команду $TSB(X, L)$, которая выполняет тестирование, установку и ветвление как одну операцию:

If X **then go to** L **else** $X := \text{true}$;

Запрограммируйте P - и V -операции над двоичными семафорами, используя TSB . (Некоторые машины имеют примитивные TSB , где счетчик команд увеличивается на 1 или на 2, в зависимости от результатов тестирования.)

3.5.2. Устранение занятого ожидания

Занятое ожидание устраняется обеспечением возможной приостановки (блокировки) процессов при P -операциях и возможной активизации процессов при V -операциях. Процесс P , который не может продолжаться при выполнении $P(S)$, будет заблокирован при сохранении p -го вектора состояния, процессор освобождается от выполнения p , и p вносится в список блокирования L_s , связанный с семафором S . Если L_s не пуст, то $V(S)$ будет активизировать некоторый процесс из L_s , скажем p , удаляя p из L_s и помещая p в общий список готовности RL . P - и V -операции попытаются также условно выделить свободный процессор процессу из RL . Процесс, выполняющий P - и V -операции, защищается от перехвата процессора во время этих операций. Чтобы препятствовать процессорам (кроме выполняющего операцию) иметь одновременный доступ к семафору, спискам блокирования и списку готовности, используется TS ; прерывания во время операций запрещены, чтобы обеспечить за-

врение выполнения P и V прежде, чем будет освобожден процессор. При таких достаточно приемлемых предположениях длительные занятые ожидания не будут иметь места. Тогда P - и V -операции определяются следующим образом:

1. $P(S)$:

```

Inhibit interrupts;
L: if  $TS(mutex)$  then go to L;
     $S := S - 1$ ;
    if  $S \leq -1$  then
        begin Block process invoking p;
             $p := RemovefromRL$ ;
             $mutex := false$ ;
            Transfer control to p with interrupts enabled
        end
    else
        begin  $mutex := false$ ; Enable interrupts end;
```

Предполагается, что $RemovefromRL$ всегда возвращает имя процесса из RL^1 ; в этом месте нас не интересует частный процесс, выбранный из списка RL . Заметьте, что $P(S)$ должна рассматриваться как неделимая машинная операция. Таким образом, если процесс q заблокирован при вызове $P(S)$, то часть счетчика ячеек его сохраненного вектора состояния будет указывать на очередную команду процесса q , следующую за $P(S)$.

2. $V(S)$:

```

Inhibit interrupts;
L: if  $TS(mutex)$  then go to L;
     $S := S + 1$ ;
    if  $S \leq$  then
        begin  $p := RemovefromLS$ ;
            if freeprocessors then
                Start executing p on a freeprocessor
            else Add p to RL
        end;
     $mutex := false$ ;
    Enable interrupts;
```

¹⁾ Удобно поддерживать один «пустой» или «фиктивный» процесс в списке RL для каждого процессора; при этом RL никогда не будет пуст, когда действительный рабочий процесс становится заблокированным. Пустой процесс выбирается с помощью $RemovefromRL$ только тогда, когда список RL не содержит рабочих процессов.

В этой реализации мы разрешаем семафору принимать отрицательные значения. Если семафор S отрицательный, то его абсолютное значение $|S|$ дает число блокированных процессов в списке L_s . Так как прерывания запрещены и существует занятое ожидание на L , важно, чтобы приведенная программа была короткой и эффективной по тем же самым причинам, которые обсуждались в последнем разделе. Однако возможно иметь длительные, даже бесконечные занятые ожидания в мультипроцессорной системе. Предположим, что единственный циклический процесс постоянно выполнял операции P и V на одном процессоре и что каждый из остающихся процессоров циклился на TS . Временные условия могли бы быть такими, что первый процесс всегда входил в CS и оставлял другие процессы на всегда зацикленными. Эту ситуацию можно предотвратить, расширяя механизм входа в CS так, чтобы он предоставлял вход на основе кругового циклического (round robin) алгоритма.

Содержимое и организация структур данных процесса, различные списки и планировщики процессов даны в гл. 7. Мы опустили также несколько проверок на ошибки, которые должны быть включены при реализации системы; например, что произойдет в $V(S)$, если $S \leq -1$ на входе и L_s пуст?

Упражнение

Модифицируйте P - и V -реализации в этом разделе так, чтобы были невозможны бесконечные занятые ожидания.

3.6. Другие синхронизирующие примитивы

В операционных системах были предложены и реализованы другие синхронизирующие примитивы, которые выполняют те же функции, что P и V . Действия всех примитивов, обсуждаемых ниже, предполагаются неделимыми. Удобно провести различие между примитивами, используемыми строго для защиты критических секций, и примитивами, применяемыми для более общей схемы взаимообмена сообщениями.

Деннис и Ван Хорн (1966) предложили очень ясный механизм блокирования CS . Критические секции открываются внутри пары $lock\ w$ — $unlock\ w$, где w есть произвольная однобитовая переменная запирания. Эти примитивы определяются следующим образом:

```
lock  $w$ :  $L$ : if  $w = 1$  then go to  $L$ 
           else  $w := 1$ ;
unlock  $w$ :  $w := 0$ ;
```

Команда запирания такая же, как команда $TSB(w, *)$ (* указывает на обращение к самой себе; упр. 3, разд. 3.5.1). Наши предыдущие рассуждения о проблемах занятого ожидания применяются также и здесь. Макрокоманды ENQ (enqueue) и DEQ (dequeue), имеющиеся в операционной системе IBM/360 (OS/360) (IBM, 1967), используются для защиты CS; они устраняют занятое ожидание и обеспечивают блокирование и возобновление процессов. Макрокоманда ENQ используется для захвата управления ресурсом, в то время как DEQ освобождает ресурс; процесс не может освободить ресурс (DEQ), если он не управляет им (предыдущая ENQ). В простейшей форме ENQ и DEQ имеют единственный параметр, имя ресурса. Их выполнение происходит по алгоритму

```
 $ENQ(r)$ : if  $inuse[r]$  then
    begin Insert  $p$  on  $r$ -queue; Block  $p$  end
    else  $inuse[r] := true$ ;
 $DEQ(r)$ :  $p := Remove from r$ -queue;
    if  $p \neq \Omega$  then Activate  $p$ 
    else  $inuse[r] := false$ ;
```

Ресурс r („разрешение войти в CS“) имеет очередь процессов (r -queue), ожидающих его доступности. Процесс, вызывающий ENQ , был назван выше p . В DEQ , если $p = \Omega$, то r -queue пуста; „Activate p “ приводит в основном к постановке p в список готовности и к вызову планировщика процессов¹⁾.

OS/360 имеет общие средства для синхронизации процессов в форме макрокоманд $WAIT$ и $POST$. В простейшей форме они имеют один параметр, который представляет событие; параметр события, по существу, обозначает область памяти для сигнала синхронизации и простых сообщений. Только один процесс может ожидать (по $WAIT$), когда данное событие будет иметь место (будет отмечено посредством $POST$), но отмечать посредством $POST$ одно и то же событие может любое число процессов. Основные действия этих макросов:

```
 $WAIT(e)$ : if  $\neg posted[e]$  then
    begin wait[e] := true; process[e] :=  $p$ ; Block  $p$  end
    else posted[e] := false;
 $POST(e)$ : if  $\neg posted[e]$  then
```

1) Это сильно упрощенная интерпретация ENQ и DEQ . Макрокоманды также разрешают условные запросы на управление и освобождение ресурсов, запросы на несколько ресурсов одновременно и разделение ресурсов (не-исключающее управление).

```

begin if wait[e] then
    begin wait[e]:= false;
          Activate process[e]
    end
    else posted[e]:= true
end;

```

Здесь снова предполагается, что *P* есть имя процесса, вызывающего *WAIT*¹). В этой версии *WAIT* подобна *ENQ* с максимальной длиной очереди 1.

Процессы в наборе обязаны *кооперироваться* друг с другом при использовании всех примитивов, обсуждавшихся до сих пор в этой главе. Более базисным (низкоуровневым) набором примитивов, который может также применяться в некоторых примерах с некооперированными процессами, является *Block* и *Wakeup* (Зальцер, 1966; Лэмпсон, 1968). Переключатель пробуждения-ожидания *wws*[*i*], связанный с каждым процессом *i*, используется во многом так же, как и *wait*[*e*] в *WAIT* и *POST*, чтобы предотвратить блокирование, когда *Block* предшествует во времени *Wakeup*. Примитивы *Block* и *Wakeup* оперируют с процессами следующим образом:

```

Block(i): if  $\neg wws[i]$  then Block process i
            else wws[i]:= false;
Wakeup(i): if ready(i) then wws[i]:= true
                  else Activate process i;

```

Проверка *ready*(*i*) переустанавливает значение переключателя в *true*, если процесс *i* развивается или готов к развитию, и *false* в противном случае. Команда, такая, как *Block*(*), может быть использована для возможной самоблокировки. Таким образом, *Block* и *Wakeup* явно оперируют процессами, а не посыпают сигналы кому-то, готовому принять их.

Взаимосвязь процессов в основном включает передачу сообщений между процессами. В этом контексте мы можем рассматривать семафор как определение очереди из нулевых сообщений, а операции *P* и *V* как низкоуровневые операции передачи сообщений (Вирт, 1969). *V*-операция добавляет сообщение к очереди, в то время как *P*-операция удаляет его из очереди, если возможно. Значение семафора есть число таких сообщений в очереди, если мы сохраняем наше первоначальное

¹) Мы снова допустили некоторый произвол с определениями IBM. Более общие *WAIT* и *POST* позволяют ожидать несколько событий или некоторого подмножества событий и обеспечивают передачу ограничительных сообщений; кроме того, *WAIT* не сбрасывает флаг *posted*[*e*].

ограничение на неотрицательные целые. Интересный и элегантный набор примитивов для передачи сообщений был разработан для системы мультипрограммирования для ЭВМ RC4000 (Бринч Хансен, 1970); примитивы были использованы для связей между внутренними процессами, а также для передачи записей периферийного ввода-вывода. Основные структуры данных — это общий пул буферов, используемых для передачи сообщений, и очереди сообщений, скажем $mq[i]$, связанные с каждым процессом i . Мы будем пользоваться при описании следующими переменными:

- r : подразумеваемый приемник сообщений
- m : содержимое сообщения
- b : буфер для хранения сообщения
- s : поставщик сообщения; s_b : первоначальный поставщик сообщений в буфер b
- a : ответ или отклик на сообщение
- t : тип ответа, фиктивный или реальный
- p : имя процесса,зывающего примитив

Параметры, выделенные ниже жирным шрифтом, устанавливаются при возврате из операций; в противном случае параметры устанавливаются при входе. Перечислим синхронизирующие примитивы Бринча Хансена:

1. *Sendmessage*(r, m, b);

Получить буфер b из пула. Скопировать m, p, r в b и поместить в очередь $mq[r]$. Активизировать r , если он ожидает сообщения.

2. *Waitmessage*(s, m, b);

Если $mq[p]$ пуста, блокировать процесс p ; в противном случае удалить элемент очереди b . Передать сообщение из b в m и имя поставщика в s .

3. *Sendanswer*(t, a, b);

Скопировать t, a и p в b и ввести b в очередь $mq[s_b]$. Активизировать s_b , если он ожидает ответа.

4. *Waitanswer*(t, a, b);

Блокировать p до тех пор, пока ответ не поступит в b . Поместить ответ в a и тип в t . Возвратить буфер b в пул.

Нормальный протокол сообщений требует использования всех четырех примитивов; поставщик выдает *Sendmessage*, за которым следует в некоторой момент *Waitanswer*, в то время как получатель выдает *Sendmessage* и последовательно *Sendanswer*. Ответ типа „фиктивный“ вставляется системой, когда адресуемый процесс потребителя сообщения не существует.

Идентификаторы поставщика и потребителя сообщения запоминаются в буфере¹⁾, чтобы позволить системе проверять тождественность процессов при *Sendanswer* или *Waitanswer*. В этом случае выявляются некооперированные процессы, которые умышленно или по ошибке пытаются взаимодействовать с помощью диалога.

Все примитивы, рассмотренные в этой главе, содержат в себе возможность логического блокирования и (или) активизации процессов. Однако потенциальное блокирование и активизация процессов происходит и тогда, когда запрашиваются или освобождаются „ресурсы“ любого типа, например, как результат запроса или освобождения блоков основной памяти. Мы подробно рассмотрим этот более общий случай в гл. 7 после того, как будут разработаны некоторые другие аспекты систем мультипрограммирования.

Упражнения

1. Реализуйте операции *P* и *V* над общим семафором в терминах макросов IBM OS/360 *ENQ*, *DEQ*, *WAIT* и *POST*.
2. Используйте *lock*, *unlock*, *Block*, *Wakeup* для описания обменов „производитель — потребитель“ в примере 1, разд. 3.4.3.
3. Повторите упражнение 2, используя примитивы Бринча Хансена для взаимообмена „производитель — потребитель“.
4. Используйте примитивы Бринча Хансена вместо *P* и *V* в примере 3, разд. 3.4.3.

¹⁾ Имеется в виду один и тот же буфер. — Прим. перев.

4. Введение в системы мультипрограммирования

4.1. Доводы в пользу мультипрограммирования

Работа классической системы пакетной обработки, в которой задания проходят через систему в строго последовательном порядке по одному во времени, крайне неэффективна в ЭВМ, имеющих независимые процессоры ввода-вывода или каналы. Рассмотрим сначала использование ресурсов процессора. При тщательном планировании иногда можно достигнуть максимального перекрытия работы центрального процессора и процессора ввода-вывода. (Это трудная задача, которая редко разрешается.) Оптимизация задания в этом смысле тем не менее все еще приводит к незанятым процессорам (центральным или ввода-вывода) для программ с ограниченными вычислениями или вводом-выводом соответственно. Даже в том случае, когда процессоры поддерживаются занятами большую часть времени, использование других ресурсов ЭВМ часто бывает недостаточным: например, любая основная память, не занятая текущим заданием (и некоторой минимальной частью операционной системы), является, по существу, потерянным ресурсом. Если бы одновременно были активны несколько программ, то не только уменьшилось бы число незанятых ресурсов, но также, что более существенно, улучшилось бы время прохождения и смены заданий. Это явилось первым соображением, положенным в основу развития мультипрограммных систем. Однако и при отсутствии независимых аппаратных процессоров ввода-вывода мультипрограммирование может по-прежнему быть выгодным. Например, при последовательном разделении времени процессора нескольким интерактивным пользователям может быть разрешено пропускать небольшие задания со своих терминалов; ЭВМ систематически выделяется каждому терминалу на короткий период времени, так что у пользователя создается иллюзия собственной ЭВМ.

Основная идея — поддерживать в основной памяти в активном состоянии несколько независимых программ или заданий¹⁾. Особенно хорошим вариантом было бы задание A с ограниченными вычислениями (например, обновление большого файла данных) и задание B с ограниченным вводом-выводом (например, решение системы частных дифференциальных уравнений).

¹⁾ Более точно мы должны говорить в терминах процесса (процессов), связанного (связанных) с заданием.

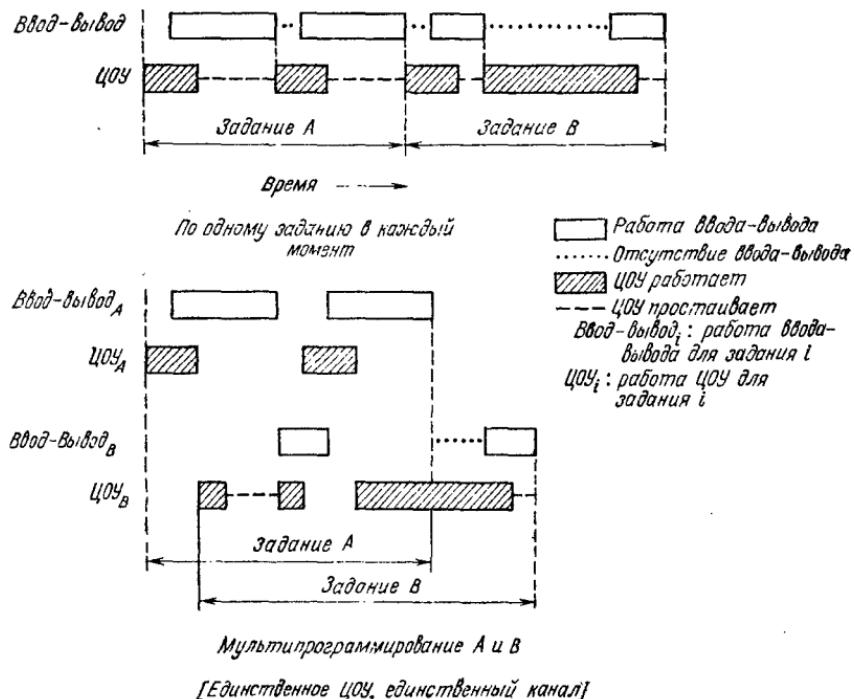


Рис. 4.1. Однопрограммная обработка заданий и мультипрограммирование.

Пока задание А ожидает ввода-вывода, задание В может выполнять вычисления; при завершении ввода-вывода управление вновь передается А. Рис. 4.1 иллюстрирует выполнение двух таких заданий как в однопрограммной (по одному заданию в каждый момент времени), так и в мультипрограммной среде; при мультипрограммном выполнении и А и В предполагаются резидентными в основной памяти и разделяют ЦОУ и канал. В этом простом примере ясно видно сэкономленное мультипрограммированием время. Заметьте, что мультипрограммирование базируется на разделении *времени* процессоров и разделении *пространства* основной памяти, а также на потенциальном разделении других ресурсов.

Существуют и другие преимущества, вытекающие из мультипрограммной организации. Для того чтобы ввести задания в систему и вывести их из системы, должны существовать системные процессы, разделяющие время и пространство с процессами пользователей. Тогда становится возможным управлять несколькими станциями ввода заданий и интерактивными вычислениями со многих индивидуальных консолей. Дополнительную выгоду, заключающуюся в экономии пространства памяти,

можно извлечь из возможности разделять программу. Если несколько активных процессов требуют один и тот же языковый транслятор или запрашивают одну и ту же системную службу, то единственная копия этих стандартных программ может разделяться асинхронно при условии, что они написаны как „чистые“ программы (программы, которые не модифицируют сами себя). Кроме того, система и пользователи получают возможность большего контроля над планированием заданий. Потребности в ресурсах и приоритеты набора ожидающих заданий — например, ожидающих во вспомогательной памяти — могут быть использованы планировщиками для оптимизации использования системных ресурсов и для обеспечения соответствующей реакции для заданий высокого приоритета. Аналогично, аппаратно-программные механизмы для переключения между процессами позволяют высокоприоритетным задачам динамически получать старшинство или даже приостанавливать задачи более низкого приоритета. И что самое важное, достижения в области аппаратно-программного разделения ресурсов посредством мультипрограммирования дают возможность предоставить большое разнообразие услуг.

В настоящее время большинство средних и крупных операционных систем пакетной обработки, интерактивных систем, систем реального времени и общего назначения являются мультипрограммными. Однако преимущества такого режима работы не даются даром. Требуются специальные средства программного обеспечения, а также дополнительная аппаратура, а переход времсни и пространства памяти, вызванный системой при выполнении, часто бывает значительным.

Цель главы 4 — дать общее представление о системах мультипрограммирования. Сначала это представление формируется при рассмотрении требований к аппаратуре и программному обеспечению для мультипрограммирования. Затем описываются два крайних уровня виртуальной машины, которые проявляются в любой ОС, — пользовательский интерфейс, представляемый языком команд и управления, и программный интерфейс первого уровня с машиной, называемой ядром мультипрограммной системы. В последнем разделе описываются некоторые общие подходы к проектированию мультипрограммной системы (МС).

4.2. Компоненты систем

4.2.1. Характеристики аппаратуры

Мы будем иметь дело с семейством вычислительных систем, каждый член которого имеет следующие основные элементы: p центральных обрабатывающих устройств P_i , m модулей ос-

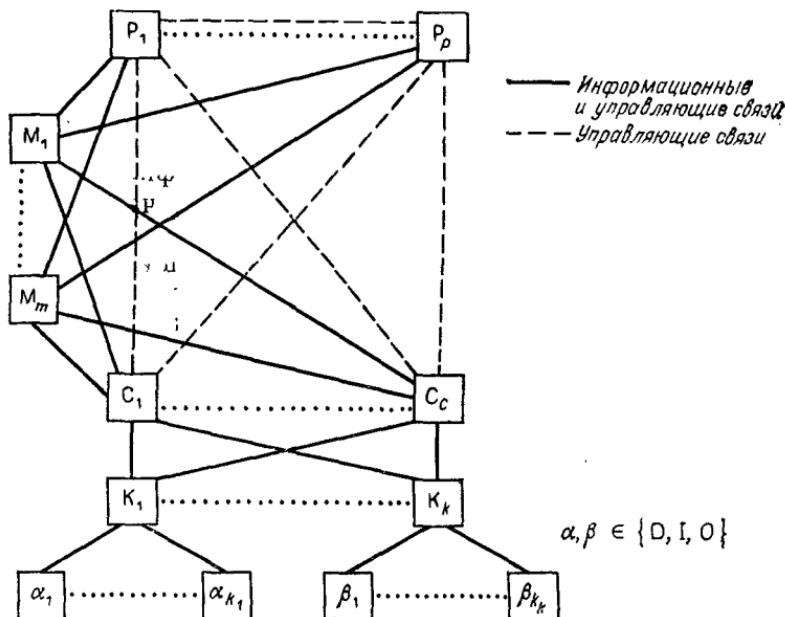


Рис. 4.2. Конфигурация аппаратуры для вычислительной системы общего назначения.

новной (оперативной) памяти M_i , с каналов данных или процессоров ввода-вывода C_i ; k контроллеров устройств ввода-вывода K_i ; d устройств вспомогательной памяти D_i и io периферийных устройств ввода-вывода и терминалов IO_i , где $p, m, c, k, d, io \geq 1$. На рис. 4.2 представлена типичная организация этих компонент; это всего лишь расширенная версия нашей минимальной машины из гл. 1.

Любая современная вычислительная система может рассматриваться как член этого семейства. Устройства P_i могут быть организованы симметрично относительно друг друга или могут работать в режиме „ведущий/ведомый“. Аналогично, модули памяти M_i могут быть различного типа и иметь различные иерархические отношения друг с другом. В качестве возможного варианта мы могли бы включить устройства управления передачей для K_i ; в этом случае линии связи соединяют K_i с удаленными устройствами (α_i или β_i на рисунке), которые тогда могут быть как обычными вычислительными системами, так и устройствами ввода (I) или вывода (O).

При такой организации для мультипрограммной работы аппаратура должна обладать рядом свойств, которые в одних случаях являются необходимыми, а в других упрощают ее.

1. Средства приоритетного прерывания. Переключение управления ЦОУ от одного процесса на другой обычно происходит в результате аппаратных прерываний, вызванных или извне, например каналами ввода-вывода и другими устройствами, или изнутри, например программными ошибками, преднамеренными прерываниями (например, командами вызова „супервизора“), таймерами и отказами аппаратуры¹⁾). Прерывания в общем случае происходят в непредсказуемые моменты времени, и одновременно может произойти несколько различных событий, вызывающих прерывания. Так как для системы необходимо учитывать относительную важность процессов, ожидавших сигналов прерывания, которые должны разбудить эти процессы, а также из-за существования ограничений на длительность интервала времени между событием, вызвавшим прерывание, и обработкой этого прерывания, прерываниям присваиваются аппаратные и программные приоритеты. Например, машинная ошибка обычно должна быть распознана немедленно; сигнал „ввод разрешен“ от устройства сбора данных, работающего в режиме „в линию“ (on-line), может потребоваться обслужить за короткий период времени, иначе данные могут быть потеряны; в то же время обработку сигнала прерывания по окончании операции ввода-вывода в момент завершения записи на магнитную ленту можно отложить на некоторое неопределенное время в будущем в зависимости от приоритета этого конкретного процесса записи на ленту. Встроенный набор аппаратных приоритетов для широких классов прерываний, а также возможность запрещать и разрешать прерывания с помощью программы позволяют системе динамически устанавливать отношения предшествования между событиями, вызывающими прерывания. Когда происходит прерывание, аппаратура должна быстро запомнить состояние текущих процессов, определить причину прерывания и изменить состояние ЦОУ, чтобы инициировать соответствующую стандартную программу обработки прерывания.

2. Защита команд и памяти. Чтобы эффективно предотвращать вмешательство ошибочных процессов или „процессов-злоумышленников“ в дела других процессов, включая системные, а также предотвращать разрушение процессов, важно иметь аппаратные средства для защиты областей основной памяти и для ограничения набора машинных команд, доступных для конкретного процесса. Например, в общем случае нежелательно предоставлять возможность программе пользователя непосредственно запрещать прерывания.

¹⁾ См. сноску в разд. 1.5 относительно систем, которые используют технику голосования, а не прерывания.

3. Динамическая настройка адресов. Аппаратура, которая динамически вычисляет адреса команд и данных во время выполнения программы, дает возможность перемещать программу или части программы в пределах или вне основной памяти без выполнения длительной статической настройки адресов. В системах с развитыми средствами подобного рода программы не требуется размещать в непрерывных областях памяти, их можно легко „перекачивать“ в память или из памяти, они могут быть сокращены и расширены по объему во время прогона и могут лишь частично размещаться в основной памяти. Динамическая настройка адреса не является необходимым, но, несомненно, является удобным средством мультипрограммных систем.

4. Таймер. В ЦОУ должен быть включен программируемый таймер, который может генерировать прерывание после подсчета произвольного числа небольших временных приращений, чтобы управлять разделением времени между несколькими процессами. Это необходимо как для неинтерактивных систем пакетной обработки, так и для интерактивных систем разделения времени.

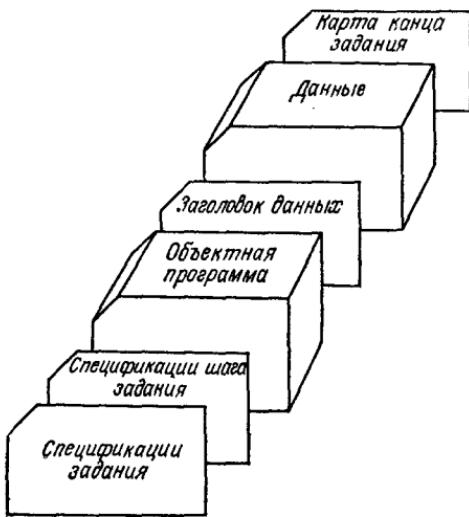
5. Базовые регистры. Разделение программы между несколькими процессами требует, чтобы она адресовалась к данным косвенно через регистры, которые будут содержать базовые адреса областей данных, связанных с процессом, использующим программу. Разделяемая программа и ее постоянные данные могут выполняться в режиме „только чтение“.

6. Вспомогательная память прямого доступа. Большая вспомогательная память с быстрым доступом необходима для запоминания заданий пользователей, библиотечных и системных программ и выходных потоков. Это придает системе гибкость в выборе заданий для загрузки, позволяет поддерживать периферийные устройства первичного ввода и вывода почти постоянно активными и уменьшает задержки при пересылке информации из основной памяти и в основную память.

Позже мы рассмотрим эти свойства аппаратуры более подробно, так как они влияют на проектирование и организацию любой операционной системы.

4.2.2. Базовое программное обеспечение

Требования к программному обеспечению в случае мультипрограммирования могут стать ясными при изучении прохождения задания пользователя через типичную систему. Рассмотрим спачала простое неинтерактивное задание *J*, заключающееся в выполнении объектной программы, полученной

Рис. 4.3. Задание *J*.

в результате предыдущей компиляции. Мы полагаем, что *J* поступает через станцию ввода пакета и имеет вид, показанный на рис. 4.3.

1. Задание *J* считывается в основную память и размещается во вспомогательной памяти (работа спулера ввода). В это время из спецификаций задания и шага задания извлекается информация для планирования *J*. Эта информация включает приоритет задания, классификацию (например, задание с ограниченными вычислениями) и общие требования к ресурсам (например, объем основной памяти, оценку времени ЦОУ, количество лентопротяжных механизмов и т. д.). Задание *J* становится известным системе при запоминании его характеристик в списке резидентных заданий.

2. В конце концов задание *J* выбирается для обработки. Это решение основывается на свойствах и состояниях текущих процессов в системе и на сопоставлении характеристик *J* характеристикам других заданий в списке резидентных заданий. Распределяется основная память и загружается часть задания с объектной программой; этот этап может также включать загрузку и связывание библиотечных программ. После загрузки входная программа больше не рассматривается как данные, а определяет программу для нового процесса ввода, скажем *j*, который создается. Процесс *j* с его начальным состоянием вносится в список готовых процессов. Поскольку используется вспомогательная память, объектная колода теперь может быть удалена.

3. Процесс j в некоторый момент распределяется на ЦОУ и начинает развиваться. Если поступает в систему или „просыпается“ более приоритетный процесс, то j может быть вытеснен со своего ЦОУ и возвращен в список готовности; вытесняющий процесс может быть независимым процессом или зависимым, таким, как процесс, отвечающий за буферизацию ввода для j . Ввод и вывод осуществляются через виртуальные читающие устройства с перфокартами и печатающие устройства во вспомогательной памяти. Всякий раз когда j должен ждать завершения операций ввода-вывода, он входит в состояние блокирования и освобождается ЦОУ. В общем случае j будет претерпевать много изменений состояний, проходя через состояния готовности, продолжения и блокирования; эти изменения являются управляемыми по прерываниям. (Процесс связывания — загрузки на этапе 2 также будет проходить через эти состояния.)

4. Процесс j завершается или естественно или в результате возникновения ошибки. Основная память перераспределяется; вспомогательная память, содержащая входные данные процесса j , освобождается, сообщения о завершении и соответствующая информация для J добавляется к выводному файлу задания J , а J вносится в список завершенных заданий.

5. Стандартная программа вывода (выводной спулер) отпечатывает выходные данные задания на периферийном печатающем устройстве. В системный журнал заносятся статистические данные о работе задания J . Затем J полностью удаляется из системы.

Задание в режиме разделения времени, предусматривающее интерактивную связь с пользователем через телетайп или графический терминал, не проходит через фазу спулера, но в других отношениях его прохождение через систему аналогично. Интерфейс пользователя с системой (задание и спецификации задания) — это язык команд в режиме „в линию“, а не карты управления заданием, как в предыдущем случае. Программы, которые нужно выполнить, загружаются по запросу или создаются в реальном времени пользователем. Процесс или процессы, порожденные этим заданием, часто изменяют состояние, когда они запрашивают и получают ресурсы от системы, и связываются с пользователем через его терминал. Необходимость быстрой реакции системы на команды пользователя предписывает способы планирования, отличающиеся от способов планирования пакетных заданий.

Из только что описанного прохождения задания мы можем вывести следующие характеристики работы систем:

1. Процессы создаются и уничтожаются, а также часто изменяют свое состояние; например, процесс j .

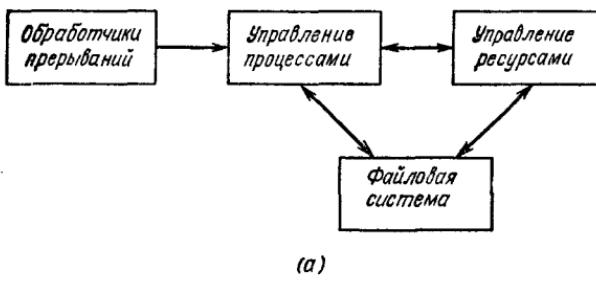
2. Процессы обычно нуждаются в связи друг с другом; например, операции, выполняемые спулером (см. также пример 2 в разд. 3.4.3).

3. Ресурсы динамически закрепляются и освобождаются; например, основная и вспомогательная память.

4. Необходима большая обработка ввода-вывода и файлов; например, создание файлов резидентного задания и манипуляция с ними.

Кросс-секция, получаемая во время работы системы мультипрограммирования, поможет обнаружить число пользовательских и системных процессов в состоянии готовности, продолжения и блокирования. Процессы в состоянии готовности ожидают доступности ЦОУ, в то время как заблокированные обычно блокируются по запросам на ресурсы. Кроме процессов в ЦОУ, будут существовать „внешние“ процессы ввода-вывода, связанные с каналами данных, периферийными устройствами и вспомогательной памятью. На рис. 4.4(а) представлены самые главные компоненты программного обеспечения, необходимые для управления такой системой.

Центральной задачей является задача *управления процессами*: создание, удаление, связь, управление и планирование процессов. Система может быть очень чувствительной к приме-



(a)



(b)

Рис. 4.4. Основное программное обеспечение при мультипрограммировании: (а) — основные компоненты программного обеспечения; (б) — расширенная машина.

няемым стратегиям планирования, т. е. алгоритмам, по которым принимается решение о выборе из многих конкурирующих процессов некоторого процесса, способного развиваться, если ему будет выделен процессор. Арбитром в состязании за все другие ресурсы является система *управления ресурсами*, которая руководит и распределяет такие ключевые ресурсы, как пространство основной и вспомогательной памяти. Поведение системы определяется в значительной степени выбранными стратегиями распределения ресурсов. Третьим компонентом является *файловая система*; эта подсистема ответственна за создание, разрушение, модификацию и восстановление информации на вспомогательной памяти и периферийных устройствах. Файловая система представляет расширенную версию IOCS (системы управления вводом-выводом) ранних систем пакетной обработки. Эти три компонента тесно связаны и взаимодействуют друг с другом; например, с помощью файловой системы запрашиваются процессы для управления ресурсами, чтобы управлять диспозицией вспомогательной памяти или получать буферы. В конечном счете эти компоненты обычно вызываются посредством набора основных стандартных программ обработки прерываний; например, создание, завершение, блокирование и активизация процесса обычно происходят через прерывания. Заметьте, что конкретная система может не иметь такой организации в явном виде и функции каждого блока могут быть распределены между другими системными модулями.

Операционная система может интерпретироваться как расширение аппаратуры ЭВМ, т. е. как виртуальная машина, обеспечивающая дополнительные и более удобные средства для спецификации вычислительного процесса и управления им. Та часть системы, которая постоянно находится в основной памяти, исторически получила название *ядра* системы (рис. 4.4(б)). Ядро будет, как правило, состоять из минимального набора примитивов и процессов для управления процессами, ресурсами и вводом-выводом.

На более высоком уровне, чем уровень указанного выше базового программного обеспечения, находятся процессы и программные модули, отвечающие за управление потоком заданий через систему, за отклик на запросы пользователей на обслуживание и за управление самой операционной системой. Мы включаем в эту категорию подсистему спулера ввода и вывода, интерпретатор языка управления заданиями или командного языка пользователя и связанные с ним процессы, создаваемые для управления каждой фазой задания, а также процессы для связи с оператором ЭВМ и для предоставления ему.

возможности управлять конфигураций ОС и ее работой. Чтобы реализовать эти функции высокого уровня, используются элементы компонент базового программного обеспечения.

4.3. Ядро операционной системы

Ядро ОС — это базовый набор *примитивов* и *процессов*, на основе которых строится остальная часть системы. Ядро традиционно всегда реализовывалось средствами программного обеспечения, но в будущем можно ожидать его появления или непосредственно в аппаратуре или в виде микропрограмм.

Различие между процессом и примитивом заслуживает некоторого пояснения. Примитив вызывается процессом в форме вызова подпрограммы или встроенной программы (макрорасширения), которая является частью вызывающего процесса, зачастую *критической секцией* внутри последнего. С другой стороны, процесс, запрашивающий действие со стороны другого процесса, посылает этому процессу запрос в виде сообщения и часто сам блокируется до тех пор, пока не будет произведено это действие, т. е. о двух процессах говорят как о почти независимых видах деятельности (работах), происходящих *параллельно*. Служебный процесс, скажем член ядра, может быть написан так, чтобы он оставался в состоянии блокировки, пока не получит запроса, или он мог бы быть постоянно *требующим* у системы работы. В первом случае процесс можно также рассматривать как операцию, причем разница заключается в удобстве концепции. Практический подход здесь заключается в том, что вызов процесса более сложный и занимает больше времени и пространства. Часто говорят об операциях ядра как о процессах независимо от метода реализации, потому что процесс, запрашивающий действие системы, обычно не имеет возможности (ресурсов) для выполнения этого действия самостоятельно; если бы он пожелал сделать это, некоторый другой процесс должен был бы уступить запрашивающему требуемые ресурсы. Одним из примеров является команда „вызов супервизора“, которая инициирует процесс в аппаратуре, чтобы изменить состояние машины для предоставления дополнительных ресурсов (например, возможность выполнять команды инициации канала).

Множество стандартных программ ядра может быть разделено на четыре подмножества в соответствии с их функциями: управление процессами, распределение ресурсов, ввод-вывод и обработка прерываний. В этом разделе приводится первоначальное описание некоторых возможностей примитивов, необходимых для каждого подмножества.

Примитивы для управления процессами и для их связи

Процессы непрерывно входят в МС и покидают ее; наглядными примерами являются процессы, определенные в заданиях пользователей. Следовательно, необходимы примитивы для создания процесса и для его уничтожения. Создание включает инициализацию вектора состояния, распределение ресурсов, достаточных для развития процесса, когда ему предоставлен процессор, а также идентификацию процесса для системы, например, с помощью введения его имени в глобальный список. Когда процесс разрушен (завершен), его ресурсы должны быть освобождены и идентификация о процессе удалена из любых системных структур данных. При создании либо неявно может быть указано состояние процесса „готов“, либо процесс может быть неактивным; в последнем случае требуется отдельная операция для установки его в состояние готовности, т. е. для запуска процесса.

Нотации параллельного программирования, описанные в разд. 3.1.2, неявно включали эти две операции. Операция „*fork e*;“ создает новый процесс. Вектор начального состояния обычно содержит по крайней мере элементы состояния: счетчик команд (*instruction counter=e*) и состояние „готов“ (*status== „ready“*); процесс может быть представлен как разделяемый ресурс, причем область памяти содержит программу, начиная с метки *e*. Напротив, операция *quit* разрушает вызывающий процесс. В менее явном виде нотация *and* также подразумевает создание и завершение процесса. Приемлемая интерпретация выражения

S₀; S₁ and S₂; S₃

обеспечивается эквивалентной программой:

S₀; t:=2, fork s₂; S₁; join t, s₃; quit;
s₂: S₂; join t, s₃; quit;
s₃: S₃;

Вторым типом являются примитивы синхронизации и связи. Мы хотим иметь базовые средства для передачи сообщений между процессами, которые включают прямо или косвенно сигналы блокировки и пробуждения. Среди процессов обработки прерываний на различных процессорах процессы обработки прерываний от внешней аппаратуры обеспечивают эту службу на самом нижнем уровне. В предыдущей главе рассматривалось несколько возможных кандидатов для более общих программных примитивов: операции *P* и *V* над семафорами, *Block* и *Wakeup*, *WAIT* и *POST*, программы связи Бринча Хансена. В основе каждой из этих операций должен быть заложен некоторый

механизм для запоминания векторов состояния и перемещения процессов между списками готовности и блокировки.

Ядро часто может быть построено так, что *планировщики процессов*, т. е. распределители процессора, вставлены в вышеупомянутые примитивы процесса и вызываются косвенно, только через эти операции. Планировщик для данного набора процессоров может распределять процессоры на каком-нибудь приоритетном базисе; он мог бы, следовательно, вызываться при любом изменении приоритетов процессов в системе, а также внутри других примитивов. Операция „изменить приоритет процесса“ могла бы тогда быть включена как часть ядра. Если мы предположим, что сообщения являются просто ресурсами другого класса, тогда примитивы связей могут быть заменены операциями над ресурсом, как показано ниже.

Примитивы для управления ресурсами

Основными аппаратными ресурсами, требующими явных стандартных программ управления, являются центральные процессоры, основная память, процессоры ввода-вывода, вспомогательная память и периферийные устройства. С каждым классом ресурсов связаны процедуры *распределения* и *освобождения* одного или более элементов класса. Общий вид этих примитивов может быть таким:

```
x := allocate(resourceclass, requestdetails);
free(resourceclass, identificationdetails);
```

Параметр *requestdetails* обычно включает такие элементы, как число желаемых устройств и признак, является ли запрос „критическим“ или нет, т. е. может ли процесс, вызывающий *allocate*, продолжаться, если ресурс не может быть предоставлен немедленно. Когда ресурс распределен, программа обычно возвращает указатель, идентифицирующий конкретный член класса.

Пример

Запрос на 3000 слов основной памяти мог бы быть сделан вызовом

```
FIRSTADDRESS := allocate(MAINSTORE, WORDS = 3000);
```

Аналогично, четыре блока основной памяти можно было бы освободить с помощью

```
free(MAINSTORE, BLOCKS = (13, 8, 21, 75));
```

Заметьте, что процесс, вызывающий команду *allocate* с критическим запросом, должен быть установлен в заблокированное состояние, если запрос не может быть удовлетворен немедленно. Следовательно, вызовы примитивов управления процессом могут быть вставлены в стандартные программы управления ресурсами. Стандартная программа *free*, аналогично, приводит к возможному пробуждению процессов, ожидающих доступности ресурса. Управление ресурсом для аппаратных устройств становится особенно сложным, когда процесс может приобретать ресурсы *динамически* в противоположность *статическому* приобретению ресурсов в момент своего создания.

Здесь могут быть также рассмотрены программные ресурсы, такие, как разделяемые данные, заполненные буферы, сообщения и даже критические секции. Это становится более очевидным, если мы заменим названия *allocate* и *free* на *request* и *release* соответственно; ресурсы всех типов могут быть „затребованы“ и „освобождены“. Однородный механизм управления ресурсами, основанный на операциях *request* и *release*, был исследован и разработан Вейдерманом (1971); мы коснемся этого подхода дальше, в гл. 7.

Методы управления различными классами ресурсов сильно различаются и являются предметом нескольких последующих глав. В большинстве ОС каждый класс ресурсов имеет отдельный набор примитивов. Однако, чтобы обнаружить, предотвратить и устранить тупиковые ситуации (разд. 2.4.4 и гл. 8), а также чтобы оптимизировать поведение системы, должно быть рассмотрено взаимодействие запросов на ресурс.

Операции ввода-вывода

Ядро ввода-вывода подобно IOCS пакетной обработки по одному заданию, рассмотренной в разд. 2.5, но усложнено требованиями мультипрограммирования. Основными примитивами являются операции *read*, *write* и *control* для связи между основной памятью и различными устройствами ввода-вывода; включено также базовое средство для передачи сообщений между ЭВМ и одной или более консолями оператора.

На самом нижнем уровне ядра операция ввода-вывода могла бы быть такой:

execute IO(channel, channelprogram);

Она обычно вызывает планировщик канала, чтобы иницирировать операцию; используя семафоры, мы можем описать

действия внутри ядра следующим образом:

```
P(channelfree[channel]);
startio(channel, channelprogram);
P(iocomplete[channel]);
V(channelfree[channel]);
```

Первая *P*-операция в последовательности включает потенциальный вызов распределителя канала, в то время как вторая *P*-операция, по существу, ждет прерывания при завершении ввода-вывода. Напротив, команда „*execute IO*“ или различные примитивы низкого уровня могли бы просто вставлять запрос на ввод-вывод в очередь планировщика соответствующего канала и сразу же осуществлять возврат в вызывающую программу, которая может затем продолжаться параллельно с вводом-выводом; вызывающая программа должна явно проверять завершение ввода-вывода в некоторый более поздний момент.

На высшем уровне запрос на ввод-вывод, такой, как *READ(CARDREADER, INPUTAREA)*, обычно вызывает элементы файловой системы (гл. 9) для отыскания справочников файла, для „открытия“ файла (т. е. для организации возможности доступа к нему), для буферизации, для выдачи команд как физического позиционирования файла, так и выполнения операции, а также для планирования результирующей последовательности операций. Все эти функции могли бы быть реализованы в составе ядра ввода-вывода. Примитивы для управления процессами и ресурсами применяются обычно во всех приведенных выше задачах.

Стандартные программы обработки прерываний

Внешние и некоторые внутренние прерывания могут интерпретироваться как сигналы пробуждения, направляемые к блокированным процессам ЦОУ. Оба этих типа прерываний сигнализируют об одном или более из следующих событий (Уотсон, 1970):

1. Процесс завершен. Примерами являются завершение аппаратных процессов канала или завершение внутреннего программного процесса; последний может быть переключен операцией *quit*.

2. Затребовано обслуживание. Оператор ЭВМ мог, например, сгенерировать прерывание „внимание“ с операторской консоли для того, чтобы инициировать выполнение некоторых директив, направленных в ОС; в ЭВМ со способом работы по

принципу ведущего и ведомого¹⁾ пользовательские программы, организованные по принципу ведомого, запрашивают через внутренние прерывания службы защищенных системных программ, таких, как распределители ресурсов.

3. Произошла ошибка. Для этого класса событий типичны как ошибки при операциях ввода-вывода, так и ошибки внутренней адресации и исчезновения/переполнения.

Программный обработчик прерываний должен завершить сохранение состояния прерванного процесса (обычно существующая аппаратура сохраняет недостаточно компонент состояния), определить, который из процессов надо разбудить или какую программу нужно вызвать при интерпретации сообщения, связанного с прерыванием, и разбудить или вызвать выбранный процесс или стандартную программу соответственно. Это требует, чтобы обработчик прерываний снабжался достаточной информацией для определения последнего условия. В простейшем случае первая команда выбранного процесса или программы является как раз той ячейкой, которой передается управление при прерывании. Однако это редко бывает возможным, так как аппаратура в настоящее время не разрешает спецификацию отдельной ячейки для каждого события, вызывающего прерывание.

Пример

На многих ЭВМ пользовательский процесс запрашивает обслуживание ввода-вывода посредством команды вызова супервизора (*SVC*), которая приводит к внутреннему прерыванию и передаче управления обработчику прерываний *SVC*. Последний будет вызывать требуемую программу ввода-вывода или процесс, будить его или посыпать ему сообщение (например, вставкой в очередь); элементы запроса должны быть переданы из пользовательского процесса обработчику прерываний *SVC*, чтобы он нашел правильную стандартную программу ввода-вывода, и программе ввода-вывода так, чтобы была определена правильная команда ввода-вывода. Стандартная программа ввода-вывода в свою очередь должна передавать свой идентификатор (например помещая его в некоторую заранее известную ячейку) обработчику прерываний. Внешнее прерывание, сигнализирующее о завершении операции ввода-вывода, вызовет обработчик прерываний ввода-вывода; этот обработчик затем определит необходимый процесс ввода-вывода (стандартную про-

¹⁾ В литературе известно другое определение способа реализации этого принципа: «привилегированный и непривилегированный режим». — Прим. перев.

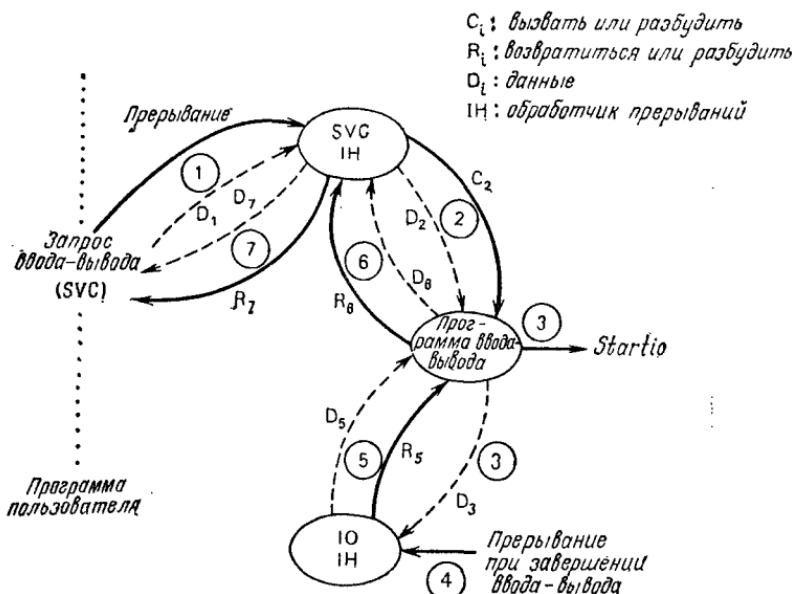


Рис. 4.5. Управление прерываниями в последовательности ввода-вывода.

граммой), который нужно разбудить (вызвать). Стандартная программа ввода-вывода в свою очередь может вернуть или послать сигнал пробуждения обработчику прерываний *SVC*, который, в конце концов, передает управление исходному запрашивающему процессу. Рис. 4.5 иллюстрирует возможную последовательность событий; порядок следования показан числами в кружках. Обработчик *SVC* прерываний, стандартная программа ввода-вывода, а также обработчик прерываний ввода-вывода могут рассматриваться или как отдельные процессы или как часть вызывающего их процесса.

Один из наиболее трудных моментов при проектировании обработчиков прерываний и стандартных программ первого уровня, которые они пробуждают или вызывают, — это вопрос *приоритетов* прерываний. Для конкретного процесса обработчика прерываний возникает вопрос — какие события, вызывающие прерывание, если такие существуют, должны быть разрешены, чтобы приостановить его, и в какой точке (точках) во время его выполнения? Другими словами, где должны быть разрешены, блокированы и запрещены прерывания различных типов, и, если доступно программное управление приоритетами, каковы должны быть эти приоритеты в любое данное время?

Общей процедурой является размещение прерываний и стандартных программ их обработки иерархическим приоритет-

ным способом так, чтобы программы на уровне i могли быть приостановлены только прерываниями, связанными с любым уровнем j , $j > i$. Это часто влечет за собой программное или аппаратное помещение в магазин (стек) или извлечение из него сигналов о прерываниях и векторов состояния по мере продвижения по иерархии. Альтернативным решением является представление обработчиков прерываний как критических секций, причем внутри секций прерывания запрещаются.

4.4. Пользовательский интерфейс

4.4.1. Командный и управляющий языки

Компоненты операционной системы непрерывно вызываются исполняющимися программами через такие запросы на обслуживание, как запросы на ввод-вывод или распределение ресурсов, а также в ответ на прерывания и ошибки. На более высоком уровне декларативная и императивная информация о программах и файлах, включая файлы, принадлежащие системе, связывается посредством *командных и управляющих языков* — CCL (называемых также языками *команд*, языками *управления*, или языками *управления заданиями*). Пользователи системы применяют эти языки для передачи спецификаций и запросов, касающихся их работ, ресурсов и файлов, а операторы ЭВМ используют подобные языки для управления и вмешательства в размещение аппаратуры и программных ресурсов и для ответа на системные сообщения и ошибки.

Часто один и тот же базовый синтаксис может быть предоставлен как пользователям, так и операторам ЭВМ, как для пакетных, так и для интерактивных пользователей. Самая простейшая и „традиционная” форма предложения в CCL есть

<command> *<parameter list>*

где *<command>* задает имя команды или декларацию, которую надо обработать, и *<parameter list>* содержит список параметров или аргументов, связанных с командой. Общепринято, чтобы полю команд предшествовал один или более специальных знаков, таких, как */,\$,!#*; назначение этих символов — однозначно идентифицировать предложение, принадлежащее CCL в отличие от записей данных другого типа, которые могут также появиться во входном потоке. В некоторых из более развитых языков предложения CCL могут быть также помечены для разрешения будущей ссылки из других предложений и системы. Наконец, в конце каждого предложения обычно разрешаются произвольные комментарии. Таким образом, типичный командный язык подобен по виду усложненному языку макроассемблирования.

Так как CCL есть первичный и общий интерфейс для *всех* пользователей с вычислительной системой, важно, чтобы при его проектировании были учтены человеческие факторы. Основной и очевидной причиной упоминания этого момента является то, что CCL были и все еще являются исключительно и необоснованно громоздкими. Они трудны для изучения и использования; запросы на простое обслуживание требуют сложных последовательностей неясных программ и параметров; предложения слишком негибкие по синтаксису.

Пример

Когда большинство пользователей представляют программу на ФОРТРАНе, они рассматривают виртуальную ФОРТРАН- машину, которая непосредственно интерпретирует эту программу (и обслуживающий персонал ЭВМ совершенно справедливо поощряет их в этом); их обычно *не* интересует тот факт, что программа будет компилироваться, редактироваться, загружаться и исполняться, и их не интересуют многочисленные подпункты, связанные с этими четырьмя операциями. Следовательно, чтобы выразить простой общий запрос, можно просто написать предложение вида

(1) **\$RUN FORTRAN**

вместо использования более сложной последовательности, например

(2) **\$EXECUTE FORTRAN(OPTIMIZED, VERSION = 10),
INPUT = CARDRDR, OUTPUT =
= (DISK, NAME = FORTOBJ)
\$LINKANDLOAD FORTOBJ, FORTRANRUNTIME,
NAME = MYPROG
\$RUN MYPROG, INPUT = CARDRDR, OUTPUT =
= PRINTER**

В то же время для опытного пользователя должны быть доступны более детализированные средства управления, такие, как показано в (2), но с более удобочитаемым синтаксисом.

„Программа“ на CCL компилируется редко или вообще никогда не компилируется; она *интерпретируется* операционной системой на уровне предложений. Таким образом, еще одним компонентом любой ОС является интерпретатор CCL, который, по существу, анализирует каждое предложение и вызывает соответствующий системный модуль. С другой точки зрения, CCL определяет виртуальную машину для пользователя; что касается пользователя, то операционная система является для него

машиной с двухшаговым циклом:

- (1) *Get next CCL statement S.*
- (2) *Interpret S.*

Действительно, процесс на самом высоком уровне, связанный с пользовательским заданием, есть часто усложненная версия приведенного выше цикла. Аналогично, существует процесс, отвечающий за чтение и интерпретацию управляющих команд, выдаваемых с консоли оператора.

4.4.2. Управление заданием

Первым предложением любого интерактивного или пакетного задания является команда CCL, которая идентифицирует пользователя и задание и „заносит его в журнал“ системы. Аргументами такой первоначальной процедуры „подключения к системе“ являются имя пользователя, имя задания, учетный номер, приоритет, ограничения на ресурсы (например, максимальное время ЦОУ, максимальное число строк, которые будут отпечатаны) и резервы ресурсов (например, памяти, периферийных устройств). Аналогично, сеанс консоли и (или) задания прекращается по явной или неявной команде CCL „конец передачи“; безусловное завершение происходит, например, когда пользователь „вешает трубку“ линии связи (например, телефона) на терминале, когда система заканчивает задание в результате анализа ошибки или когда обрабатывается команда „начало передачи“, прекращая тем самым предыдущее задание.

Основной задачей, которую выполняет виртуальная машина, представляемая операционной системой, является выполнение программ пользователей. Таким образом, любой CCL будет содержать команды, ориентирующие ОС на загрузку и выполнение пользовательских программ и языковых процессоров (например, ассемблеров, компиляторов). В некоторых CCL делается различие между пользовательскими программами и поставляемыми изготовителями языковыми процессорами и другими программами, в то время как в других такого различия не делается. Типичными командами для этой работы по управлению программами являются команды выполнения данной программы, загрузки программы, связывания нескольких программ вместе или вызова языкового процессора. С этими командами могут быть связаны (в качестве параметров или отдельных предложений): запросы на резервирование ресурсов и ограниченные данные, аналогичные используемым для предложения „начало передачи“ (sign on); аргументы, специфичные для конкретной вызываемой программы, такие, как имена файлов, которые будут использоваться программой; запрашиваемые службы отладки, такие, как трассировки и дампы.

Третий главный набор предложений CCL относится к обработке файлов. Эти предложения выполняют такие основные функции, как идентификация первичных вводных и выводных файлов, которые должны использоваться программами, запрос на распределение устройств ввода-вывода и описание файлов, на которые ссылаются пользовательские программы. В зависимости от уровня управления, с которым работает пользователь, существует потенциально настроенный ряд детальных параметров, соответствующих файловым командам. Вот примеры:

1. Диспозиция файла, например временный или постоянный.
2. Защита (управление доступом) данных для нового файла.
3. Параметры буферизации и блокирования.
4. Резервирование пространства во вторичной памяти.
5. Устройства ввода-вывода, которые будут применяться.
6. Файловая организация, например, последовательный или прямой доступ.

Средства для редактирования файла, такие, как слияние, копирование и модификация файлов, также могут вызываться через CCL. Это может быть отдельная система со своим собственным языком, или в CCL могут содержаться все операции редактирования.

В дополнение к приведенным выше основным командам для обработки задания, программы и файла могут присутствовать несколько других типов предложений, особенно в высоконактивной среде. Например, часто существуют команды, которые позволяют пользователю приостанавливать, изучать, изменять и возобновлять некоторый исполняющийся процесс, а также предложения, которые позволяют программисту определять ряд недостающих параметров для всех его заданий. Большинство крупных систем также обеспечивает некоторый тип определений макро или процедур CCL, так что группа предложений CCL может кратко вызываться как группа. В примере в начале разд. 4.4 предложение (1) могло быть вызовом макро, которое автоматически транслируется в последовательность (2).

Примеры

1. IBM OS/360 (IBM, 1965). Язык управления заданиями для IBM OS/360 имеет относительно простой набор команд с довольно сложной системой параметров команд. По существу, имеются только три основные команды:

- a) Предложение (*JOB*) ограничивает начало задания.
- b) Исполнительная команда (*EXEC*) указывает на начало шага задания и определяет или программу, которую надо выполнить, или „кatalogизированную” процедуру, которая в свою

очередь содержит серию управляющих предложений, определяющих несколько шагов.

с) Предложение описания данных (*DD*) описывает файлы для каждого шага задания и запрашивает ресурсы для обработки файла.

Большинство трудностей в этом языке относится к параметрам для *DD*-предложения.

2. MULTICS (1967). Это система ориентирована главным образом на интерактивное использование. Каждая команда CCL транслируется в вызов процедуры, который может возвращать значение; из-за этой ясной однородной интерпретации пользователи могут расширять язык добавлением своих собственных команд и соответствующих процедур. Аргументы команды оцениваются системой, прежде чем вызывается связанные с ней процедура. CCL-команды могут быть вложены в поле аргумента так, что оценка аргумента может также привести к вызову процедур команды, чтобы возвратить значение. Доступны следующие классы команд:

- a) Управление заданием и процессом.
- b) Выполнение языкового процессора.
- c) Редактирование и создание файла.
- d) Управление справочником файла и доступом.
- e) Связывание процедур и периферийный ввод-вывод файлов.
- f) Отладка.
- g) Пользовательские вызовы ввода-вывода, которые обходят недостатки стандартных систем.
- h) Резервирование ресурсов.
- i) Учет.

Мы привели только очень краткое описание командных и управляющих языков, опуская много их подробностей. В настоящее время они сильно различаются от системы к системе и полностью связанное и модельное проектирование для таких языков еще не появилось (Боэтнер, 1969). Одним из текущих подходов является комбинирование CCL со стандартным языком программирования высокого уровня так, чтобы пользователь описывал управление операционной системой и определял свои вычислительные потребности на одном и том же языке. В этой области предстоит еще много работы.

4.5. Элементы методологии проектирования

Основными функциями операционной системы являются обеспечение обслуживания пользователей и управление, эффективное распределение ресурсов. Для введения этих функций МС рассматривается как динамический набор взаимодействующих

процессов, которые связываются через общие структуры данных и выполняют управление процессами, ресурсами и файлами для пользователей и для себя. Основными задачами при проектировании МС являются описание пользовательских служб (т. е. виртуальной машины пользователя) и определение процессов и их взаимосвязей для реализации этой виртуальной машины.

Существует много подходов к проектированию МС, но нас интересуют только два из них, которые являются *систематическими и управляемыми* и с успехом применялись при изготовлении реально действующих операционных систем. Это иерархический подход к абстрактной машине, разработанный Дейкстрой (1968а), и методы ядра Бринч Хансена (1971). После выявления основных идей этих двух методов мы представим методологию проектирования, которая в действительности является их комбинацией.

Подход Дейкстры основан на размещении процессов ОС в иерархию, где каждый уровень определяет последовательно более абстрактную виртуальную машину. Под более абстрактной мы имеем в виду то, что чем дальше вверх осуществляется движение по иерархии, тем больше выполняется задач управления ресурсами; таким образом, процессы на одном уровне могут предполагать доступность ресурсов, управляемых процессами на более низких уровнях, или, другими словами, определенные классы ресурсов могут быть проигнорированы на каждом уровне. Например, если динамическое распределение основной памяти выполняется на уровне i , то процессы на уровне $j (j > i)$ могут, по существу, игнорировать проблемы памяти в самом общем случае.

Эта техника была применена для проектирования операционной системы „THE“ в Технологическом университете в Эйндховене (Дейкстра, 1968а). Мультипрограммная система „THE“ имеет следующую иерархию процессов:

Уровень	Задачи процессов на каждом уровне
0	Распределение процессора
1	Управление основной и вспомогательной памятью «контроллер сегментов»)
2	Связь между внутренними процессами и консолью оператора («интерпретатор сообщений»)
3	Буферизация ввода-вывода и управление периферийными устройствами
4	Программы пользователей
5	Оператор

Процессы выше уровня 0 не должны интересоваться числом доступных физических процессоров; процессы выше уровня 1 работают только в „сегментах“ информации независимо от ее размещения в основной или вспомогательной памяти; процессы выше уровня 2 предполагают свою собственную частную переговорную консоль, хотя только одна консоль существует в системе.

Синхронизация достигается аппаратными прерываниями и программными семафорными операциями. Например, прерывание от таймера связано с уровнем 0, а прерывание от операций ввода-вывода при связи со вспомогательной памятью действует как один из сигналов пробуждения для процессов на уровне 1.

Методология иерархических абстрактных машин очень привлекательна. Изолируя функции ОС на основе иерархических уровней и устанавливая ясные и четко определенные каналы связи между уровнями и внутри уровней, можно значительно снизить сложность проекта и отладки; действительно, было сформулировано требование, чтобы логическая правильность системы могла быть доказана до реализации и чтобы реальная программа могла быть исчерпывающе и полностью отлажена (Дейкстра, 1968а). Творческой и трудной задачей проектирования является определение структуры уровня и функции каждого уровня.

Методология, основанная на ядре, была разработана и использована для МС на RC4000 (Бринч Хансен, 1971). Суть состоит в том, чтобы концентрировать усилия на проектировании ядра МС, которое достаточно универсально для конструирования и модификации множества ОС.

Аргументом в пользу такого подхода является то, что стратегии организации и распределения ресурсов в системах непрерывно должны расширяться и модифицироваться в ответ на ввод изменений и требования новых служб и новых способов работы. Бринч Хансен был первым, кто разработал, внедрил и доказал подход к ядру „снизу вверх“, а ядро и базовая МС для RC4000 является одной из самых элегантных существующих систем. Мы не обсуждаем здесь никаких деталей, так как в гл. 7 рассматриваются спецификации комплексного ядра.

Причины, разработанные Дейкстрой и Бринч Хансеном, а также опыт других разработчиков легли в основу направлений проектирования МС. Мы приводим их не как точную и жесткую формулу, которой можно механически следовать, но как рациональную основу для приближения к сложной задаче, которая в настоящее время является отчасти искусством, отчасти инженерным расчетом, отчасти наукой и отчасти управлением. Наши элементы методологии проектирования могут быть сформулированы как последовательность главных шагов, которые должен сделать проектировщик.

1. Определите виртуальную машину пользователя. Определяется внешнее управление заданием или командный язык *L* и описываются службы, выполняемые каждой командой в *E*.

2. Опишите „пути“ заданий пользователя и процессов через аппаратуру и МС. (Логично описать, что система должна сделать прежде, чем определять как она делает это).

3. Определите процессы, необходимые для выполнения задач, выдвигаемых на шагах 1 и 2. Определите функцию каждого процесса и как он взаимодействует с другими процессами, а также структуры данных, требуемые для связи. Разместите процессы в иерархию, следуя Дейкстре, если это возможно.

4. Определите компоненты ядра системы и их структуры данных. Специфицируются стратегии распределения ресурсов, по крайней мере в форме наброска, на шагах 3 и 4¹⁾.

5. Используя моделирование и аналитический аппарат, докажите правильность проекта и предскажите поведение МС.

Процедуры для поддержания системы и восстановления при отказах должны быть спроектированы в то же самое время; они могут рассматриваться как „специальные“ типы пользовательских заданий и процессов, сгенерированных „специальными“ пользователями, и могут быть включены как таковые в приведенные выше шаги. Шаги выполняются в итеративной последовательности, причем результаты одного шага генерируют изменения в проекте на других шагах. Когда проект удовлетворителен, можно приступать к действительному кодированию и реализации.

В настоящее время шаг 5 исключительно труден для выполнения и большинство современных систем полностью обходят фазы коррекции и прогнозирования. На практике моделирование требует почти столько же усилий (и почти такого же объема программ), как и реализация самой моделируемой системы. Одной перспективной методикой, исследуемой в настоящее время, является объединение этапов проектирования, моделирования и реализации так, чтобы изменения в проекте и действительная реализация легко вытекали из моделирования. Основная идея заключается в том, чтобы заставить моделирование развиваться в реальную операционную систему во многом тем же самым путем, как проектируют программу сверху вниз в терминах набора общих блоков или обращений к процедурам, которые в конце концов становятся полностью определенными как элементарные предложения (Цуркер и Рэнделл, 1968; Вейдерман, 1971).

¹⁾ Шаги 3 и 4 настолько взаимосвязаны, что они могли бы выполняться или параллельно или в обратном порядке.

5. Управление основной памятью

Основная память, т. е. память, к которой может быть организован прямой доступ центрального процессора за данными или командами, часто является критическим и ограничивающим ресурсом в вычислительной системе. В этой главе мы рассмотрим основные программные методы и аппаратуру для распределения пространства основной памяти и управления ее использованием.

5.1. Статическая и динамическая настройка адресов

В разд. 2.2 рассматривались возникновение и основные принципы статической настройки адресов. Этот метод *настройки* обычно предопределяет статический алгоритм *распределения* памяти. Вся необходимая основная память для пользовательских программ и данных назначается *до начала выполнения* программы, а все адреса настраиваются так, чтобы отразить это назначение. Говорят, что адреса настраиваются *динамически*, когда настройка происходит во время выполнения, *непосредственно* предшествуя *каждому* обращению к памяти. В этом случае „*эффективные*“ адреса всех команд и данных (результат сложения содержимого указанных индексных и/или базовых регистров с адресными полями) являются настраиваемыми, адреса „*привязываются*“ к реальной машине в последний возможный момент. Динамическая настройка делает возможным динамическое *распределение* памяти; таким образом, становится практически целесообразным назначать и освобождать память для данных и компонентов программы во время ее выполнения.

5.1.1. Аппаратурная настройка адреса

Динамическая настройка обычно выполняется аппаратурой¹⁾, и она скрыта от всех пользователей, кроме некоторых системных программистов. Чтобы проиллюстрировать это

¹⁾ Она, конечно, могла бы быть выполнена в режиме интерпретации программным обеспечением, но в большинстве случаев это слишком неэффективно для практического использования.

положение, мы рассмотрим командный цикл простой одноадресной машины как с динамической настройкой, так и без нее.

Пусть $M[0:m]$ представляет основную память, reg есть регистр общего назначения, ic — счетчик команд, $Address(w)$ вычисляет эффективный адрес команды w , и $Operator(w)$ выбирает код операции oc из команды w , где $oc = 1$ указывает на сложение, $oc = 2$ есть операция запоминания содержимого регистра, и $oc = 3$ обозначает безусловный переход. Без динамической настройки основной аппаратурный цикл команды для такой машины выглядит следующим образом:

```

Next:    $w := M[ic];$ 
           $oc := Operator(w);$ 
           $adr := Address(w);$ 
           $ic := ic + 1;$ 
plus:  if  $oc = 1$  then  $reg := reg + M[adr]$ 
          else
store: if  $oc = 2$  then  $M[adr] := reg$ 
          else
xfer:  if  $oc = 3$  then  $ic := adr$ 
          else
          .
          .
go to Next;
    
```

Если имеется аппаратура динамической настройки адреса, то ic и adr рассматриваются как настраиваемые адреса и отображаются в реальные адреса памяти в момент обращения. Мы обозначаем как $NLmap$ функцию для отображения положений имен (*Name Location map*) (Денис, 1965):

$NLmap$: {перемещаемые эффективные адреса} \rightarrow {реальные адреса памяти}

Чтобы приведенный выше основной алгоритм командного цикла описывал машину с динамической настройкой, в него нужно внести два следующих изменения:

1. Производится замена $M[ic]$ на $M[NLmap(ic)]$ в предложении, помеченному *Next*.

2. Производится замена $M[adr]$ на $M[NLmap(adr)]$ как в предложениях *plus*, так и в *store*.

Функцию $NLmap$ можно рассматривать как аппаратурный блок, помещенный между ЦОУ и основной памятью, через который должны проходить все адреса памяти. На рис. 5.1 по-

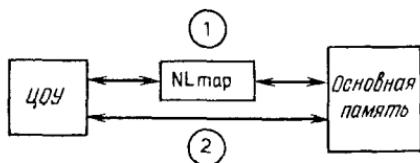


Рис. 5.1. Обращение к памяти при динамическом перемещении.

- (1) : Динамическая настройка адресов
- (2) : Читать или писать

казано такое отображение, выполняемое с помощью аппаратуры; после определения эффективного адреса доступ к памяти состоит из двух шагов — на первом эффективный адрес динамически настраивается с помощью *NLmap*, на втором выполняется сама операция чтения или записи. Во время отображения *NLmap* может обращаться к основной памяти за таблицами и к ЦОУ за специальными регистрами.

Система памяти, которую „видит“ типичный пользователь, т. е. пространство настраиваемых (перемещаемых) эффективных адресов, называется пространством имен или виртуальной памятью. Пространство реальных адресов иногда называют пространством ячеек. Обратите внимание, что если программы или данные перемещаются в основной памяти, то должна быть изменена только функция *NLmap*; адреса пространства имен остаются инвариантными к распределению пространства ячеек.

Статическое и динамическое перемещение имеют свои ограничения и преимущества для МС. Они обсуждаются в следующих разделах.

5.1.2. Аргументы в пользу статического и динамического перемещений

Многие коммерческие мультипрограммные системы работают на машинах без аппаратуры настройки адресов, и для перемещения применяются статические методы. Связывание, перемещение, распределение памяти и часто загрузка программ как единого целого выполняются до того, как процессы переходят в состояние готовности, т. е. когда может логически начаться их развитие. Относительными достоинствами этого способа работы являются возможность использовать стандартную и экономичную адресацию на аппаратурной основе, простота связывающего загрузчика и его взаимодействий с другими системными компонентами и существование большого практического опыта в проектировании таких систем. Однако эти системы обладают некоторыми свойствами, которые

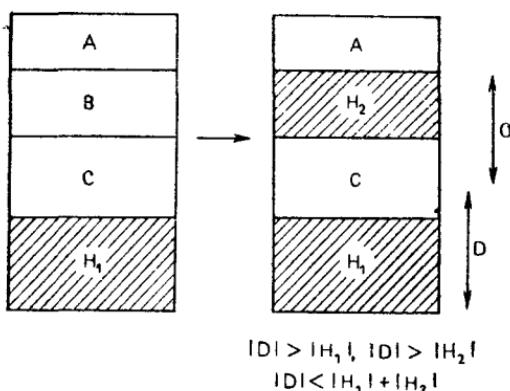


Рис. 5.2. Фрагментация памяти.

препятствуют эффективному использованию ресурсов и которые часто бывают неудобны для пользователей.

Наиболее общей дисциплиной распределения памяти в системе со статическим перемещением является назначение большой непрерывной области памяти каждому пользователю для его задания или шага задания¹⁾; обычно пользователи должны заранее договариваться об определении своих потребностей в отношении памяти. Одним из возможных последствий распределения памяти непрерывными областями в мультипрограммной среде является фрагментация основной памяти, т. е. память может стать подобием шахматной доски из-за неиспользуемых (и часто непригодных для использования) „дыр“. На рис. 5.2 показан простейший случай этой ситуации. На нем H_1 представляет „дыру“, а $|X|$ есть количество памяти, занятой программой или „дырой“ X . Предположим, что программа B заканчивается, образуя „дыру“ между A и C ; если программа D теперь готова к загрузке, то непрерывной доступной памяти достаточного объема не будет, хотя общего количества неиспользованной памяти достаточно. Теоретически назначение непрерывных областей памяти не является необходимым, и области процедур и данных могут размещаться по отдельности; это, однако, значительно усложняет управление памятью и защиту. (Фрагментация возникает в различных формах и в различной степени независимо от дисциплины распределения; в дальнейшем мы вернемся к этой проблеме.)

Типичная МС будет предоставлять средства *перекрытия*, которые позволяют пользователю при выполнении заданий с боль-

¹⁾ Исключением из этого правила является операционная система, которая часто размещается в отдельной области и *разделяется* всеми пользователями.

шими требованиями к памяти сократить объемы ее использования. Основная идея заключается в том, чтобы статически распределять одну и ту же область памяти более чем одной стандартной программе или части программы и сначала загружать только первый сегмент перекрытия. Когда управление передается стандартной программе перекрытия, новая программа переписывается на место некоторых программ, находящихся в этот момент в памяти; действительная загрузка абсолютных кодов („образов памяти“) происходит динамически во время выполнения. Программист-пользователь должен запланировать и идентифицировать свои перекрытия и память для них и включить эту информацию как часть данных для управления своим заданием; это предполагает, что пользователь знает *a priori*, по крайней мере в общем, порядок выполнения подпрограмм в своей программе. В итоге пользователь должен выполнять большую часть процедуры управления памятью самостоятельно. Перекрытия, определяемые пользователем, представляют собой одну из попыток сэкономить основную память, и их широкое распространение отражает как большие потребности многих программ в памяти, так и тот факт, что большинство процессов в течение заданного интервала времени будут использовать только маленький процент своих программ; например, программа часто содержит много стандартных программ обработки ошибок, которые редко используются, но которые тем не менее могут занимать значительное пространство памяти.

В схеме перекрытия программы то *помещаются* в память, то *исключаются* из нее, но любая программа всегда будет помещаться в *одну и ту же* область памяти в течение однократного выполнения. Связывание и перемещение происходит только однажды, в начале выполнения. Можно было бы изобрести более гибкую схему, в которой данная программа могла бы по мере выполнения помещаться в различные области; связывание и перемещение тогда необходимы при каждой загрузке. При этом обычно также требуется, чтобы

- 1) программы не изменяли сами себя,
- 2) никакие абсолютные адреса памяти не встречались в данных,
- 3) неизменяемые программы были бы строго отделены от своих изменяемых данных.

В системах с разделением времени со статическим перемещением из-за приведенных сложностей программы обычно помещаются в одни и те же области.

Статическое перемещение также затрудняет (но не делает невозможным) для пользователей разделение одной и той же копии каждой из процедур в основной памяти (почему?).

обычно для каждого пользователя назначается отдельная копия. Таким образом, если несколько пользователей одновременно выполняют компиляцию с ФОРТРАНом, в памяти может находиться несколько копий одного и того же компилятора с ФОРТРАНом. Однако могут быть реализованы специальные случаи разделения, причем наиболее очевидными среди них являются случай разделения части операционной системы и, возможно, резидентных компиляторов, но *каждый* случай разделения обычно явно вносится в первоначальный проект МС; это или более поздняя модификация МС, или полностью независимая подсистема.

Основные аргументы в пользу динамического перемещения состоят в том, что в принципе оно делает возможным гибкое и эффективное использование основной памяти и в то же время создает удобный для пользователей интерфейс в виде виртуальной памяти. При более сложной аппаратуре для настройки адресов распределение памяти может быть выполнено динамически *по требованию*, а не статически перед выполнением; при этом система имеет возможность задерживать назначение памяти до момента первой адресной ссылки к блоку команд или данных. Следовательно, подобным же образом откладывается связывание и загрузка конкретной процедуры, и, если это необходимо, память, к которой не было обращений в недавнем прошлом, может быть освобождена. Шахматное разбиение памяти может уменьшиться потому, что дисциплина распределения непрерывными областями для каждого пользователя не является необходимой, и единицы распределения могут быть маленькими. (Это свойство сохраняется для дисциплин статического и динамического *распределения*.) В пределах одного прохода процедура может быть легко помещена в различные области памяти или вытеснена оттуда. При динамическом перемещении также легче разделять одну копию процедуры несколькими процессами. Наконец, можно предоставить пользователю большое непрерывное пространство виртуальной памяти и освободить его от каких-либо задач управления, относящихся к перекрытию и учету памяти. В этом случае система несет почти полную ответственность за управление памятью и, как считают, может выполнять эту функцию более эффективно, чем индивидуальные пользователи. (Приведенные выше аргументы несправедливы, если доступен только простейший тип аппаратуры для настройки адресов, — например единственный регистр смещения; основным достоинством этого довольно примитивного типа является способность легко перемещать пространство пользователя в различные области основной памяти.)

Эти преимущества должны быть сопоставлены с дополнительной стоимостью и сложностью как аппаратуры, так и про-

граммного обеспечения. Действительно, можно значительно ухудшить МС, если дисциплины динамического перемещения не были тщательно отобраны и проанализированы. Мы поговорим об этом позднее.

Упражнение

Почему условия, перечисленные в списке (5.1) этого раздела, являются необходимыми?

5.1.3. Типы виртуальной памяти

Организация виртуальной памяти (ВП) или пространства имен зависит от аппаратуры отображения, *NLmap*, которая выполняет преобразование пространства имен в пространство ячеек. Однако удобно описать две принципиальные организации ВП прежде, чем мы рассмотрим их реализацию¹⁾.

Простейшей и самой очевидной формой ВП является непрерывное линейное пространство, соответствующее нашей обычной точке зрения на память. Виртуальная память — это большая, линейно адресуемая последовательность элементов (слов, байтов, ...) с адресами, обычно образующими последовательность $0, 1, 2, \dots, n - 1$, где $n = 2^k$. Мы называем это *односегментным* пространством имен.

Многосегментная ВП разделяет пространство имен на набор сегментов S_i , где каждый S_i есть непрерывное линейное пространство. *Сегмент* — это определяемый пользователем объект, который может рассматриваться как логическая независимая единица, например процедура, алгольный блок или массив данных. Можно также рассматривать программный сегмент как тексты, которые становятся или являются перемещаемым объектным модулем. Адреса могут быть заданы в форме пары $[s, w]$, где s представляет идентификатор сегмента, а w — идентификатор слова (имя или число).

Примеры адресов ВП

1. [5, 927]: слово 927 в сегменте 5. Это могло бы быть результатом вычисления эффективного адреса непосредственно перед динамическим перемещением.

2. [MATRIX, 315]: 315-е слово в сегменте, названном *MATRIX*. Адреса в перемещаемой объектной программе обычно имеют эту форму.

¹⁾ Рэнделл и Кюнер (1968) исследовали виды организации и вопросы, рассмотренные в этом разделе.

3. [*COMPILER, ENTRYPOINT3*]: это типичное обращение к внешнему сегменту, оно может появиться в перемещаемой процедуре или сегменте данных перед статическим или динамическим связыванием.

Иногда можно манипулировать именами сегментов таким же способом, как обычными адресами; таким образом, некоторая функция могла бы быть применена к сегменту с именем S_i , чтобы породить другое имя S_j ; например, если S_i целое, то f могла бы быть некоторой арифметической операцией, содержащей S_i . Это нарушает до некоторой степени независимость сегментов, и более желательны системы, которые не допускают обработку имен сегментов.

По ряду причин виртуальная память представляется в форме набора логических сегментов. Во-первых, то, что образует сегмент, обычно задается пользователем, когда он определяет области своей программы и данных; таким образом, сегменты являются естественными единицами, с помощью которых пользователь обращается к своей информации. Как следствие операционным системам удобно осуществлять доступ к информации пользователя, пользуясь ее распределением по сегментам. Разделение информации (имеется в виду разделение программ и данных), защита от некорректного доступа и ошибок адресации, распределение памяти, а также трансляция с языка могут осуществляться системой на основе сегментной организации. Так как сегменты почти независимы друг от друга, часто бывает возможным разрешать в системах с динамической памятью увеличение и сокращение числа сегментов по мере исполнения при условии, что ВП достаточно большая.

Чтобы реализовать ВП, имеется три возможности управления основной памятью. Рэнделл и Кюнер (1968) называют их стратегиями *выборки, размещения и замены*.

Стратегия *выборки* определяет правило — когда и в каком объеме производить выборку информации из ВП за один раз. Одна крайность состоит в том, что статический связывающий загрузчик загружает все содержимое ВП перед выполнением; другая крайность заключается в том, чтобы осуществлять загрузку в последний возможный момент, т. е. в момент обращения к сегменту или части сегмента. В этом последнем случае (*загрузка по требованию*) должно быть принято решение о том, сколько загружать. Недостатками статической загрузки являются возможная потеря основной памяти и времени на ввод-вывод, когда в пределах данного интервала времени или прогона требуется доступ только к небольшой части ВП. С другой стороны, загрузка по требованию может привести к большим системным накладным расходам и потреблению чрезмерного количества

времени на ввод-вывод, особенно в ситуации, близкой к насыщению, когда во время одного прогона одна и та же программа может быть многократно загружена и замещена.

Стратегия размещения определяет, куда в рабочей памяти (т. е. в памяти, где происходит непосредственное выполнение программ) загрузить содержимое ВП или ее части. Например, можно было бы назначить первую пригодную по размеру „дыру“ или же наименьшую „дыру“, подходящую по размерам. Если имеется несколько уровней рабочей памяти, например быстрая и медленная ферритовая память, то возникает вопрос, на какой уровень надо поместить данную часть программы?

И наконец, в системах, в которых применяются дисциплины динамического распределения памяти, возникает проблема замещения. Здесь мы должны решить, что перемещать или выталкивать из рабочей памяти, когда нет достаточного места для массива информации, который должен быть загружен; интуитивное решение — выбрать ту область памяти, к которой с наименьшей вероятностью будут обращения в ближайшем будущем. Мы рассмотрим различные решения этих проблем в контексте специфической аппаратуры для динамической настройки адресов.

5.2. Принципы сегментации и страничной организации

5.2.1. Односегментное пространство имен

5.2.1.1 Непрерывное распределение памяти

В одном из самых ранних методов динамического перемещения односегментное пространство имен отображалось в непрерывную зону основной памяти. Это отображение может быть выполнено посредством регистра перемещения, скажем *RR*, который всегда содержит адрес базы пространства ячеек для процесса, в настоящее время развивающегося на ЦОУ. Отображение *NLmap* является просто версией, описанной в разд. 2.2.1 процедуры основного статического перемещения, предназначенней для выполнения во время прогона¹⁾:

```
integer procedure NLmap(ea);
comment ea есть эффективный адрес;
NLmap := ea + RR;
```

Для целей мультипрограммирования пространства ячеек (или части этих пространств) для нескольких различных про-

¹⁾ Вплоть до разд. 5.4 мы игнорируем проблему защиты памяти во время отображения.

цессов одновременно могут находиться в основной памяти. Вектор состояний каждого процесса p должен минимально содержать:

1. Счетчик команд ic_p виртуальной памяти.
2. Базовый адрес ba_p реальной памяти (пространства ячеек) процесса p .

Переход от процесса p_1 к процессу p_2 тогда включает следующие условия:

$ic_{p1} := ic$; (Запомнить состояние p_1)
 $ic := ic_{p2}; RR := ba_{p2}$; (Загрузить состояние p_2)

где ic есть машинный счетчик команд (рис. 5.3).

Каждое пространство имен по размеру должно быть меньше или равно основной памяти. Пример фрагментации из разд. 5.1.2, следовательно, подходит и для этого случая. Действительно, фрагментация встречается в той или иной форме почти во всех статических и динамических системах памяти. Основное преимущество использования простого регистра перемещения состоит в том, что программы в процессе свопинга могут легко переходить из одной области памяти в другую; достаточно только изменить базовый адрес пространства ячеек. Программы и данные по-прежнему статически связываются и перемещаются в ВП перед первоначальной загрузкой. В качестве примера приведем IBM 7090, широко распространенную одноадресную машину второго поколения, которая была модифицирована для системы с разделением времени MAC в проекте MIT, чтобы выполнять динамическое перемещение с помощью единственного регистра перемещения (IBM, 1963).

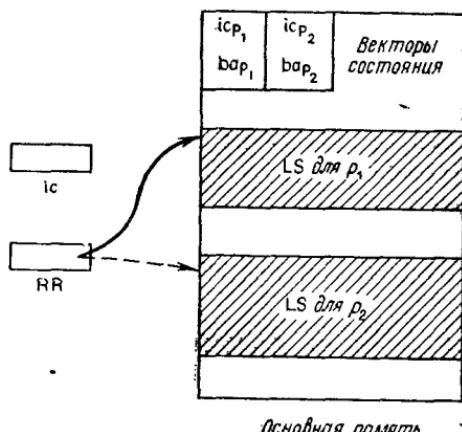


Рис. 5.3. Переключение процессов.

Если имеется несколько регистров перемещения, требование непрерывности пространства ячеек уже не является необходимым и могут быть реализованы более общие сегментные организации ВП. Следует заметить, что регистры перемещения как часть *NLmap* не должны быть доступны обычновенной программе пользователя. (Почему?)

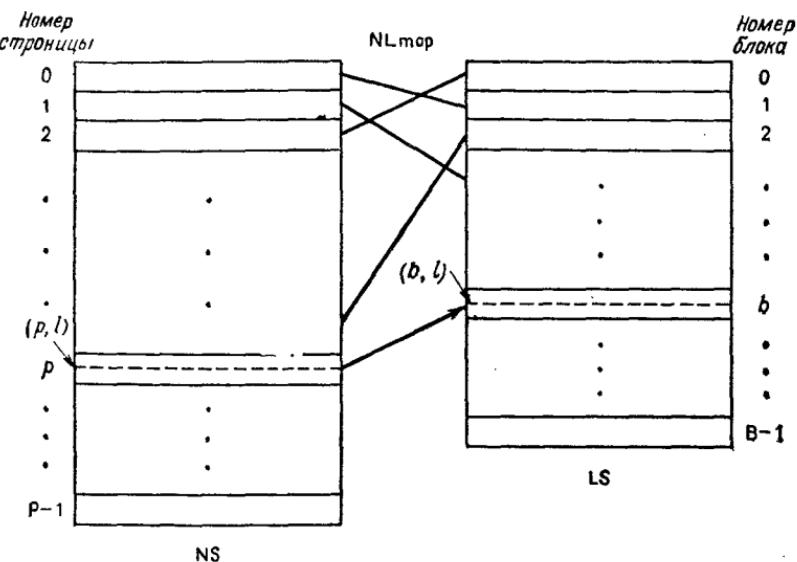
Упражнение

Во многих ЭВМ базовые адреса запоминаются в регистрах, которые участвуют в вычислении эффективного адреса. Базовые регистры могут быть интерпретированы и как регистры перемещения. Если желательно использовать эти регистры для динамического перемещения, чтобы сделать возможным, например, различные распределения памяти для одной и той же программы во время выполнения, то какие ограничения должны быть наложены на программы пользователей? Как их можно обойти? Представьте ваш ответ в терминах некоторой конкретной машины, например IBM/360.

5.2.1.2. Страницчная организация

Термин *страницчная организация* используется, чтобы описать конкретную реализацию виртуальной памяти и структуру основной памяти. Последняя разделена на ряд непрерывных блоков или участков равной длины b_0, b_1, \dots, b_{w-1} ; размер блока обычно 512 или 1024 слов. Абсолютный адрес памяти a указывается парой (b, l) , где b есть номер блока и l — номер слова. Например, если размер блока равен 2^m , то обычно, b — целая часть от $(a/2^m)$ и $l = a - b \times 2^m$. Аналогично, сегмент пространства имён разделяется на некоторое число непрерывных страниц одинакового размера p_0, p_1, \dots, p_{P-1} , где размер страницы равен размеру блока. (Это утверждение не абсолютно; ВП прежде всего выбирается по размеру так, чтобы $|V\!P|$, размер пространства имён, был кратен размеру блока.) Эффективный адрес ea , т. е. адрес ea в ВП, рассматривается как пара (p, l) , где p — номер страницы, а l — номер строки или слова в ней; соответствие между ea и (p, l) получается аналогично полученному ранее соответствуанию между a и (b, l) . Тогда *NLmap* связывает страницу ВП с реальным блоком памяти, как показано на рис. 5.4.

В системе со страницочной организацией основной единицей распределения памяти является блок. Не требуется, чтобы программы и данные занимали непрерывные области памяти, вместо этого они могут быть распределены по ней на страницной основе. Таким образом, если выдано требование на k блоков памяти и есть по крайней мере k свободных блоков, решение о размещении очень простое — любой блок может быть назначен любой странице. Можно ошибочно заключить, что страницная организация устраивает проблему фрагментации. Конечно,



$$\text{NLmap } ((p, l)) := (b, l);$$

Рис. 5.1. Страницчная организация.

верно, что при этом мы не сталкиваемся со специфической трудностью, отображенной на рис. 5.2. Однако, так как размер информационных единиц в виртуальной памяти редко бывает кратен размеру блоков, то обычно существует много блоков, которые только частично заполнены информацией из ВП. Таким образом, существует проблема „внутренней“ фрагментации (мы анализируем эту проблему в разд. 5.5). Большинство реализаций страницочной организации включают механизмы защиты, которые сигнализируют об „отсутствующей странице“, когда производится обращение к странице, не находящейся в основной памяти. При этом операционная система может обеспечить пользователей средствами получения больших пространств виртуальной памяти; для пространства имен допустим размер, гораздо больший, чем размер основной памяти. Динамическое распределение памяти может быть реализовано загрузкой страниц при первом обращении к ним, а не статически. Эта дисциплина называется *страницочной организацией по требованию*.

Функция *NLmap* переводит адрес виртуальной памяти (p, l) в реальный адрес $(b, l) = b \times 2^m + l$, где размер блока равен 2^m . Один из подходов к реализации *NLmap* заключается в том, чтобы поддерживать в памяти таблицу, содержащую B элементов, которая устанавливает соответствие между каждым блоком и страницей. Пусть эта таблица есть массив $pn[0 : B -$

— 1], где $pn[b]$ содержит или номер страницы, хранящейся в блоке b для текущего процесса, или некоторое указание на то, что блок b не распределен текущему процессу. В предположении, что каждая адресуемая страница находится в основной памяти, $NLmap$ можно записать в следующем виде:

```
integer procedure NLmap((p, l));
begin
  for b := 0 step 1 until B - 1 do
    if pn[b] = p then go to adr;
    adr: NLmap := b × 2↑m + l
end
```

Такой последовательный поиск, как приведенный выше, крайне неэффективен, даже если он выполняется аппаратурой, так как функция $NLmap$ должна сделать в среднем $B/2$ сравнений при каждом обращении к памяти. Чтобы этот подход имел практическое значение, таблица запоминается в ассоциативной памяти, и эффективно выполняется параллельный поиск.

Ассоциативная память — это память, в которой к ячейкам обращаются по их содержимому, а не по их адресу. Поясним это понятие на известном примере. Каждая запись в телефонном справочнике адресуется триадой (*страница, столбец, строка*). Чтобы найти телефонный номер какого-нибудь лица, необходимо производить поиск в справочнике пары (*фамилия, имя*); эта задача нетрудная только потому, что записи рассортированы в алфавитном порядке по фамилиям. Однако если мы по номеру телефона захотим найти фамилию или адрес владельца телефона с этим номером, то это почти безнадежная задача, на которую уйдет уйма времени. Хранение телефонного справочника в общей ассоциативной памяти позволило бы осуществлять доступ к любой записи, используя в качестве ключа поиска фамилию или адрес, или номер телефона; иными словами, в записи может быть использовано любое поле, и поиск происходит по его содержимому. Сортировка в этом случае не требуется. Аппаратурные реализации ассоциативных запоминающих устройств не настолько универсальны и обеспечивают обычно поиск только по одному полю в каждой записи. Для моделирования такой памяти имеется стандартный программный метод — кодирование перемешиванием¹⁾ (Моррис, 1968).

Страницчная организация была разработана проектировщиками вычислительной машины ATLAS, которые использовали ассоциативную память для реализации функции $NLmap$ (Киль-

¹⁾ В оригинале „hash coding”. В литературе также используется термин «хеширование». — Прим. ред.

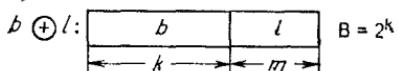
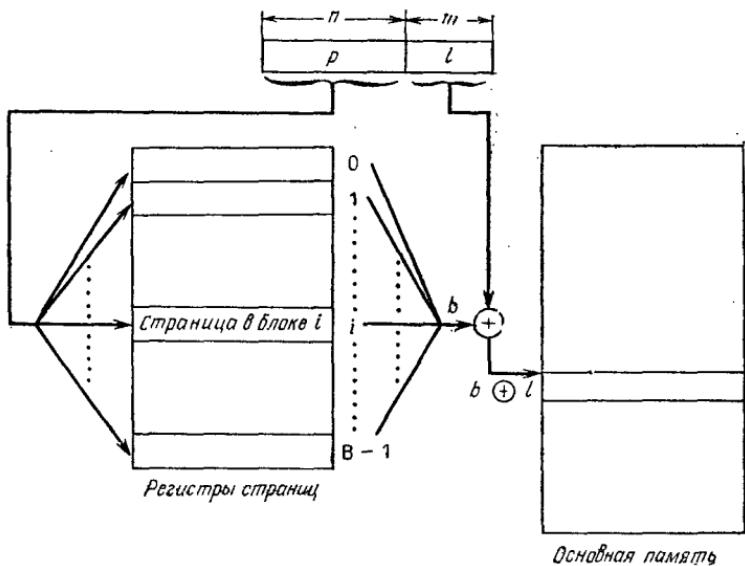


Рис. 5.5. Преобразование адресов в системе ATLAS.

бурн и др., 1962). Эффективный адрес (p, l) есть $(n + m)$ -разрядное число, где старшие n битов соответствуют номеру страницы, а младшие m битов — номеру строки. Номер страницы преобразуется в номер блока b параллельным аппаратным поиском в ассоциативной памяти из B страничных регистров. Конкатенация b с l дает физический адрес памяти (рис. 5.5). Для машины ATLAS $m = 9$ (страница и блок размером в 512 слов), $n = 11$ и $B = 32$; таким образом, была обеспечена виртуальная память объемом 2^{20} слов для машины с реальной памятью в 2^{14} слов (16К). Первоначальная операционная система ATLAS применяла страничную организацию исключительно как средство реализации большой ВП; мультипрограммирование процессов пользователей сначала не предполагалось. И в том и в другом случае вектор состояния процесса должен включать содержимое регистра страниц.

Более современные схемы содержат в основной памяти таблицу страниц для каждого процесса. Любая i -я запись в каждой таблице страниц идентифицирует блок, если таковой существует, содержащий страницу с номером i для этого процесса. Регистр таблицы страниц (PTR) будет содержать базовый адрес таблицы страниц, соответствующей процессу, кото-

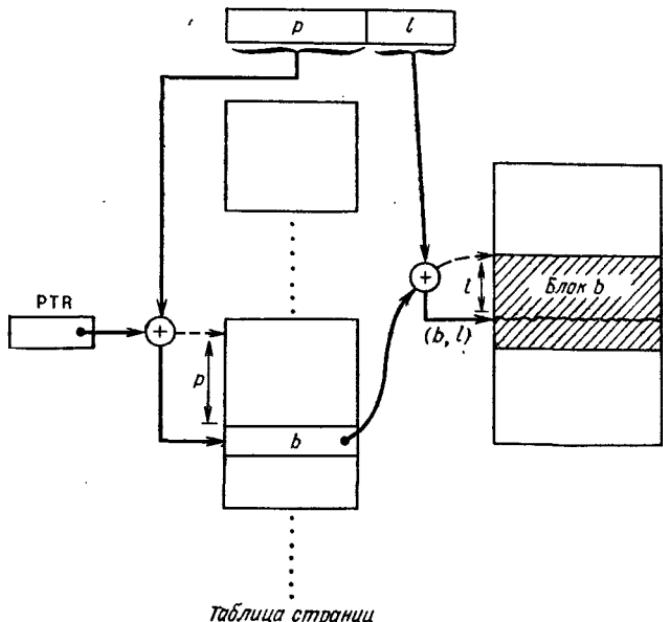


Рис. 5.6. Процедура *NLmap* для чистой страницной организации.

рый развивается в настоящее время на ЦОУ. Процедура *NLmap* вычисляет адрес следующим образом:

$$NLmap((p, l)) := M[PTR + p] \times 2^{\uparrow m} + l,$$

где M обозначает основную память. На рис. 5.6 это отображение представлено в графической форме. Например, предположим, что таблица страниц для текущего процесса была размещена, начиная с адреса 1000, с содержимым

1000: 21; 1001: 40; 1002: 3; 1003: 15; ...;

и размер блока был 1024 слова, тогда регистр PTR содержит 1000, а эффективный адрес, равный $(2, 100)$, динамически преобразуется в $M[1000 + 2] \times 1024 + 100 = 3 \times 1024 + 100 = 312$. Следует заметить, что должно быть сделано по крайней мере *два* обращения к реальной памяти, чтобы прочитать или записать любое слово в ней во время работы с виртуальной памятью — одно обращение к таблице страниц и второе собственно для чтения или записи. Чтобы устранить первое обращение, чаще всего используют или ассоциативную память, или набор регистров для всей таблицы страниц текущего процесса или для ее части. Вектор состояния процесса должен, конечно, включать в себя адрес таблицы страниц.

Пример

Вычислительная машина XDS 940 (XDS, 1969) является страничной машиной с максимальным пространством имен 16К слов и размером памяти в 64 К слов. Она не имеет большой виртуальной памяти, но использует страничную организацию для обеспечения несплошного распределения памяти и простого свопинга и замещения страниц в основной памяти. Размер блока равен 2К слов. Восемь записей, содержащих номера блоков в таблице страниц пользователя, загружаются в быстрые регистры перед выполнением; преобразование при динамической настройке адресов происходит чаще через эти регистры, а не через основную память. Регистр PTR не является необходимым. Аппаратура страничной организации, имеющаяся на XDS Sigma 7, позволяет иметь большую виртуальную память до 132К слов для каждого пользователя. Она имеет 256 быстрых регистров, которые реализуют отображение аналогично XDS 940; размер страницы в Sigma 7 равен 512 слов.

5.2.2. Многосегментное пространство имен

Существуют два различных подхода к реализации многосегментной ВП. Первый рассматривает сегмент как основную единицу при распределении памяти и предполагает, что память может выделяться динамически блоками переменной длины с динамической настройкой адресов. Второй подход применяет страничную организацию. Адрес виртуальной памяти $[s, w]$ рассматривается как триада (s, p, l) , где номер слова w разбит на пару (*страница, строка*). Реальная память имеет блочную структуру с блоками фиксированной длины и размером блока, равным размеру страницы. На данном этапе не очевидно, которая из этих двух методик является более предпочтительной.

5.2.2.1 Сплошное распределение на сегмент

Память распределяется и адреса настраиваются на сегментной основе. Наиболее известный пример использования этого метода можно найти в ЭВМ Burroughs B5500 и B6500 (Burroughs, 1964, 1967). Это стековые машины, ориентированные на эффективную компиляцию и выполнение блочно-структурированных языков. Сегменты соответствуют естественным единицам исходного языка, таким, как блоки, процедуры и массивы в АЛГОЛе.

Таблица сегментов (в фирме Burroughs ее называют таблицей обращений к программе) содержится в основной памяти для каждого активного пользовательского процесса. Каждая

запись в таблице содержит информацию о положении программы или данных, а также информацию по защите. Регистр таблицы сегментов (*STR*) указывает на таблицу сегментов для процесса, развивающегося в настоящее время на ЦОУ; *STR* аналогичен *PTR*, описанному в предыдущем разделе. Эффективный адрес $[s, w]$ отображается в реальный адрес памяти следующим образом:

$$NLmap([s, w]) := M[STR + s] + w,$$

где мы полагаем, что сегменты нумеруются последовательно и i -я запись в таблице сегментов ($i = 0, 1, \dots$) соответствует номеру сегмента i . Сама таблица сегментов также рассматривается системой как сегмент. В большинстве случаев отображение выполняется более эффективно для текущего сегмента s при хранении $M[STR + s]$ в аппаратурном магазине (стеке).

Основным недостатком этой организации по сравнению со страничной организацией является возможность существования большой неиспользованной памяти (внешняя фрагментация) из-за переменного размера единицы распределения и сложности дисциплины размещения (Деннинг, 1970). С другой стороны, страничные системы часто подвержены серьезной внутренней фрагментации, а в случае страничной организации по требованию — интенсивному вводу-выводу. Опыт систем Burroughs показывает, что может быть достигнута значительная производительность; однако не очевидно, что системы со страничной организацией, использующие дисциплины динамического распределения, являются более эффективными¹⁾.

5.2.2.2 Страницчная организация с сегментацией

Страницчная организация в системе с сегментацией сначала использовалась в больших системах разделения времени общего назначения. Вычислительные машины GE 645 (MULTICS, 1967), IBM 360/67 (Комфорт, 1965) и RCA Spectra 70/46 (Оппенгеймер и Вейзер, 1968) являются примерами машин первого „поколения“, в которых использовалась эта организация.

Память организована блоками фиксированной длины, которые как и прежде назначаются страницам программ и данных. Введен еще один уровень косвенного указания в динамическом отображении. Регистр таблицы сегментов *STR* указывает на таблицу сегментов для текущего процесса; записи в таблице сегментов указывают на таблицы страниц, которые в свою оче-

¹⁾ Читатель должен знать, что это положение является спорным, о котором много говорилось, но имеется мало доступной количественной информации для сравнения.

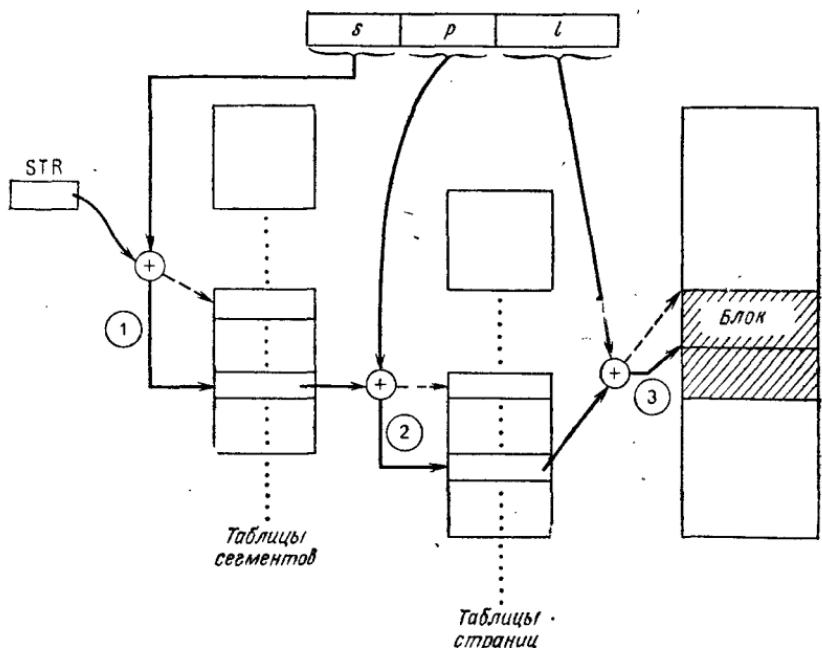


Рис. 5.7. Преобразование для страничной организации и сегментации.

редь указывают на блоки памяти. Все таблицы хранятся в основной памяти. Функция $NLmap$ для эффективного адреса $ea = [s, w] = (s, p, l)$ есть

$$NLmap((s, p, l)) := M[M[STR + s] + p] \times 2^{\uparrow m} + l;$$

как показано на рис. 5.7.

В этих системах каждая таблица сегментов и страниц рассматривается как страница. Каждый процесс имеет свою собственную недоступную другим таблицу сегментов и может иметь или не иметь собственные таблицы страниц. Разделение¹⁾ может быть относительно легко выполнено разделением таблиц страниц и страниц программы или данных. В таком случае страничная организация по требованию становится привлекательной; в принципе она допускает эффективное использование памяти и динамические изменения размера сегмента. Недостатки страничной организации с сегментацией — это дополнительная память, требуемая для таблиц, накладные расходы на управление памятью в такой сложной системе, а также неэффективность двух обращений к памяти при каждом отображении.

¹⁾ Вероятно, автор имеет в виду разделение памяти при реализации режима разделения времени. — Прим. ред.

Примеры

Для трех рассматриваемых машин эффективные адреса (*ea*) разделены на триады (*s*, *p*, *l*). По числу разрядов в каждом поле мы можем легко вычислить размер блока и максимальное число сегментов и страниц на сегмент для каждого виртуального пространства:

<i>ea</i>	<i>s</i>	<i>p</i>	<i>l</i>
GE 645 (адресация на уровне слова)	18	8	10 битов
IBM 330/67 (адресация на уровне байта)	4 или 12	8	12
RCA Spectra 70/46 (адресация на уровне байта)	5	6	12

Эффективность отображения достигается использованием небольшого числа ассоциативных регистров (в случаях GE и IBM) или запоминанием в быстрой памяти „трансляции“ (RCA) соответствия всей ВП реальной памяти для текущего процесса. Ассоциативные регистры содержат записи в форме (*номер сегмента, номер страницы, номер блока*), где *номер блока* определяет блок памяти, соответствующий сегменту и странице. Собственно отображение происходит только в случае, если (*сегмент, страница*) не находится в ассоциативном регистре; в этом случае аппаратурная замена содержимого одного из регистров информацией, соответствующей текущему отображению, происходит так, чтобы будущие ссылки осуществлялись через этот ассоциативный регистр. В реализации RCA не используются отдельные таблицы страниц и сегментов, вместо этого для каждого процесса полный набор элементов (*сегмент, страница, блок*) хранится и загружается в специальную память трансляции, когда управление передается процессу.

5.3. Защита реальной и виртуальной памяти

Средства защиты памяти требуются как для управления доступом процесса к его собственным командам и данным, так и для того, чтобы предотвратить доступ процесса (например, чтение, запись, выполнение) к информации, связанной с другими процессами; иначе говоря, процессы должны быть защищены сами от себя и от других процессов. Очевидным примером является защита операционной системы от перезаписи или от

ошибочного выполнения программами пользователей. Другие примеры реально встречаются в ситуациях разделения, когда имеют дело с информацией, которую имеет право использовать только ее владелец, а также при отладке программы.

Возможно несколько различных способов защиты. Самым основным является тип „все или ничего“. Информация в пределах выделенной процессу области памяти (виртуальной или реальной) может быть доступна для любой цели; адресация вне этой области запрещена и вызывает прерывание, приводящее к выполнению стандартной системной программы. Это может быть реализовано с помощью: регистров *границ*, которые определяют верхний и нижний адреса непрерывной области памяти, индикаторов *длины*, содержащих размер областей памяти, битов *присутствия*, которые показывают, находится ли конкретный сегмент или страница в основной памяти, ключей идентификации, связанных с блоками памяти, или некоторой комбинации этих средств. Далее кратко рассматриваются специфические примеры. Более подробно защита от доступа к области памяти *S* может быть охарактеризована как булева комбинация следующих элементарных типов защиты:

1. *Разрешено чтение (R)*. Содержимое памяти *S* может быть считано. Допустимы команды, которые загружают информацию из *S* в регистр или в некоторую другую область памяти или которые сравнивают элементы *S*; запись в *S* или выполнение в *S*, например, при передаче управления по некоторому адресу в *S*, недопустимы, если вид доступа „только R“ (только чтение).

2. *Разрешена запись (W)*. Разрешено переписывать из регистра или некоторой другой области памяти в *S*.

3. *Разрешено выполнение (X)*. Содержимое области *S* может быть выполнено как программа; обычно допустимы также внутренние чтения констант.

Часто защита типа *R* интерпретируется как включающая также и тип *X*. В терминах этих базовых типов наиболее общеупотребительными формами ограничений на доступ являются:

$\neg(R \vee W \vee X)$: Не разрешен никакой доступ.

$(R \wedge \neg(W \vee X))$: Разрешено „только чтение“. Этим обозначением можно определить таблицу констант или часть файла данных, например архивный файл об успеваемости студентов.

$(R \vee W) \wedge \neg X$:

В такой форме защищается область данных при чтении или записи; например, массив при научных расчетах защищается от попыток выполнения его неправильной программой.

 $\neg(R \vee W) \wedge X$:

Разрешено „только выполнение“. При этой комбинации обеспечивается секретность команд в системных программах и программах, представляющих собой личную собственность. S не может рассматриваться как данные.

 $R \vee X \wedge \neg W$:

Запись не разрешена.

 $R \vee X \vee W$:

Обеспечивается неограниченный доступ.

Информация о защите может быть связана с системой памяти любым из следующих трех способов. Очевидным и, вероятно, простейшим подходом является привязка ее непосредственно к физической памяти; конкретный непрерывный блок реальной памяти в любой момент будет иметь фиксированный набор правил, ограничивающих доступ; правила будут относиться ко всем процессам, которые могут пожелать получить доступ к этому блоку. Однако, если программы или данные можно разделять, это слишком негибко; часто для различных процессов бывает желательно иметь различные возможности по отношению к одной и той же информации.

Примеры

1. Во многих МС пользователь может запросить состояние системы с консоли, чтобы получить такую информацию, как загруженность системы (число заданий или процессов в различных очередях) и состояние своего задания; параллельно с этим происходит также непрерывное обновление информации о состоянии. Тогда тип доступности области данных, определяющих состояние, в общем случае есть $R \vee W$ для обновляющего процесса и R только для системного процесса, который представляет эти данные пользователем¹⁾.

2. Доступ к компилятору, право производства и продажи которого принадлежит какой-то компании, может быть типа „только X “ для покупателя, типа R и X для продавца или системного программиста, который осваивает новые системы в

¹⁾ Здесь мы явно признаем, что программы часто изменяются и редко не содержат ошибок; конечно, процесс отображения данных о состоянии никогда не должен проводить запись в области этих данных, но все же в системе должна быть введена защита ввиду высокой вероятности ошибок.

компании, поставляющей компилятор на рынок, и типа $R \vee X \vee W$ для разработчиков компилятора. Для первых двух из этих классов пользователей в МС можно одновременно иметь доступ к одной и той же копии. (Почему нельзя для всех трех одновременно?)

Если бы информация о защите была непосредственно связана с защищаемой областью физической памяти, то было бы необходимо ее явно изменять каждый раз, когда переключаются процессы; положение значительно усложняется при мультиобработке, так как несколько процессов, которые обращаются к *одной и той же* области памяти, могут развиваться одновременно. Более совершенный метод заключается в том, чтобы рассматривать защиту с точки зрения *пространства имен*; иначе говоря, каждая часть (обычно сегмент) пространства имен для процесса будет иметь свои собственные ограничения на доступ. Защита, однако, не является статической по смыслу даже в пределах одной ВП. В общем случае предпочтительнее, если ограничения на доступ могут меняться динамически как функция состояния процесса.

Пример

Рассмотрим многофазную подсистему „компилировать и выполнить“, которая транслирует и выполняет программу, представленную на некотором исходном языке высокого уровня; пусть существуют n фаз в последовательности трансляции и выполнения, причем выполнение происходит в последней фазе. Мы считаем всю операцию единым процессом. Во время выполнения на фазе i , $i = 1, \dots, n$, доступ к командной части пространства имен, соответствующего фазе i , может быть типа „только X “, в то время как эта часть для фазы j , $j \neq i$, $j \neq n$, не может быть доступной; доступ к области выходных данных, в которой формируется программа на выходном языке, может быть типа „только W “, в то время как тип доступа к таблице символов изменяется от типа R к типу $R \vee W$ в зависимости от фазы. Для фазы n доступ к области вывода имеет тип „только X “. Таким образом, каждая фаза выполнения определяет различный набор ограничений на доступ процессов к виртуальной памяти.

Динамическое управление доступом получается при присыпывании каждому сегменту в пользовательской ВП не только своих собственных ограничений защиты, но также типа доступа, который этот сегмент может иметь к другим сегментам. В порядке увеличения общности информации о защите может быть связана с физической памятью, статически с ВП, а также дина-

мически с ВП. Теперь мы опишем реализацию защиты памяти на нескольких ЭВМ второго и третьего поколения.

1. IBM 7090 с аппаратурой динамической настройки адресов

На этой ЭВМ был реализован тип защиты „все или ничего“; были использованы регистры верхней и нижней границ, UR и LR соответственно, чтобы хранить допустимые верхние и нижние предельные значения адресов. Тело функции $NLmap$ из разд. 5.2.1.1 тогда выглядит так:

```
adr := ea + RR;
if adr > UR ∨ adr < LR then Interrupt
else NLmap := adr;
```

2. Система со страничной организацией ATLAS

Каждый блок ферритовой памяти имеет однобитовый „замок“. Любая попытка получить доступ к „запертому“ блоку приводит к прерыванию. Замок может быть использован, чтобы предотвратить доступ к блоку, который участвует в операции ввода-вывода, а также предотвратить отпирание любых блоков, которые не являются частью пространства реальной памяти текущего процесса. Прерывание также происходит, если addressуется страница, не находящаяся в реальной памяти.

3. Чистая страничная организация в XDS 940

Поле номера блока, связанного с каждой страницей ВП, имеет однобитовый замок, допускающий только чтение, и называемый rol . Если замок заперт ($rol = 1$), то блок используется только для чтения; иначе разрешен любой тип доступа. Поля номеров блоков в таблицах отображения в случае пространств имён размером менее восьми страниц заполняются нулями. Блок 0 резервируется для системы¹⁾.

4. Сегментация в B5500

Каждая запись в таблице сегментов s содержит, кроме прочего, длину l_s сегмента и бит присутствия a_s , показывающий, находится ли сегмент в основной памяти. Для эффективного адреса $[s, w]$, если $w > l_s$ или $a_s = 0$, требуется вмешательство системы, и с помощью аппаратуры $NLmap$ вырабатывается прерывание. (Аппаратурная проверка длины сегмента может быть очень эффективно использована для обнаружения ошибок индексации в языках высокого уровня во время выполнения программы.)

¹⁾ Это не означает ссылки на блок с номером 0, а только является признаком того, что соответствующий блок размещен не в оперативной памяти, а в быстрых регистрах. — Прим. ред.

5. Нестраницные и страницные организации в IBM/360

Защита памяти в стандартной модели IBM/360 (при отсутствии динамической настройки адресов) достигается посредством четырехбитового ключа защиты, который связан с каждым 2К-байтовым блоком основной памяти. Это тип защиты „все или ничего”. Каждый процесс имеет такой ключ как часть своего вектора состояния. При всех доступах к памяти ключ в векторе состояния должен соответствовать ключу блоков; иначе происходит прерывание.

Модель 360/67, которая имеет сегментацию и страничную организацию, имеет дополнительный бит защиты выборки (обозначим его f), усиливающий функции ключа памяти. При соответствии ключа любой способ доступа допустим как и прежде; при несоответствии ключа значение $f = 0$ разрешает только операции чтения, а значение $f = 1$ запрещает доступ, что приводит к прерыванию. Регистр таблицы сегментов и каждая запись таблицы сегментов содержит индикатор длины, который определяет текущий размер таблицы сегментов и таблиц страниц соответственно. Наконец, предусмотрен однобитовый флаг присутствия в каждой записи таблиц страниц и сегментов; этот флаг показывает, находится ли таблица страниц или страница в основной памяти. Мы обозначаем содержимое регистра отображения и записей таблиц следующим образом:

STB, STL : Содержимое регистра таблицы сегментов STR — адрес базы таблицы сегментов и ее длина соответственно.

$PTB[S], PTL[S], PTA[S]$: База таблицы страниц, длины и флаг присутствия для таблицы страниц, соответствующей сегменту, который представлен индексом S .

$BB[P], BA[P]$: Адрес блока и флаг присутствия, соответствующие странице, которая представлена индексом P .

Тогда $NLmap$ для эффективного адреса $[s, w] = (s, p, l)$ есть¹⁾

if $s > STL$ **then** *Interrupt* („Недопустимый номер сегмента“);

$S := STB + s;$

if $PTA[S] = 1$ **then** *Interrupt* („В памяти нет таблицы страниц для сегмента“);

if $p > PTL[S]$ **then** *Interrupt* („Недопустимый номер страницы“);

$P := PTB[S] + p;$

if $BA[P] = 1$ **then** *Interrupt* („Страницы нет в памяти“);

$NLmap := BB[P] + l;$

¹⁾ Мы здесь не будем обращать внимание на ассоциативные регистры.

Таким образом, защита „все или ничего“ обеспечивается на основе виртуальной памяти с ограничениями на доступ типа „только чтение“, связанными с реальным пространством через флаг f и систему ключей памяти.

6. Система MULTICS на ЭВМ GE 645

В ЭВМ GE645 применяются индикаторы длины и флаги присутствия таким же образом, как в IBM 360/67 для первого уровня статической защиты ВП. Записи таблицы сегментов и страниц содержат также поля для всех комбинаций более детализированных типов ограничений на доступ R , X и W и определяют возможный режим управления процессом, использующим сегмент или страницу (процесс ведущий, ведомый или то и другое сразу). Когда эти ограничения нарушаются, происходят аппаратурные прерывания.

Форма динамической защиты ВП обеспечивается также программно и аппаратурно путем использования *кольцевой* схемы (Грэхем, 1968). Целью является разрешить изменяться полномочиям процесса, когда перемещаются его программы и данные по ВП. Процессы в MULTICS развиваются в составе иерархической системы защиты, которая может быть представлена последовательностью концентрических колец; сегменты, находящиеся во внутренних кольцах, защищены в наибольшей степени, в то время как сегменты во внешних кольцах в общем более доступны. На рис. 5.8 показано начальное распределение колец в системе MULTICS; распределения колец закреплены для данного сегмента и представляются как часть записи в справочнике файла каждого сегмента. Операционная система занимает первые три кольца. Наиболее критические части (т. е. ядро) занимают кольцо 0; большая часть системы находится в кольце 1 и остальная часть, состоящая из менее чувствительных сегментов, находится в кольце 2. Кольца от 3 до 32 могут быть использованы процессами пользователей. Пусть \mathcal{S}_i — набор сегментов, находящихся в кольце i в любой момент времени. Основная идея заключается в том, что процесс, развивающийся в любом сегменте $S \in \mathcal{S}_i$, может „свободно“ иметь в своей ВП доступ к сегменту $T \in \mathcal{S}_j$, $j \geq i$, подлежащему защите способом X , R , W для T . Попытка получить доступ внутрь кольца, т. е. $j < i$, приводит к прерыванию, и управление передается системной стандартной программе для проверки допустимости обращения. Фактически вызовы как внутрь ($j < i$), так и наружу ($j > i$) приводят к прерываниям. В обоих случаях, если управление успешно передается из текущего кольца, то номер „выполняющего“ кольца должен быть изменен на номер этого нового кольца, а аргументы должны быть проверены. Например, обращение к процедуре T во внешнем кольце могло бы

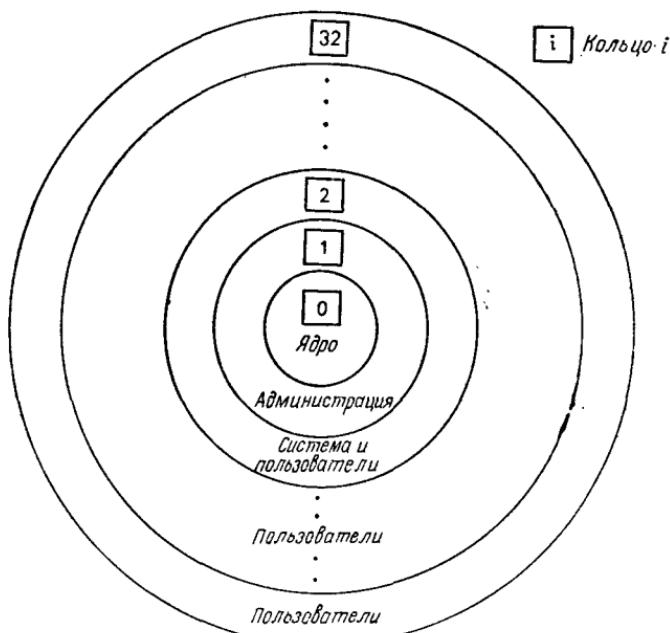


Рис. 5.8. Кольца защиты в системе MULTICS.

включать, в качестве аргументов, адреса переменных в кольце i , т. е. вызывающем кольце; $T \in \mathcal{P}_i$ будет не в состоянии осуществить доступ к этим переменным внутреннего кольца. Решение, которое выбрано, это копировать аргументы в область данных, доступную для T .

7. Корпорация по управлению научными исследованиями SCC6700 (Уотсон, 1970)

Автоматическая динамическая защита ВП реализована посредством матриц путей доступа. Каждый элемент a_{ij} матрицы определяет тип доступа, разрешенный сегменту i к сегменту j ; a_{ij} может принимать значения из набора $\{\neg(R \vee W \vee X), R \wedge \neg(W \vee X), \neg(R \vee W) \wedge X, R \vee W \vee X\}$. Каждый процесс имеет матрицу, определяющую защиту пути доступа к его сегментам. Когда процесс продолжается, небольшая часть из его матрицы запоминается в регистрах, чтобы обеспечить с помощью аппаратуры быструю проверку законности доступа.

В этой ЭВМ также используются механизмы защиты памяти как средство для защиты управления (команд) способом, первоначально предложенным Лэмпсоном (1968). Основная идея — управлять использованием привилегированных команд, выборочно помещая их в „память только для выполнения“, вместо

того, чтобы использовать ЦОУ в стандартных режимах ведущего и ведомого. Процесс использует затем эти вынесенные (out-of-line) команды через команды типа „выполнить“¹). Такая система имеет полное и гибкое управление классами привилегированных команд, которые может использовать частный процесс. Устраняются переходы от режима ведомого к режиму ведущего и некоторые обращения к системным программам.

Каким образом поддерживается и становится известной системе вся информация о защите, связанная с системными и пользовательскими процессами? Ясно, что владельцы различных сегментов должны определять эту информацию при создании или редактировании файлов программ и данных. Файловая система обычно ответственна за верификацию и инициализацию механизма защиты доступа, когда сегменты в первый раз загружаются в память, а также за обслуживание данных защиты; взаимодействие между файловой системой и системой защиты основной памяти рассматривается в гл. 9 (разд. 9.4.3).

Упражнения

1. Можете вы представить себе какие-нибудь другие способы защиты в дополнение к комбинациям R , W , X ?
2. Имеет ли практическое значение защиты «только запись»? Попытайтесь найти некоторые применения.

5.4. Стратегии распределения памяти

5.4.1. Распределение памяти в нестраничных системах

В этом разделе рассмотрены проблемы распределения и освобождения блоков основной памяти *переменного* размера в системах без аппаратуры страничной организации.

Эти операции могли бы быть выполнены двумя командами:

1. *Request(MAINSTORE, (SIZE, BASEADDR))*
2. *Release(MAINSTORE, (SIZE, BASEADDR))*

где *SIZE* — число элементов непрерывной памяти, участвующих в операции, а *BASEADDR*, первый (нижний) адрес блока, является выходным параметром операции *Request* и входным параметром операции для *Release*. Выделенный блок обычно содержит логический сегмент программы или данных некоторого процесса.

¹) Команда типа „выполнить“, скажем в форме *EX OPADR*, будет выполнять команду, расположенную по адресу ее операнда (*OPADR*), а затем будет продолжено выполнение очередной команды, следующей за *EX*, что во многом похоже на косвенную адресацию команд.

При наличии запроса на блок памяти размером k наши действия в общем случае могут быть следующими. Если существует „дыра“ H размером h , такая, что $h \geq k$, то распределяем k единиц из H запрашивающему процессу. (Правило для выбора одной из нескольких конкурирующих дыр достаточного размера было названо в разд. 5.1.3 стратегией размещения.) Если нет „дыры“ достаточного размера, можно выбирать из ряда альтернативных стратегий: 1) запрашивающий процесс может быть переведен в блокированное состояние до тех пор, пока не станет доступной свободная память достаточного объема; 2) могут быть освобождены один или более используемых блоков, чтобы создать „дыру“ соответствующего размера (дисциплина замещения); 3) используемая память может быть уплотнена¹⁾ (передвинута в единую сплошную область) в стремлении создать достаточно большую дыру. Сначала мы рассмотрим две широко распространенные стратегии размещения.

Стратегия размещения типа „первый подходящий“ и „наилучший подходящий“

Пусть основная память разделена на два множества блоков переменной длины: множество пустых блоков (неиспользуемых дыр) $\{H_i | i = 1, \dots, n\}$ и множество (распределенных) заполненных блоков $\{F_i | i = 1, \dots, m\}$. При получении запроса на блок размером k по стратегии размещения типа „первый подходящий“ ищут любую „дыру“ H_i , например первую встретившуюся, такую, что $h_i \geq k$, где h_i есть размер „дыры“ H_i . По стратегии типа „наилучший подходящий“ выбирают ту дыру H_i , которая подходит „точнее“, т. е. $h_i \geq k$ для всех H_i , таких, что $h_j \geq k$, $h_j - k \geq h_i - k$ для $i \neq j$. Для обеих стратегий, когда разность $h_i - k$ мала и ненулевая, предпочтительно распределить всю „дыру“ H_i , а не только k единиц; в противном случае вновь созданная „дыра“ размером $h_i - k$ может быть слишком мала, чтобы иметь какую-либо ценность в будущем, и может привести к потере времени и пространства при последующих действиях по управлению памятью. Стратегия „первый подходящий“ — быстрая стратегия, но интуитивно видно, что она расходует большие „дыры“, прежде всего распределяя их части. Стратегия „наилучший подходящий“ оставляет большие дыры для использования в будущем, но является более медленной; она также имеет тенденцию создавать много маленьких и бесполезных дыр. Легко привести естественные примеры, в которых любой из двух методов превосходит другой.

¹⁾ Если применяется сжатие, то информация в памяти не должна быть зависимой от положения; например, не должны присутствовать абсолютные адреса памяти.

Примеры

Для простоты мы рассмотрим только последовательность запросов на память и распределений и предположим, что блоки памяти не освобождаются.

1. Последовательность запросов, при которой лучше работает стратегия „наиболее подходящий“:

Запрос на память	Размер «дыр» в доступном пространстве	
	«первый подходящий»	«наилучший подходящий»
(1) Начальное «состояние»	2000, 1500	2000, 1500
(2) 1200	800, 1500	2000, 300
(3) 1600	процесс заблокирован	400, 300

2. Последовательность запросов, при которой лучше работает стратегия „первый подходящий“:

Запрос на память	Размер «дыр» в доступном пространстве	
	«первый подходящий»	«наилучший подходящий»
(1) Начальное «состояние»	2000, 1500	2000, 1500
(2) 1200	800, 1500	2000, 300
(3) 1400	800, 100	600, 300
(4) 700	100, 100	процесс заблокирован

Эксперименты с обоими этими методами в широком диапоне моделируемых данных были выполнены Кнутом (1968). В большинстве случаев система со стратегией „наилучший подходящий“ прекращает работу раньше (процесс „блокирован“ в приведенных выше упрощенных примерах) из-за недостаточности памяти. Моделирование показывает, что нельзя всегда полагаться на интуицию. Следует также заметить, что наибольшее время до прекращения работы не обязательно является наилучшим критерием, который следует использовать; например, могут быть получены различные результаты, если из-за неудовлетворенных запросов процессы помешались в блокированное состояние. Прежде чем рекомендовать какую-нибудь конкретную стратегию для МС, следует провести расширенные эксперименты, использующие реалистические статистические распределения запрашиваемого размера, моментов поступления запроса и „времен жизни“ распределений памяти.

Метод „первый подходящий“ при поиске доступного пространства может быть ускорен с помощью простого приема (Кнут, 1968). Предположим, что данные о „дыре“ (размер и адрес) хранятся в списке. Прямой поиск мог бы всегда начинаться с вершины списка и продолжаться последовательно, пока не будет найдена подходящая „дыра“. К сожалению, это ведет к концентрации меньших „дыр“ вблизи начала списка, так как распределения выполняются сначала там, если они возможны; с течением времени тогда требовалось бы все больше и больше времени для получения подходящей „дыры“. Эффективная схема, которая бы обеспечивала более равномерное распределение размера „дыр“ по списку, заключается в том, чтобы начинать поиск в некоторой переменной точке Q внутри списка. Одна из возможностей для указания Q состоит в том, что при успешном поиске следующий номер следует за выбранной „дырой“.

Управление доступным пространством

Сначала исследуем структуры данных, предназначенных для хранения информации о доступном пространстве, или „дырах“. Нашей целью является получение структуры, в которой могут быть удобно выполнены поиски по стратегиям „первый подходящий“ и „найлучший подходящий“ и могут быть добавлены новые „дыры“, по мере освобождения памяти. В первых рассматриваемых методах вместо выделения специальной области памяти для структуры данных мы будем использовать непосредственно пространство „дыр“.

Рассмотрим прямое использование простого односвязного списка (рис. 5.9). Первые ячейки в каждом пустом блоке („дыре“) содержат длину блока и указатель на следующую „дыру“. Указатель начала списка (обозначим его $AVAIL$) указывает на первый блок списка, указатель A указывает на последний элемент списка. Следует заметить, что даже если мы начинаем с цепочки указателей, упорядоченной по адресам памяти, например, в восходящей последовательности, то упорядочение быстро нарушится, когда „дыры“ добавляются в произвольные места в списке (если, конечно, мы не побеспокоимся об установлении указателя на каждую новую „дыру“, на соответствующее ему по упорядоченности место в списке). Возникает следующая трудность. Блок, который освобождается, может быть с любой стороны смежным со свободным или с заполненным блоком (рис. 5.10). Желательно объединить при освобождении свободные блоки, но, чтобы такое объединение было осуществимо, необходимо иметь возможность без труда узнавать как расположены блоки, окружающие освобождаемый блок. По

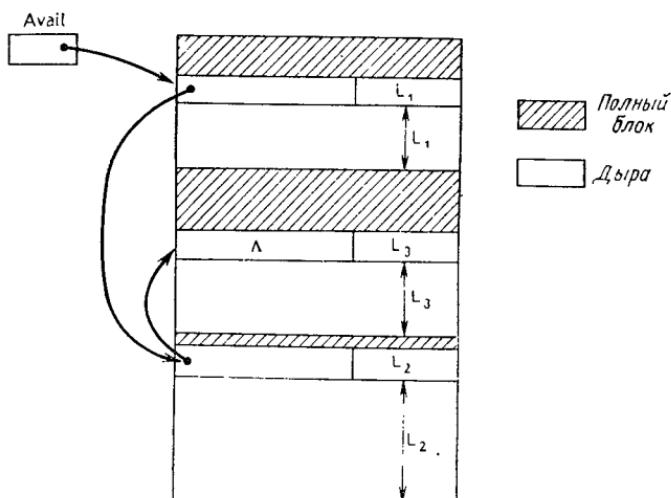


Рис. 5.9. Управление доступным пространством, использующее односвязный список.

структуре данных, изображенной на рис. 5.9, было бы очень неудобно определять, является ли „дыра“ или заполненный блок смежным для любого данного блока. (Как можно было бы это сделать?) Если список доступного пространства хранится в упорядоченном виде, то объединение может быть выполнено легко, но сначала список должен быть последовательно просмотрен, чтобы найти окружающие „дыры“, если такие существуют.

Эти проблемы устраняются „прикреплением этикетки“ на верхней и нижней границе каждого заполненного и пустого блока и соединением „дыр“ в цепочку с помощью двусвязного списка. Поместим однобитовую этикетку (она предполагается здесь булевой переменной) и размер блока в качестве первых двух и последних двух элементов каждого блока; этикетка **true** означает „дыру“, а этикетка **false** указывает на заполненный блок. Кроме этого, в начале каждой „дыры“ хранится прямой

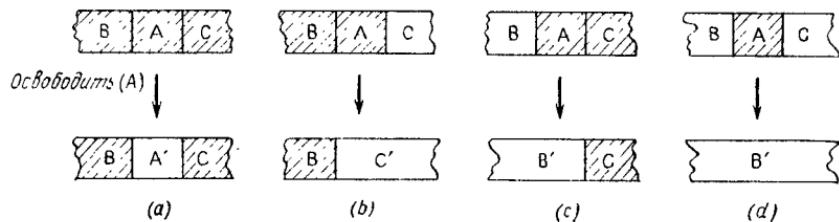


Рис. 5.10. Соединение дыр при освобождении.

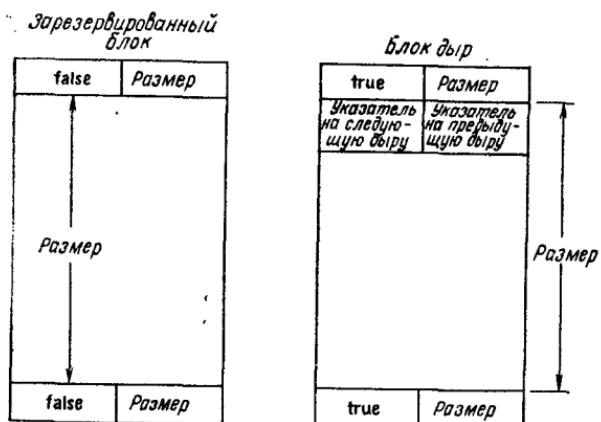
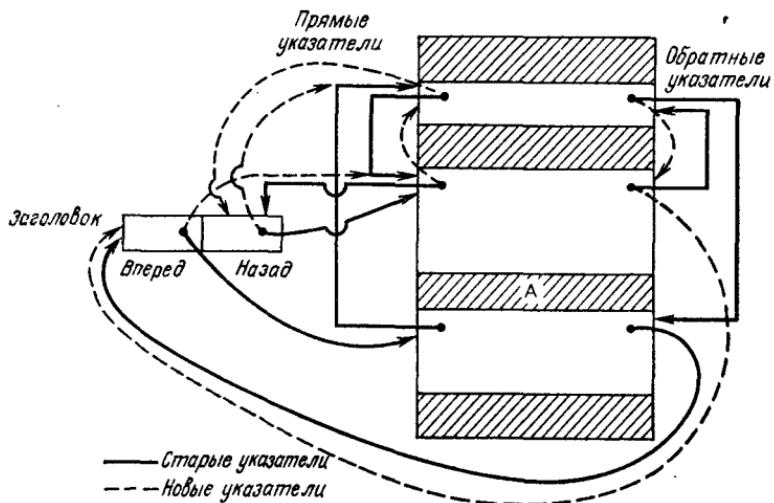


Рис. 5.11. Структура данных с ограничительными этикетками.

Рис. 5.12. Выполнение команды *Release(A)*.

и обратный указатель как часть цепочки „дыр“ (рис. 5.11). Изменения указателя при освобождении блока в случае (d) на рис. 5.10 указаны на рис. 5.12. Новые „дыры“ вставляются в начало списка, так как это наиболее удобное доступное место, особенно в случае (a) на рис. 5.10. (Это относится также и к концу списка.) Эта схема с „этикетками границ“ была предложена Кнутом (1968).

Связные списки не всегда являются лучшим средством для учета и просмотра доступного пространства. Есть память рас-

пределяется непрерывными последовательностями блоков *фиксированного размера*, то более эффективной может оказаться отдельная таблица, содержащая „битовый“ план. Например, предположим, что память для целей распределения разбита на блоки размером 32К слов и имеется 256К слов (8×32) основной памяти. Тогда состояние распределения памяти может быть представлено битовой строкой: $B = b_0 b_1 \dots b_7$, где $b_i = 0$ или 1 в зависимости от того, свободен блок i или используется соответственно. Операция освобождения памяти может быть выполнена с помощью операции логического „И“ над строкой B и соответствующей битовой маской. Например, освобождение блоков 4, 5 и 6 выполняется с помощью предложения:

$B := B \wedge \text{„}11110001\text{“};$

Поиск по методу „первый подходящий“ k непрерывных блоков, $1 \leq k \leq 8$, вызывает ряд сдвигов и логических операций, определяемых следующим образом:

```

Mask := „ $0^k 1^{8-k}$ “; Blockstart := -1;
(0k означает строку из  $k$  нулей)
for i := 0 step 1 until 8 - k do
  if Mask  $\wedge$  B = B then begin Blockstart := i go to L end
  else Mask := „10000000“  $\vee$  Rightlogicalshift(Mask, 1);
L: ...

```

Некоторый анализ фрагментации и использования памяти

При применении приведенных выше стратегий размещения и освобождения можно получить некоторые оценки фрагментации и степени использования основной памяти. Мы полагаем, что запросы и освобождения равновероятны и что система достигла состояния равновесия, где среднее изменение числа „дыр“ в единицу времени равно нулю. Пусть n и m есть среднее число „дыр“ и заполненных блоков соответственно при равновесии. Рассмотрим снова рис. 5.10. Блоками A на рисунках (a), (b), (c) и (d) представлены все возможности расположения заполненных блоков. Пусть a, b, c, d есть среднее число заполненных блоков каждого типа в памяти при равновесии. Игнорируя граничные условия на обоих концах памяти, мы имеем

$$(1) \quad m = a + b + c + d \\ n = \frac{2d + b + c}{2}$$

(2) Так как $b = c$, то $n = d + b$.

Пусть q — вероятность нахождения „дыры“, точно соответствующей запросу, и $p = 1 - q$. На рис. 5.10 в результате

освобождения число „дыр“ увеличивается на 1 в (a), уменьшается на 1 в (d) и остается постоянным в (b) и (c). Вероятность того, что n увеличивается на 1 при любом изменении в распределении памяти, равна

$$(3) \quad \text{Prob}(\text{release}) \times \frac{a}{m},$$

Вероятность того, что n уменьшается на 1, равна

$$(4) \quad \text{Prob}(\text{release}) \times \frac{d}{m} + \text{Prob}(\text{request}) \times (1 - p),$$

где $\text{Prob}(\text{release})$ и $\text{Prob}(\text{request})$ — это взаимосвязанные вероятности запросов и освобождений памяти.

Условие равновесия подразумевает, что (3) = (4). Так как $\text{Prob}(\text{request}) = \text{Prob}(\text{release})$,

$$\frac{a}{m} = \frac{d}{m} + (1 - p),$$

т. е.

$$(5) \quad a = d + (1 - p)m.$$

Подставляя (2) и (5) в (1), мы имеем

$$\begin{aligned} m &= d + (1 - p)m + b + c + d = (1 - p)m + 2b + 2d = \\ &= (1 - p)m + 2n, \end{aligned}$$

что дает конечный результат

$$n = \frac{1}{2} pm.$$

Эти оценки фрагментации памяти были первоначально получены Кнутом (Кнут, 1968), который назвал его правилом *пятидесяти процентов*, так как n стремится к 50% от m , когда p приближается к 1.

Деннинг (1970) получил следующий результат по использованию памяти: пусть средний размер „дыры“ есть $h = kb$, где b — средний размер заполненного блока и $k > 0$, и пусть p — вероятность, использованная в правиле 50%, которая равна 1.

Тогда доля памяти f , занятой „дырами“ в состоянии равновесия, есть

$$f = \frac{k}{k+2}.$$

Это легко может быть получено при применении правила 50% в случае $p = 1$. Пусть M — размер памяти. Тогда

$$h = kb = \frac{M - mb}{m/2}.$$

Это приводит к

$$M = mb \left(\frac{k}{2} + 1 \right).$$

Следовательно,

$$f = \frac{\frac{m}{2} kb}{M} = \frac{\frac{m}{2} kb}{mb(k/2+1)} = \frac{\frac{k}{2}}{\frac{k}{2} + 1} = \frac{k}{k+2}.$$

Модели Кнута (Кнут, 1968) показывают, что необходим средний размер запаса $b \leq M/10$, чтобы получить долю „дыр“ $f \approx 0,1$ ¹⁾; в этом случае размер „дыры“ приблизительно равен $0,22b$. Когда b больше, скажем около $M/3$, f увеличивается до 0,5 и k тогда равно 2. Очевидные выводы заключаются в том, что в динамической среде размер M должен быть большим относительно b , иначе останется ненспользованной еще много памяти, и что некоторая доля памяти f всегда будет теряться („дыры“).

Когда запрос не может быть удовлетворен, уплотнение памяти может рассматриваться как средство получения достаточно большой „дыры“. Однако часто бывает так, что в данный момент память используется интенсивно и результаты не оправдывают затрат. Альтернативным решением является выталкивание (сплонг) достаточного числа заполненных блоков, чтобы создать „дыру“ требуемого размера. В случае сплонга также применимы стратегии замещения, рассмотренные для систем со стационарной организацией в следующем разделе.

Упражнение

Напишите процедуры для запроса и освобождения памяти, использующие стратегию размещения «первый подходящий» и структуру данных «этикетка границ» для доступного пространства. Заголовками процедур являются:

- (1) **integer procedure Request(size);**
comment Вернуть адрес блока, если запрос может быть удовлетворен. В противном случае вернуть 0. Параметр *size* — это число запрошенных слов;
- (2) **procedure Release(addr);**
comment Добавить заполненный блок, начиная с адреса *addr*, к доступному пространству;

Пусть $M[a]$ есть содержимое ячейки памяти a . Заполненный блок, начинающийся с адреса a , будет иметь следующие заголовки границ:

$$M[a] = \text{false}; M[a+1] = s; M[a+s] = \text{false}; M[a+s+1] = s;$$

¹⁾ Эксперимент моделирования завершался при «переполнении памяти», т. е. по запросу, который не мог быть удовлетворен.

Пустой блок, начинающийся с адреса a , будет иметь в заголовке:

$$\begin{aligned} M[a] &= \text{true}; M[a + 1] = s; \\ M[a + 2] &= \text{адрес следующей „дыры“}; M[a + 3] = \text{адрес предыдущей „дыры“}; \\ M[a + s] &= \text{false}; M[a + s + 1] = s; \end{aligned}$$

Список «дыр» имеет заголовок:

$$\begin{aligned} M[ih] &= \text{указатель на первую „дыру“ в списке}; \\ M[lh] &= \text{указатель на последнюю „дыру“ в списке}. \end{aligned}$$

5.4.2. Распределение в страничных системах

5.4.2.1 Статическое и динамическое распределение

Аппаратура страничной организации может оказаться полезной как в системах, которые распределяют память для каждого пользователя статически, так и в системах, делающих тоже динамически. В первом случае страничная организация допускает назначение несплошных блоков памяти фиксированного размера единому пространству имени; нет необходимости в каких-либо стратегиях размещения, таких, как „первый подходящий“ или „наиболее подходящий“, поскольку данная страница может находиться в любом блоке. Управление доступной памятью также значительно упрощается; неупорядоченный список свободных блоков является достаточной базой данных для запросов и освобождений памяти. Следует заметить, что статическое распределение предполагает, что пространство имен пользователя меньше или равно по размеру реальной основной памяти или что используются определенные пользователем перекрытия в виртуальной памяти.

Аргументами в пользу динамического распределения, например такого, как страничной организации по требованию, является экономия основной памяти, уменьшение времени ввода-вывода и взаимовлияния по памяти между процессорами ввода-вывода и центральным процессором, а также увеличение числа активных процессов в основной памяти. ЭВМ RCA Spectra 70/46 представляет собой конкретный пример реализации второго преимущества (Оппенгеймер и Вайзер, 1968): время передачи при вводе-выводе одной страницы (4К байтов) между вспомогательной памятью (барабаном) и ферритовой памятью составляет приблизительно 12,5 мс. Степень взаимовлияния по памяти составляет 1,44 мкс/2 байта, т. е. для каждого двух байтов, передаваемых через канал, основная ферритовая память недоступна ЦОУ в течение 1,44 мкс. Таким образом, для каждой читаемой или записываемой страницы канал ввода-вывода отнимает у ЦОУ около 3 мс. Если посмотреть на эти цифры с других позиций, то это означает, что во время ввода-вывода страницы имеется налицо деградация активности ЦОУ на 24%.

(3/12,5); если к странице не обращаются, то это абсолютная потеря.

При реализации виртуальной памяти сумма размеров пространства имен для активных процессов в общем случае много больше, чем доступная основная память, и она часто распределяется полностью. Следовательно, большинство запросов на память не может быть удовлетворено немедленно. Часто возникает необходимость выбора: или замещать некоторую страницу (страницы), чтобы сделать доступным пространство для запрашиваемого блока (дисциплина перехвата ресурса), или задерживать запрашивающий процесс до тех пор, пока пространство не станет доступным „естественными“ средствами. Чаще используется замещение, так как к значительной части распределенной памяти повторных обращений или совсем не будет или не будет в ближайшем будущем, а страничная организация в принципе реализует эффективные замещения. Выбор конкретной страницы для замещения был предметом многих современных исследований. Результаты также могут быть применены к решениям о замещении в чисто сегментных системах и в системах с высокоскоростными буферными ЗУ (разд. 5.6). Мы также увидим, что при страничной организации по требованию стратегии вызова и размещения определяются *a priori*, оставляя схему замещения как единственную „переменную“, посредством которой можно оптимизировать поведение системы по отношению к дисциплинам управления памятью¹⁾). В оставшейся части этого раздела представлена полезная модель для изучения дисциплин распределения со страничными системами и замещением.

Рассмотрим всю трассу Z выполнения конкретной программы с конкретным набором входных данных. Z задается в форме

$$r_0 \ r_1 \ r_2 \dots r_k \dots r_T,$$

где r_k есть страница команд или данных, к которой в трассе Z зафиксировано обращение при k доступе к памяти, а r_T представляет последнее такое обращение; этот тип трассы назван *цепочкой обращений программы*. Индексы в цепочке обращений могут рассматриваться как моменты времени. Предположим, что программа и данные, соответствующие Z , размещаются в области основной памяти размером m блоков, где $1 \leq m \leq n$, а n — число различных страниц в Z . Дисциплина распределения P определяет для каждого момента времени k множество страниц p_k , которые загружаются в память в момент k , и множество страниц q_k , которые удаляются из памяти или замещаются в момент k при обработке Z . Общепринятым

¹⁾ Мы предполагаем, что размер страницы и объем основной памяти, распределенной программе, фиксированы.

критерием качества для дисциплины P является общее число страниц $n_p(Z)$, загружаемых в память, когда Z обрабатываются, а именно

$$n_p(Z) = \sum_{k=0}^T |p_k|.$$

Пример

Пусть $Z = wzxyyxwxxxxz$ и $m = 2$. Число различающихся страниц $n = 4$, так как в Z зафиксированы обращения к страницам $\{w, x, y, z\}$.

а) Рассмотрим дисциплину P , которая основана на том, что память содержит только один из наборов $\{w, x\}$, $\{x, y\}$ или $\{y, z\}$. Тогда план памяти мог бы изменяться по мере того, как следуют зафиксированные обращения в Z следующим образом:

Z Время k	w_0	z_1	x_2	y_3	y_4	x_5	w_6	x_7	x_8	x_9	z_{10}
p_k	$\{w, x\}$	$\{y, z\}$	$\{x\}$	—	—	—	$\{w\}$	—	—	—	$\{y, z\}$
q_k	—	$\{w, x\}$	$\{z\}$	—	—	—	$\{y\}$	—	—	—	$\{w, x\}$
Блок 1	w	y	y	y	y	y	w	w	w	w	y
Блок 2	x	z	x	x	x	x	x	x	x	x	z

$$n_p(Z) = 8.$$

(б) Пусть P есть дисциплина страничной организации по требованию, в которой страница не загружается до тех пор, пока к ней нет обращения. Если страница должна быть замещена, чтобы обеспечить свободное пространство для „затребованной“ страницы, то выбор удаляемой страницы будет производиться по принципу FIFO. Тогда план памяти изменяется таким образом:

Z Время k	w_0	z_1	x_2	y_3	y_4	x_5	w_6	x_7	x_8	x_9	z_{10}
p_k	$\{w\}$	$\{z\}$	$\{x\}$	$\{y\}$	—	—	$\{w\}$	$\{x\}$	—	—	$\{z\}$
q_k	—	—	$\{w\}$	$\{z\}$	—	—	$\{x\}$	$\{y\}$	—	—	$\{w\}$
Блок 1	w	w	x	x	x	x	w	w	w	w	z
Блок 2	—	z	z	y	y	y	y	x	x	x	x

$$n_p(Z) = 7.$$

Мэттсон и др. (1970) показали, что для любой дисциплины P и вида распределения страниц Z существует дисциплина страницной организации по требованию P_d , такая, что

$$n_{P_d}(Z) \leq n_P(Z),$$

т. е. соответствующим образом выбранная дисциплина страницной организации по требованию будет давать минимальное $n_p(Z)$. Однако необходимо быть очень осторожным при интерпретации результатов такого анализа, так как эта модель и исходные предположения могут быть недопустимыми для некоторых систем и критерий оптимальности, $n_p(Z)$, может быть не подходящим. В частности, результат, относящийся к страницной организации по требованию, не „доказывает“, что это наилучшая для использования дисциплина; он говорит только, что она приводит к наименьшему количеству загружаемых страниц. Предположим, например, что „стоимость“ $h(k)$ загрузки k страниц в какое-то время была равна $C_1 + C_2k$, где C_1 и C_2 — константы. Это не является вымышленной стоимостью за передачу из вспомогательной памяти в исполнительную; в C_1 включается стоимость обслуживания и ожидания, и C_2k предполагает, что стоимость передачи прямо пропорциональна числу страниц. Тогда стоимость загрузки для дисциплины (а) в приведенном выше примере есть $5C_1 + 8C_2$ и для (б) — $7C_1 + 7C_2$; наилучшая возможная дисциплина страницной организации по требованию для (б) (см. следующий раздел) приведет к $n_p(z) = 6$ с соответствующей стоимостью $6C_1 + 6C_2$. Для всех $C_1 > 2C_2$ имеем $5C_1 + 8C_2 < 6C_1 + 6C_2$, и страницная организация по требованию будет „стоить“ больше.

При реализации большой виртуальной памяти на страницной машине с мультипрограммированием реальная рабочая память может рассматриваться как общий „пул“, с помощью которого удовлетворяются запросы на память и освобождение памяти; с другой стороны, каждой активной программе может быть выделено свое собственное рабочее пространство, внутри которого производится динамическое управление памятью. Замещения в последнем случае производятся только из собственного рабочего пространства процесса, в то время как в первом случае отсутствие страниц для одного процесса может вызвать замещения страниц полностью независимых от него программ. Приведенная выше модель предполагает, что рабочее пространство имеет фиксированный размер (t блоков) и связано с развивающимся процессом. Оба подхода применялись как успешно, так и неудачно (Денинг, 1970; Оппенгеймер и Вайзер, 1968).

5.4.2.2 Схемы замещения

Чтобы провести последующие сравнения и дать представление о практических методах, мы сначала опишем оптимальную, но нереализуемую схему замещения. Предполагаются модель и критерий качества, рассмотренные в предыдущем разделе. Иначе говоря, цепочки обращений программы известны *aприори* и требуется минимизировать $n_p(Z)$ для каждой цепочки Z . Так как существует дисциплина страничной организации по требованию с минимальным $n_p(Z)$, то будет рассматриваться только этот тип дисциплины. Тогда такие величины, как минимальное значение $n_p(Z)$, число замещений и число отсутствий страниц, будут эквивалентны. Можно доказать, что следующая стратегия замещения является оптимальной (Биледи, 1966; Мэттсон и др., 1970; Ахо и др., 1971).

Выберем для замещения ту страницу, к которой дальше всего не будет обращений.

Если замещение, вызванное требованием отсутствующей страницы, происходит во время k , выберем страницу r , такую, что

(1) $r \neq r_i$ для $k < i \leq T$ (к странице r больше не обращаются)

или если такого r не существует, то

(2) $r = r_t$ для $k < t \leq T$,

$r \neq r_{t'}$ для $k < t' < t$ и $t - k$ — максимально.

Пример

Оптимальная стратегия замещения дает $n_p(Z) = 6$ для примера в последнем разделе.

Конечно, цепочки обращений программы никогда не известны заранее. Практические схемы замещения стремятся предсказать будущие обращения в предположении, что прошлая история обращений будет повторяться в ближайшем будущем.

Первая схема, которую можно рассмотреть — стратегия *случайной* выборки; она выбирает страницу, которую надо заменить, случайным образом, используя генератор случайных чисел. Эта схема может быть полезна, если вышеупомянутое предположение не имеет силы (по крайней мере статистически) или как средство сравнения с другими методами. Вероятно, простейшими алгоритмами замещения являются алгоритмы, основанные на принципе FIFO. Если память распределяется в рабочем пространстве с числом блоков m , то упорядоченный список страниц $P[0], P[1], \dots, P[m-1]$ и указатель k поддержи-

ваются такими, что $P[k]$ определяет самую последнюю загруженную страницу, а $P[k + mi]$ указывает на i -ю страницу, загруженнную в хронологическом порядке ($i = 1, 2, \dots, m - 1$). Тогда при отсутствии страницы (при попытке обратиться к странице, которая не присутствует в основной памяти) выбирается для замещения страница $P[k + mi]$, а k увеличивается на 1 (по модулю m); номер загруженной страницы последовательно присваивается $P[k]$ (новое k). Этот метод предполагает, что страницы, дольше всего находящиеся в памяти, будут с наименьшей вероятностью работать в будущем. Третьей основной стратегией (и самой широко распространенной) является дисциплина LRU¹). Здесь замещается страница, которая не использовалась в течение относительно длительного периода времени в прошлом. Доводом в пользу такого решения является то, что страницы, к которым не обращались относительно долго, вероятно, не потребуются в ближайшем будущем. Стратегия LRU требует непрерывного контроля за всеми обращениями к странице, чтобы накопить историю использования, в то время как при использовании принципа FIFO вспомогательные учетные действия необходимы только во время отсутствия страницы.

Замещаемая страница должна быть записана во вспомогательную память, если ее содержимое изменилось в период ее последнего нахождения в основной памяти; в противном случае можно просто переписывать нужную страницу на место замещаемой страницы. Чтобы следить за изменением содержимого страницы, обычно используется специальный бит, скажем w , связанный с каждой записью в таблице страниц или с каждым блоком. После загрузки новой страницы соответствующий бит w устанавливается в состояние *OFF*; w устанавливается в *ON* только тогда, когда информация в странице модифицируется (при операции записи). При общепринятой реализации дисциплины LRU с каждой страницей или блоком в памяти также связывают бит использования, который обозначим u . При любом обращении к странице бит u устанавливается в состояние *ON*; все биты u могут сбрасываться (переключаться в *OFF*) или периодически, или когда устанавливается последний бит u , или во время замещения, если все биты u находятся в состоянии *ON*. Тогда для целей замещения множество страниц в памяти делится на 4 подмножества. Это совокупности всех страниц со значениями битов:

1. $(u, w) = (\text{OFF}, \text{OFF})$
2. $(u, w) = (\text{OFF}, \text{ON})$

¹⁾ Сокращение от „Least recently used”, т. е. стратегия замещения по давности использования. — Прим. ред.

3. $(u, w) = (ON, OFF)$
4. $(u, w) = (ON, ON)$.

Блок для замещения выбирается случайно из самого первого непустого подмножества, что приближает этот метод к LRU.

Существует несколько более усложненных методов для ведения записей об истории использования. В одной из предлагаемых схем для каждого блока применяется конденсатор. При обращении конденсатор заряжается; последующий экспоненциальный спад заряда позволяет непосредственно вычислить временной интервал. При другом привлекательном способе для каждого блока используется регистр старения из n разрядов;

$$R = R_{n-1}R_{n-2} \dots R_1R_0.$$

При обращении к странице разряд R_{n-1} устанавливается в 1 и через каждые τ единиц времени регистр R сдвигается вправо на один разряд. Если к странице обращались в последние приблизительно $n\tau$ единиц времени, то

$$\bigvee_{i=0}^{n-1} R_i = 1.$$

По алгоритму LRU тогда выбирают подмножество страниц с минимальным значением R , рассматривая содержимое R как положительное целое. Начальная модель ATLAS содержала более сложный механизм для хранения истории использования, которая включала информацию об использовании блоков во вспомогательной памяти; решение о замещении было основано как на учете времени, прошедшего с момента последнего обращения, так и на учете времени предыдущего периода неактивности на барабане.

Биледи (1966) испытал ряд алгоритмов замещения на множестве программ, написанных на ФОРТРАНе и языке ассемблера, и сравнил результаты с результатами, использующими оптимальное, но нереализуемое замещение. В общем случае основные схемы по методу LRU требовали примерно в два раза больше загрузок страниц по сравнению с оптимальным и были лучшими по сравнению с алгоритмами замещения в ATLAS, FIFO и основанными на случайной выборке. Стратегии FIFO и стратегия случайного выбора были наименее удовлетворительными, требуя приблизительно в три раза больше загрузок страниц по сравнению с оптимальным. Размер блока и размер памяти, т. е. два других параметра, варьируемые в экспериментах Биледи, влияли на результат намного более значительно (мы проработаем эти вопросы в следующем разделе).

В приведенном выше рассмотрении в общем предполагалось, что каждый процесс размещен в области фиксированного раз-

мера, из которой производится замещение страниц. Как эта дисциплина, так и другая дисциплина, при которой вся память объединяется в пул, может привести к недопустимому движению страниц из-за динамически меняющихся требований к памяти со стороны различных процессов и к возможной перегрузке системы избытком активных процессов. Модель *рабочего множества* Деннинга (1968) — это дисциплина управления памятью, которая в принципе устраивает эти проблемы. Основополагающая концепция этой дисциплины заключается в том, что процесс должен иметь „приемлемый“ объем памяти для своей программы и данных, прежде чем он будет активизирован, и что требуемый объем определяется динамически по его недавнему поведению в прошлом. Каждый процесс в данный момент t имеет рабочее множество страниц $W(t, \tau)$, определяемое как множество страниц, к которым обращается процесс в течение временного интервала $(t - \tau, t)$; $w(t, \tau)$ есть число страниц в W (мощность W). Предполагается, что и W и w приблизительно постоянны на малых временных интервалах. Память управляет в соответствии со следующими правилами (Деннинг, 1970):

Программа может работать тогда и только тогда, когда ее рабочее множество находится в памяти; страница не может быть удалена, если она является членом рабочего множества выполняющейся программы.

Таким образом, данная схема замещения есть вариант метода LRU.

Стратегия рабочего множества кажется очень привлекательной, но на сегодня накоплено мало опыта по ее практическому использованию; например, одним из вопросов, который еще не решен, является вопрос, как выбрать значение параметра τ . В одной из публикаций о результатах моделирования планирования и смены страниц для системы разделения времени RCA Spectra 70/46 показано, что одна из форм стратегий рабочего множества дает наилучший результат (Оппенгеймер и Вайзер, 1968). В данном случае не требовалось наличия W ; было необходимо только, чтобы w блоков памяти были распределены процессу прежде, чем он был активизирован.

Упражнения

1. Получите картину развития плана памяти для цепочки обращений программы и значения t , взятых из примера в разд. 5.4.2.1, используя
 - (a) оптимальное размещение,
 - (b) LRU.
2. Порекомендуйте структуры данных и алгоритмы для

- (а) поддержания рабочего множества процесса,
- (б) выборки замещаемой страницы, используя принципы рабочего множества.

5.5. Оценка страничной организации

Для изучения динамического поведения программ при страничной организации был проведен ряд экспериментов (Биледи, 1966; Файн и др., 1966; Коффман и Вариан, 1968; Фрейбергс, 1968; Баэр и Сейгер, 1972). Общий метод заключался в выполнении в режиме интерпретации выборочного набора программ, моделируя механизм страничной организации; из наиболее важных параметров варьировались размер страницы и объем рабочей памяти. Результаты этих независимых исследований были сопоставлены и могут быть обобщены следующим образом:

1. Большинство процессов требует значительную долю своих страниц в течение очень короткого периода времени после активизации; например, при типичной работе в режиме разделения времени в течение одного кванта в среднем были обращения приблизительно к 50% страниц процесса.

2. В пределах страницы выполняется относительно небольшое число команд прежде, чем управление передается другой странице; для размеров страницы в 1024 слова (самого общепотребительного размера) в большинстве исследованных случаев до обращения к следующей странице было выполнено менее 200 команд.

3. В диапазоне 64 — 1024 слов на страницу при постоянном размере доступной памяти число отказов при обращении к странице увеличивается по мере увеличения размера страницы (страничная организация по требованию).

4. Если объем памяти уменьшается при фиксированном размере страницы, то существует точка, начиная с которой страничные отказы растут экспоненциально (страничная организация по требованию).

На рис. 5.13 качественно изображено поведение программ при страничной организации.

Одним из выводов из сказанного выше является то, что размеры страниц должны быть относительно малыми — конечно меньше, чем общепотребительный размер, равный 1024 слова. Некоторые факторы, которые мы рассмотрим, объясняют причину использования страниц большого размера, в то время как другие говорят в пользу первого решения.

Среди многих причин, приводящих к использованию страничной организации, было уменьшение фрагментации памяти. Несмотря на то что шахматная организация памяти устранена,

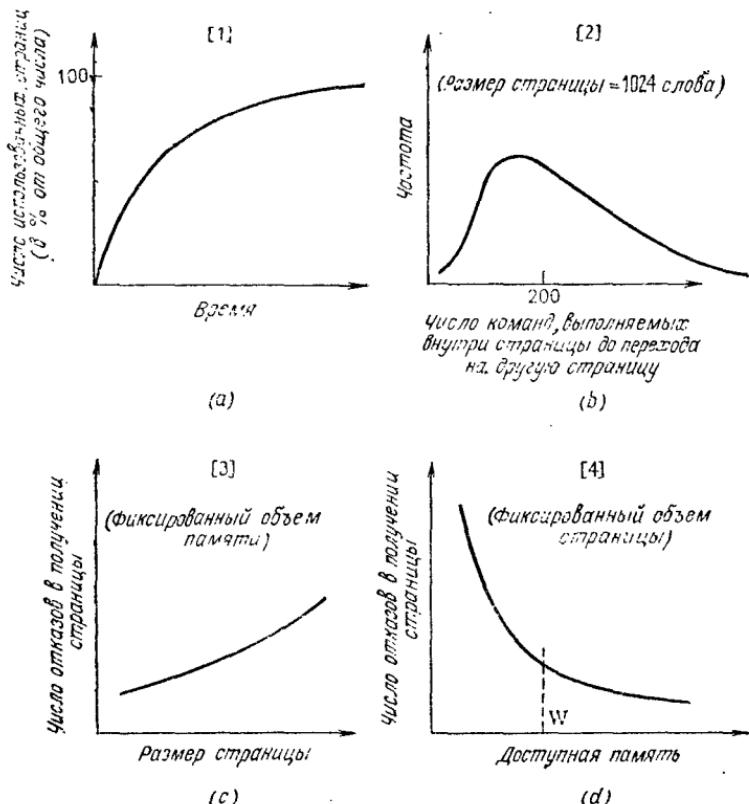


Рис. 5.13. Качественное поведение программ при страницной организации.

в страницных системах имеют место два других типа потерь памяти. Потери первого типа, изученные Рэнделлом (1969), происходят потому, что размеры сегмента редко (если вообще когда-нибудь) бывают точно кратны размеру страницы; в последней странице сегмента тогда неизменно будет потерявшая память. Рэнделл назвал эту потерю *внутренней фрагментацией* и провел несколько экспериментов, которые показали, что потери при этом могли превышать потери, обусловленные более общей (внешней) фрагментацией в нестраницных системах. Потери второго типа — это таблицы, требующиеся для хранения функций отображения виртуального пространства в реальное, т. е. память для таблиц страниц. Если предполагается, что средний размер сегмента s_0 много больше, чем размер страницы s , и что таблицы страниц хранятся целиком в основной памяти, то может быть выведен оптимальный размер страницы (Денинг, 1970; Уолман, 1965).

Теорема

Пусть c_1 — стоимость потери слова основной памяти из-за использования таблицы страниц и c_2 — стоимость потери слова при внутренней фрагментации. Если $z \ll s_0$ и каждый сегмент начинается на границе страницы, то оптимальный размер страницы z_0 приблизительно равен $\sqrt{2cs_0}$, где $c = c_1/c_2$.

Доказательство (неформальное). Для каждого сегмента стоимость потерь памяти C_z при страницочной организации с размером страницы z есть

$$C_z = C_t + C_i,$$

где C_t — стоимость внутренней фрагментации и C_i — стоимость таблицы страниц. Если $z \ll s_0$, то приблизительно $C_i = c_2 z/2$, так как в среднем половина последней страницы в сегменте теряется.

Значение C_t приблизительно определяется равенством $C_t = c_1 s_0/z$, так как каждый сегмент требует s_0/z слов таблицы страниц. Следовательно, $C_z = c_2 z/2 + c_1 s_0/z$.

Чтобы найти оптимум z_0 , решим уравнение $dC_z/dz = 0$, т. е. $c_2/2 - c_1 s_0/z_0^2 = 0$, что приводит к нашему результату.

Этот результат подтверждает наш вывод о малых размерах страниц, если мы примем $s_0 \leq 5000$ и $c = 1$; в этом случае $z_0 \leq 100$ слов. Размер z_0 даже меньше, если, как измерено на B5500 (Бэтсон и др., 1970), типичные размеры сегмента в 100 слов (!) рассматриваются как универсальные. Однако меньшие размеры страниц требуют больших таблиц страниц, чтобы адресоваться к фиксированному объему виртуальной памяти, а также предполагают, что основная память должна состоять из большого числа блоков.

На практике применяются большие размеры страниц из-за

1. Программных накладных расходов при обработке таблиц страниц.

2. Стоимости аппаратуры разделения памяти на блоки фиксированных размеров.

3. Физических характеристик устройств вспомогательной памяти с врачающимися носителями информации, которые работают более эффективно при передаче блоков информации больших размеров (гл. 9 содержит обзор аппаратуры вспомогательной памяти).

Как правило, дорожка на барабане или диске будет хранить несколько тысяч слов, т. е. несколько страниц; прежде чем первое слово желаемой страницы появляется под механизмом чтения-записи, в среднем возникает заметная задержка. Стра-

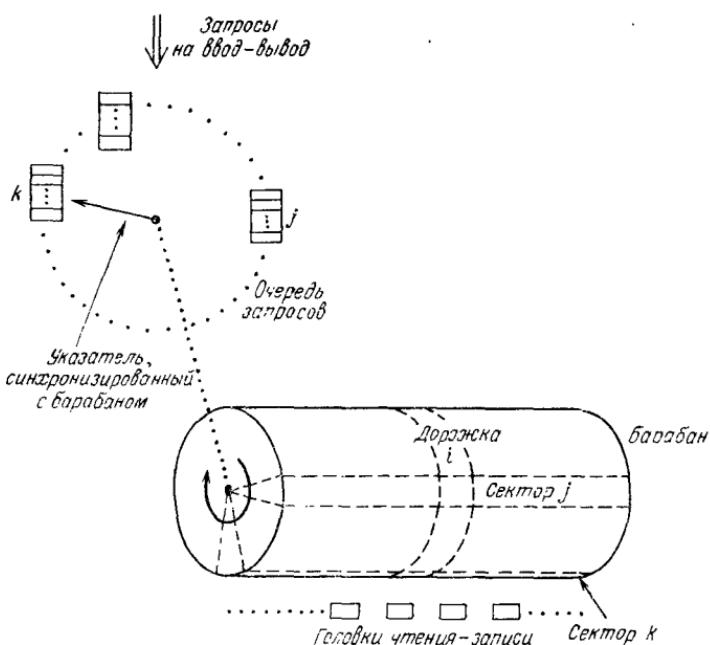


Рис. 5.14. Страницный барабан.

ничный барабан или диск (Деннинг, 1970, 1972) обеспечивает эффективный способ устранить, когда возможно, эту задержку. Каждая дорожка барабана i разделена на ряд секторов фиксированного размера, причем каждый сектор соответствует странице. С вращающимся барабаном синхронизирован аппаратуры секторный указатель, и для каждого сектора устанавливается очередь запросов на ввод-вывод; каждый элемент очереди содержит номер дорожки и индикатор чтения или записи (рис. 5.14). Когда указатель достигает сектора с непустой очередью, в канале барабана автоматически инициируется операция чтения или записи по заявке, находящейся в начале очереди. Работа этого барабана теоретически во многом превосходит работу барабана, который планирует каждую операцию на основе дисциплины FIFO. Мультипрограммная система Мичиганского университета (MTS, 1967), работающая на IBM 360/67, является одним из примеров системы, которая использует такой страницный барабан.

Вторым выводом из экспериментов по динамическому поведению программ является заключение, что работа по вводу-выводу, являющаяся следствием отсутствия страниц в основной памяти, может стать недопустимо интенсивной и непродуктивной, если каждый активный процесс не имеет достаточного

объема памяти, например рабочего множества, доступного для нужд программ и данных. Под „недопустимо высокой“ и „не-производительной“ работой мы подразумеваем, что ЦОУ уделяет слишком много времени на организацию выталкивания и ввод страниц, что ЦОУ и каналы работают последовательно, а не параллельно, так как все активные процессы постоянно нуждаются в пространстве для страниц и что одни и те же страницы непрерывно выталкиваются и вводятся; другими словами, существует катастрофическое сокращение выполняемой полезной работы. Точка, помеченная \dot{W} на рис. 5.13 [d], представляет минимальный объем памяти, требуемый для устранения этих эффектов. Большинство больших систем разделения времени общего назначения, использующих страничную организацию по требованию (например, MULTICS, 1965, MTS, 1967; TSS, 1967; CP-67/CMS, 1969), стремятся управлять количеством вводимых и выталкиваемых страниц или путем ограничения числа активных процессов, которые могут конкурировать за использование ЦОУ, или применяя некоторый вариант стратегии рабочего множества, или, более часто, откладывая запросы на страницы, если в это время существует слишком много неудовлетворенных запросов.

Несмотря на сомнения, которые возникли на начальном этапе, и значительные затраты усилий, динамическое распределение памяти, основанное на страничной организации по требованию, успешно используется во многих системах разделения времени. Однако, несмотря на элегантность этой концепции, еще не очевидно, что более простые стратегии, например использующие свопинг и рабочие пространства фиксированного или переменного размера, не могут иметь превосходства при долгой работе или может быть более полезным распределение на сегментной основе, как реализовано на старших моделях ЭВМ Burroughs. Чтобы судить об этом, в настоящее время имеется слишком мало данных для сравнения.

5.6. Иерархии памяти¹⁾

Ограничение производительности ЭВМ часто является следствием:

1. Малого объема основной памяти относительно требований программы.
2. Низкой скорости доступа к основной памяти относительно скорости регистров ЦОУ.

¹⁾ Терминология и определения в этом разделе взяты из обзорной статьи Денинга (1970).

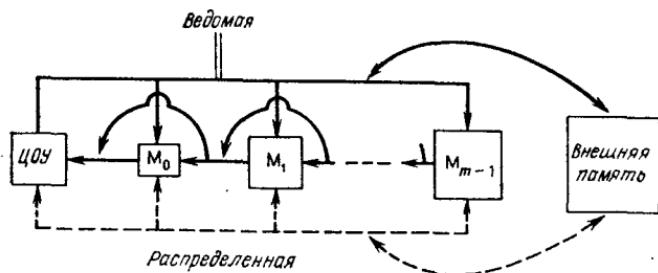


Рис. 5.15. Иерархии памяти.

3. Различия порядков времени доступа к вспомогательной памяти (механика) и времени доступа к основной памяти (электроника).

Наши предыдущие рассуждения неявно предполагали, что основная память состояла из множества однородных адресуемых ячеек с одинаковыми временами доступа. В этих условиях страничная организация вводилась, чтобы улучшить использование памяти и эффективность обработки. В настоящее время получил широкое распространение иной подход, который основан на использовании иерархий памяти.

Основная память больше не однородна, а состоит из иерархии устройств памяти M_0, M_1, \dots, M_{m-1} , адресуемых с помощью электронной аппаратуры, причем каждое имеет свои скорости и размеры (рис. 5.15). Характеристики каждого уровня таковы, что

$$\text{Время доступа}(M_{t-1}) < \text{Время доступа}(M_t)$$

и

$$\text{Размер}(M_{t-1}) < \text{Размер}(M_t), \quad t = 1, \dots, m - 1.$$

Аппаратура и программное обеспечение спроектированы так, чтобы хранить наиболее часто используемую информацию в M_0 с целью приближения к производительности системы с однородной памятью того же типа, что и M_0 , но объемом, равным

$$\sum_{i=0}^{m-1} \text{Размер}(M_i).$$

Существуют две базовые схемы соединения ЦОУ и вспомогательной памяти в иерархию, называемые *подчиненной* и *распределенной* организациями. При методе „распределения“ вся основная память непосредственно доступна как для вспомогательной памяти (через канал), так и для ЦОУ для чтения и записи. В подходе „подчинения“, впервые предложенном Уилксом (1965), предполагается, что M_0 является (почти) устройством,

к которому обращается только ЦОУ, тогда как связи с вспомогательной памятью в общем случае ограничены более медленными уровнями памяти M_{m-1}, M_{m-2}, \dots . Когда данные из ЦОУ записываются на любой уровень памяти M_i , аппаратура обеспечивает появление новой копии в M_{i+1}, \dots, M_{m-1} с помощью передачи данных через иерархию. Напротив, когда данные выбираются с любого уровня M_i , они также запоминаются в M_{i-1}, \dots, M_0 для использования в будущем. Это предполагает использование некоторой дисциплины замещения, если любой из упомянутых уровней памяти заполнен; следует заметить, что такое замещение на любом уровне $M_i, i \neq m-1$, состоит только из перекрытия замещаемой области памяти, так как новые копии существуют в M_{i+1}, \dots, M_{m-1} . Используется также комбинация организаций подчинения и распределения; например, процессоры ЭВМ часто будут иметь скрытую высокоскоростную память для просмотра команд вперед и небольших программных циклов, которая, по существу, действует как подчиненная память для некоторой видимой однородной, подчиненной или распределенной памяти.

Пример

Одной интересной и успешной реализацией подчиненной памяти является реализация в IBM 360/85 (Конти и др., 1968; Липтей, 1968). В этой системе имеется два уровня в иерархии ($m = 2$). Уровень M_0 , называемый кэш-памятью, или высокоскоростным буфером, является невидимой (для программы) 16К-байтовой памятью с временем выборки 80 нс, разделенной на 16 секторов по 1К. Основная память M_1 — стандартная память с временем выборки в микросекундном диапазоне аналогично разделена на секторы размером 1К. Поле, связанное с каждым сектором s в M_0 , используется для хранения адреса сектора, выделенного для s в M_1 . Информация передается между M_0 и M_1 элементами по 64 байта, называемыми блоками; каждый сектор, таким образом, разделен на 16 блоков. Каждому блоку в кэш-памяти соответствует однобитовая этикетка t ; значение $t = 1$ показывает, что блок содержит информацию, в то время как $t = 0$ соответствует неопределенному содержимому. Распределение M_0 выполняется на базе единственного сектора, а каждая передача информации включает только 1 блок. Операции записи и выборки между ЦОУ и основной памятью выполняются следующим образом:

1. Записать данные α в X (X есть адрес памяти M_1).
 - a) Если X находится в M_0 , то запомнить α как в M_0 , так и в M_1 .

- (b) Иначе, записать α в M_1 .
2. Выбрать данные α из X .
 - a) Если X находится в M_0 (сектор, находящийся в памяти с адреса X , находится в M_0 и $t = 1$ для блока X), тогда вызвать α из M_0 .
 - (b) Если сектор с адресом X находится в M_0 , но $t = 0$, то вызвать α из M_1 и загрузить блок с X в M_0 (параллельно).
 - c) Если сектор с адресом X не находится в кэше, то
 - i) если кэш заполнен, использовать стратегию LRU, чтобы выбрать сектор для замещения, и
 - ii) вызвать α из M_1 и загрузить одновременно блок с адресом X в M_0 .

Записи во вспомогательную память происходят только из M_1 , а считывания, которые изменяют память, рассматриваются аналогично операциям записи (случай 1 выше). Все динамическое управление памятью выполняется аппаратурой. Эта организация больших секторов и маленьких блоков оказывается очень удачным инженерным компромиссом между меньшей стоимостью аппаратуры для обеспечения больших страниц и экспериментальной и теоретической обоснованностью малых размеров страниц. Ранние модели и реальный опыт показали, что внутренняя производительность ЭВМ составляет приблизительно от 60 до 90 процентов от „идеальной“ конфигурации аппаратуры с памятью того же размера, что и M_1 , имеющей время выборки 80 нс. В качестве побочного полезного эффекта можно отметить, что значительно уменьшается взаимодействие с каналом, так как рабочее множество находится обычно в M_0 , а большинство операций ввода-вывода имеет дело исключительно с M_1 .

Проблемы распределения памяти между основным и вспомогательным запоминающими устройствами (ЗУ) аналогичны неиерархической ситуации. Они, однако, не такие трудные, так как могут быть использованы в системе основной памяти значительно большего объема, но более медленной. Иерархические системы памяти являются также одним из решений инженерной проблемы увеличения объема и скорости ЗУ за пределы, доступные в настоящее время; в настоящее время при стандартной технологии скорость доступа ограничивается не временем переключения компонентов памяти, а временем передачи сигнала к каждому компоненту.

Часто распределенные системы являются более гибкими, так как программные средства распределения полностью управляют памятью. На практике существует обычно два уровня иерархии — стандартная быстрая основная память M_0 и более

медленная, но большая по объему ферритовая память M_1 ; обе они работают в микросекундном диапазоне. Управление M_0 и M_1 намного сложнее, чем одноуровневой памятью, причем критическое значение имеют вопросы, куда загружать информацию (в M_0 или M_1 ?) и когда передавать информацию между тремя уровнями, состоящими из M_0 , M_1 и вспомогательной памяти. В одной из установленной в университете МС общего назначения, где память статически распределяется по заданиям (CLASP, 1971), M_0 содержит пакетные задания, системное ядро и интенсивно используемые системные программы, тогда как M_1 содержит интерактивный текстовый редактор, языковый процессор разделения времени и связанный с ним подсистему, буферы ввода-вывода и остающуюся часть операционной системы; здесь разумно предположить, что пакетные задания и ядро системы потребляют большую часть циклов ЦОУ и должны находиться в более быстрой памяти. Система разделения времени университета Карнеги — Меллона на базе IBM 360/67 (Вареха и др., 1969) представляет собой пример динамически распределенной системы, использующей страничную организацию. При одной дисциплине распределения памяти рабочие множества для диалоговых задач хранятся исключительно в большой бункерной ферритовой памяти (M_2) и там выполняются, тогда как интенсивно используемым компонентам операционной системы выделяется высокоскоростная память (M_0) и в как можно большем объеме. Применяются также более сложные дисциплины, в которых частота доступа пользователя и системных программ динамически определяет их положение в иерархии. Группа университета Карнеги — Меллона продемонстрировала полезность иерархии как экономичного метода, уменьшающего движение страниц и время ответа для интерактивных задач. Однако все еще требуется аналитический и экспериментальный поиск дисциплин распределения для иерархии памяти.

6. Разделение процедур и данных в основной памяти

Мультипрограммная организация обязательно включает разделение разнообразных ресурсов между пользовательскими (и системными) процессами. Разделяются аппаратурные ресурсы (в первую очередь время активного ЦОУ, канала, процессоров и устройств ввода-вывода), а также пространство областей центральной и вспомогательной памяти. Несмотря на то что одним из исходных мотивов развития мультипрограммирования было разделение аппаратуры, оказалось, что основные программные ресурсы могут также быть разделяемыми в пространстве и во времени и что можно получить реальные выгоды от такого разделения.

6.1. Необходимость разделения ресурсов

Разделение программ и данных в ЭВМ часто заключается в предоставлении пользователю собственной частной копии разделяемой информации. Копия может быть вручную включена в задание перед вводом его в ЭВМ (например, в виде колоды карт) или может автоматически добавляться загрузчиком, обращающимся к библиотечному или вспомогательному файлу. Это может также осуществляться в мультипрограммной среде. Однако часто несколько активных процессов нуждается в одной и той же программе или данных, расположенных в одно и то же время в основной памяти. Если бы каждый процесс имел свою собственную копию, системные накладные расходы были бы неизбежны; они состояли бы из расходов на ввод-вывод при загрузке копий и расхода памяти для хранения этих копий. Серьезная проблема возникает, когда обновляются многочисленные копии одного и того же файла данных. Почти всегда необходимо иметь один представительный файл, который содержит *все* обновления. Проблема здесь чем-то напоминает уже знакомую нам проблему критической секции, рассмотренную в гл. 3, и, если работать невнимательно, могут произойти ошибки того же типа. По этим причинам более чем одним процессом часто разделяется единственная копия в основной памяти; разделение происходит путем мультиплексирования времени, когда несколько процессов разделяют единственное ЦОУ, или

на самом деле одновременно, когда процессы развиваются на независимых процессорах, имеющих общую память. В этой главе идет речь о структуре программ (и до некоторой степени данных), которая нужна для правильного и эффективного разделения единственной копии. Сначала мы рассмотрим некоторые типичные случаи применения разделения данных и процедур.

Разделение данных

1. Несколько процессов (возможно вызываемых интерактивными пользователями) одновременно опрашивают файловую систему. Одновременно могут использоваться справочники или индексы файлов.

2. Разделение системными программами базы данных. Производится непрерывный поиск и обновление различными системными процессами информации о состоянии ресурса (например, какие элементы ресурса свободны или заняты). Кроме того, может существовать процесс „показа“, который отображает информацию о состоянии на некоторое выходное устройство.

3. Критические секции прежде всего используются для управления разделением данных последовательным образом. (См. приложения и примеры в гл. 3.)

Разделение процедур

1. *Программы системного ядра.* В МС все процессы разделяют одно и то же множество программ ядра, например программы ввода-вывода.

2. *Утилиты и системные службы.* Широко используемые стандартные программы, такие, как различные программы получения дампов, утилита типа „лента — печать“ или связывающий загрузчик, могут существовать в виде единственной разделяемой копии.

3. *Языковые процессоры и текстовые редакторы.* Широко используемые компиляторы, ассемблеры и интерпретаторы часто содержатся в памяти, выделенной для выполнения, так чтобы их одновременно могли использовать несколько заданий. Текстовые редакторы и интерактивные языковые подсистемы, такие, как системы для APL или BASIC, проектируются во многих случаях так, чтобы единственная копия такой системы могла в одно и то же время обслуживать несколько пользователей, работающих за терминалом.

4. *Прикладные программы.* Некоторые интенсивно используемые прикладные программы могут разделяться, но часто бывает проще загрузить копию для каждого запрашивающего

процесса; в противном случае необходимо просмотреть память, чтобы определить, существует ли пригодная для разделения копия, и вести учет процессов, использующих эту программу. Б разд. 3.1.1 и 3.1.2 по параллельному программированию приведены примеры простых ситуаций, в которых наиболее удобна единственная копия. В этих примерах несколько процессов в пределах *единственного* задания разделяют одну и ту же программу; средства мультизадачности в некоторых языках высокого уровня, например в PL/I, неявно включают разделение единственной копии.

6.2. Условия разделения программ

На заре программирования часто заявляли, что своей производительностью и гибкостью ЭВМ во многом обязана возможность модифицировать программы во время их выполнения; команды рассматривались как данные, и наоборот. Это утверждение было не столько теоретическим, сколько практическим. Структура машин первого поколения была такой, что для создания эффективных и удобных программ модификация команд оказывалась необходимой. Два основных случая требовали самоизменения команд: построение циклов и переход к подпрограммам с возвратом. Например, следующая последовательность (частичная) самоизменяющихся команд суммировала множество чисел в массиве A ; предполагается, что машина одноАдресная и однорегистровая (R).

Инициализировать

<i>LOOP</i>	<i>CLA</i>	$*+4$	Модифицировать „ <i>ADD A</i> “
	<i>ADD</i>	$=1$	чтобы получить новый элемент A .
	<i>STO</i>	$*+2$	
	<i>CLA</i>	A	$R := SUM;$
	<i>ADD</i>	A	$R := R + Ai;$
			Модифицируемая команда
	<i>STO</i>	<i>SUM</i>	$SUM := R;$
			<i>Конец проверки цикла</i>
	<i>TRA</i>	<i>LOOP</i>	
		\vdots	
	<i>SUM</i>	<i>DS</i>	1
	A	<i>DS</i>	100

Позднее использование индексных регистров для запоминания и вычисления адресов сделало самомодификацию команд необязательной.

Модификация команды во время выполнения сегодня не считается хорошим стилем в практике программирования по крайней мере по двум причинам. Логика программы часто становится очень запутанной и сложной для реализации; в результате программы трудно отлаживать и понимать. Неуправляемые самоизменения программы также делают невозможным разделять ее; процесс может быть прерван после того, как он изменил команду (или элемент данных), а другой процесс, разделяющий ту же область программ и данных, может или изменить ту же команду (элемент данных) или выполнить (использовать) ее, неправильно полагая, что никаких изменений не было сделано. По обеим этим причинам (ясность и разделение) желательно устраниć операции запоминания в пределах областей программы. Для этого обрабатываемые данные не должны находиться в области программы.

Программы, инициализирующие сами себя

В ситуациях, когда разделение производится последовательно, можно изменять программу и данные, локальные относительно этой программы, если она сама себя инициализирует, т. е. если начальные значения команд и данных устанавливаются самой программой, а инициализирующая часть программы никогда не изменяется. Следовательно, достаточно единственной копии, если каждый процесс полностью использует эту копию прежде, чем другому процессу будет разрешено к ней обращаться. Программа является, таким образом, критической секцией и выступает в качестве „последовательно используемого ресурса“. Такие программы обычно применяются в МС, которая выделяет один ресурс или блок основной памяти для последовательной пакетной обработки программ на выбранном языке (например, для компиляции, загрузки и запуска); эта языковая подсистема постоянно находится в разделе в форме, обеспечивающей самоинициализацию ее компонентов. Подсистема может тогда включать память для переменных вместе с командами; правило отделения команд от данных игнорируется. Этот подход является удовлетворительным, когда в подсистеме желательно или необходимо иметь в незначительной степени или совсем не иметь мультипрограммирования. Например, эта подсистема может быть так тщательно написана, что ввод-вывод и вычисления хорошо сбалансированы, или задания подсистемы находятся на самом нижнем приоритетном уровне, или уже существует эффективное мультипрограммирование в операционной системе.

Чистые процедуры

Чистые процедуры — это программы, которые не модифицируют свои собственные команды или локальные данные; это значит, что внутри этих процедур запоминание (запись) в операциях с памятью запрещены. Любая программа, которая является только выполняемой и читающей, называется также *реентерабельной* (IBM, 1965), так как ее выполнение может быть „продолжено“ в любое время с уверенностью, что ничего не изменилось. Таким образом, чистые процедуры могут разделяться более чем одним процессом как в режиме мультиплексирования времени, так и в режиме мультиобработки.

Изменяемые данные, связанные с процессом, использующим чистую процедуру, должны храниться в отдельных собственных областях. Эти данные обычно включают аргументы, адреса возврата и промежуточные данные. Наиболее удобен способ получения доступа к переменной информации через базовые регистры, которые будут указывать на личную память во время выполнения.

Пример

Пусть *ARG* — символьическое имя ячейки, содержащей некоторую переменную; $[ARG]$ означает содержимое *ARG*. Чтобы запомнить содержимое регистра *Ri* в *ARG*, может быть использована следующая команда в процедуре, не являющейся чистой:

STO Ri, ARG [ARG]:= [Ri];

Однако программа в общем случае некорректна, если эта команда является частью процедуры *P*, разделяемой несколькими процессами, так как каждый процесс может одновременно запоминать различные значения в *ARG*. (Это могло бы случиться, например, если бы команда находилась в середине итерационного цикла.) Чтобы избежать такого положения, *ARG* обычно является смешенным или относительным адресом внутри собственной области любого процесса, выполняющего процедуру *P*; предполагается, что некоторый (базовый) регистр, скажем *Rb*, во время выполнения содержит базовый адрес этой области. Тогда правильная команда обращения, которая сохраняет процедуру *P* „чистой“, может быть такой:

STO Ri, ARG(Rb) [[Rb] + ARG]:= [Ri];

Таким образом, регистры ЦОУ, которые запоминаются как часть вектора состояния процесса, обеспечивают простые средства для связывания чистых процедур с данными, доступными для изменения. Прежде чем вызывается чистая процедура, должна быть сделана подготовка для передачи через регистры параметров, адреса возврата и адреса ячеек временной памяти.

6.3. Разделение в системах со статическим распределением

Пусть память *статически* распределяется для каждого задания или шага задания в результате операции связывания, перемещения и загрузки. Внешние ссылки, появляющиеся в таблицах использования объектных модулей, могут относиться к потенциально разделяемым объектам; кроме того, программы, вызываемые посредством команд и предложений языка управления, также могут оказаться разделяемыми. Мы хотим коротко остановиться на проблемах и следствиях общего разделения единственной копии несколькими независимыми процессами, осуществляемого в общем виде в этих условиях. Предполагается, что несколько базовых регистров доступны для обращения к личным и разделяемым данным и для связи с разделяемыми процедурами.

Сначала рассмотрим машины *без* аппаратуры динамической настройки. Фактически во всех МС этих ЭВМ выполнено строгое разграничение между системными и пользовательскими программами, часто соответствующее привилегированному (ведущему) и ведомому режимам работы аппаратуры соответственно. Пользователи вызывают разделяемые системные программы и данные, выполняя команды супервизорных прерываний (*SVC*), но обычно не могут разделять информацию других пользователей. Чтобы обеспечить разделение на более общей основе, связывающие загрузчики должны быть модифицированы для поиска внешних ссылок и основных программ в основной памяти, а также во вспомогательной памяти и входных файлах. Кроме этого, обычное требование непрерывности для пространства памяти пользователя должно быть снято. (Почему?) Одна из важных проблем заключается в обеспечении соответствующей защиты. Когда защита реализуется только с помощью ключей памяти (разд. 5.4), разделяемая информация и вся память, связанная с разделяющими процессами, должна иметь один и тот же ключ, тем самым полностью открывая разделяющие процессы и их данные друг другу. Разделяемые области могут быть защищены объявлением их только для чтения и (или) выполнения, если этот тип защиты реализован в аппаратуре. Конечно, это решение неприемлемо, если рассматриваемая область должна модифицироваться (переписываться), как часто бывает, например, в рабочих областях пользователей, разделяющих компилятор или интерпретатор.

Трудно реализовать совершенно универсальные средства разделения на виртуальной памяти машины, которая динамически перераспределяется с помощью *чистой страничной организации* (без сегментации). Разделение данных, к которым производит-

ся доступ со стороны частных процедур, не представляет сколько-нибудь серьезных проблем. Загрузчик может присвоить данным каждого частного процесса свой набор страниц с последовательными номерами и настроить таблицы страниц для указания на соответствующие блоки. Во время выполнения в базовый регистр засыпается адрес конкретного виртуального пространства данных; частная процедура, выполняющаяся внутри процесса, обращается к данным через базовый регистр по способу, аналогичному примеру из прошлого раздела. Следует заметить, что области разделяемых данных при этом не могут содержать собственные адреса или адреса других областей, так как эти адреса неизбежно включают в себя номера страниц; могут использоваться только указатели относительно базы или другой фиксированной точки области.

Процедуры требуют больше осторожности, так как номера их страниц *заданы* (приписаны) во время связывания и загрузки; это значит, что все допустимые эффективные адреса пространства имен (адреса команд и локальных данных) внутри такой процедуры определяются во время загрузки. (Вспомните, что смена страниц невидима для пользователя.) Следовательно, разделяемые процедуры должны быть доступны для *всех* процессов, разделяющих эту программу в страницах с *одним и тем же* номером; когда новый процесс требует несколько независимых, разделяемых и уже загруженных сегментов, могут возникнуть конфликты относительно номеров страниц и бесполезный расход пространства памяти для таблиц (см. следующее упражнение). Можно обойти эту проблему, но существующие методы кажутся слишком грубыми. Если набор разделяемых сегментов известен системе в целом, часто бывает возможным *постоянно резервировать* номер их страниц; одни и те же номера используются для тех процедур и сегментов данных, которые взаимно отделены друг от друга (два сегмента отделены одним от другого, если они никогда не используются вместе одним и тем же процессом).

Когда механизм динамического перемещения действует через таблицы сегментов, с которыми, возможно, связаны таблицы страниц, упомянутая выше трудность исчезает, если мы предполагаем, что номера сегментов выполняющихся процессов появляются явно в базовых регистрах. Номера сегментов становятся частью вызывающего процесса, а не частью программы¹⁾; элементы таблицы сегментов у различных пользователей одной и той же процедуры могут тогда находиться в различных ячей-

¹⁾ Таким способом выполнена сегментация в GE 645. Системы B5500 и B6500 не используют базовых регистров для хранения номеров сегментов; номера сегментов появляются явно в командах.

ках. Элементы таблицы указывают на абсолютные области расположения разделяемой процедуры или, если память также страничная, могут указывать на разделяемые таблицы страниц. Защита, как и в случае чистой страничной организации, выполняется с помощью аппаратуры преобразования и распределения адресов. Ссылки на разделяемые данные, а также передача и возврат управления между разделяемыми процедурами также не прямые, так как номера сегментов для одной и той же разделяемой информации будут отличаться для каждого процесса; методы связывания, описанные в оставшейся части главы, применимы также к статическому случаю. В сегментных системах могут также быть выполнены ограниченные формы разделения с помощью постоянного резервирования некоторых номеров сегментов для разделяемой информации.

Упражнение

Рассмотрите проблемы статического разделения в системе с чисто страничной организацией (нет сегментации), предполагая следующую ситуацию: процесс p_a требует программы Q_1 и Q_2 , временную память T_1 , а также данные D_1 размером q_1 , q_2 , t_1 и d_1 страниц соответственно. Они объединяются и связываются в программу виртуального пространства размером $q_1 + q_2 + t_1 + d_1$ страниц в приведенном выше порядке и загружаются в память. В то время как процесс p_a остается активным, другой процесс p_b требует загрузки информации Q'_1 , Q_2 , T'_1 и D_1 с размерами (в страницах) q'_1 , q_2 , t'_1 , d_1 соответственно, где $q'_1 < q_1$ и $t'_1 > t_1$. Q_2 и D_1 должны быть разделены между p_a и p_b . Опишите возможное содержимое таблиц страниц для p_a и p_b . Какая проблема возникает, если теперь входит в систему новый процесс p_c и хочет разделять Q'_1 и Q_1 ? Можете ли вы придумать какой-нибудь общий способ устранения конфликтов в отношении номеров страниц, по-прежнему допуская разделения?

6.4. Динамическое разделение

Динамическое разделение рассматривается в самой общей форме, когда связывание процедур с разделяемой информацией (другими процедурами или данными) откладывается до последнего возможного момента — до времени обращения; этот подход может быть назван *динамическое связывание*. Предполагается, что мы имеем дело с многосегментным виртуальным пространством имен; как и в гл. 5, каждый адрес пространства имен состоит из пары: сегмент и слово $[s, w]$. Имеется аппаратурная настройка посредством таблиц сегментов, и совместно со связыванием может быть использована дисциплина динамического распределения памяти. Мы будем основываться на системе MULTICS (MULTICS, 1965; Далей и Денис, 1968) из-за

элегантности и общности методики, разработанной ее проектировщиками¹⁾.

Так как номера сегментов назначаются во время выполнения и в зависимости от обращающегося процесса могут различаться для одной и той же процедуры или сегмента данных, требуется, чтобы номера сегментов не появлялись в сегментах явно. Это относится как к внутренним (к самим себе), так и к внешним обращениям. Два различных процесса, разделяющие один и тот же сегмент информации, будут в общем случае обращаться к этому сегменту по различным номерам сегмента, причем конкретный приписанный номер является функцией текущего назначения номеров сегмента для процесса.

Внутренними ссылками легко управлять, используя базовый регистр для хранения номера сегмента выполняющегося процесса; номер сегмента тогда является частью вектора состояния процесса. Таким образом, в разделяемой программе появляются явно только номера слов. Внешние ссылки должны оставаться символическими по двум причинам. Во-первых, это единственный способ разрешить динамическое связывание и назначение номеров сегментов различным процессам, одновременно разделяющим ресурс. Вторая причина — обеспечение гарантии того, чтобы процедуры, которые используют разделяемую информацию, были чувствительны к большинству изменений в последней (например, в случае перекомпиляции). С другой стороны, если внешние ссылки всегда символические, то время доступа недопустимо большое из-за необходимости работы в режиме интерпретации. Хотелось бы, чтобы первое обращение данного процесса к каждому внешнему объекту выполнялось с интерпретацией символьической информации; это позволяло бы искать сегмент файловой системой, проверять защиту и назначать номера сегмента и, возможно, слова. Однако при последующих обращениях было бы более желательно обойти интерпретирующее программное обеспечение и осуществлять доступ через аппаратуру только через нормальный адрес вида $[s, w]$. Как это сделать и в то же самое время сохранить символьические ссылки и программы в чистом виде для потенциального использования другими процессами, описывается в следующих нескольких разделах. Мы будем использовать заглавные буквы для обозначения символьических ссылок и малые буквы для числовых (назначенных) адресов виртуальной памяти. Например, $[SUB, ENTRY]$ может быть символьской внешней ссылкой, тогда как $[sub, entry]$ может быть соответствующей парой типа „номер сегмента/номер слова“ после динамического связывания.

¹⁾ Мы здесь не будем обращать внимания на страничную организацию системы MULTICS; методы связывания почти независимы с точки зрения организации памяти.

6.4.1. Форма процедурного сегмента

Решение проблемы динамического разделения, принятное в системе MULTICS, основано на использовании двух типов интерфейсов:

1. Секция связи LS определяется для каждого потенциально разделяемого или разделяющего сегмента. Передача управления и внешние обращения к данным происходят с помощью косвенной адресации через секции связи. Каждый процесс получает собственную копию секции связи, когда он использует сегмент.

2. Личный магазинный сегмент назначается каждому процессу для запоминания аргументов, адресов возврата, состояний процессора и временной информации; сегмент организован как магазин для удобства вызова и возврата и так, что при эксплуатации каждая процедура может быть использована рекурсивно.

Адреса магазина и секции связи во время выполнения содержатся в форме $[s, w]$ в двух базовых парных регистрах: lp (указатель связи) и sp (указатель магазина) соответственно. Базовый регистр процедуры pbr всегда содержит номер сегмента программы, выполняющейся в настоящее время.

Сегмент каждой чистой процедуры состоит из трех частей — таблицы символов, чистой программы и секции связи (рис. 6.1). Для простоты мы допустим некоторые вольности при описании содержимого и структуры указателей этих частей. Таблицу символов можно рассматривать как таблицу определения и использования символов в перемещаемом объектном модуле. Рис. 6.2 содержит схему таблицы символов для гипотетического процедурного сегмента, названного P . Внутри P для возможного внешнего использования определены n символов E_1, E_2, \dots, E_n (например, точки входа); это означает, что они могут быть использованы как пары $[P, E_i]$ в других сегментах для обращения к соответствующей точке в P . Имена e_1, e_2, \dots, e_n обозначают номера слов, соответствующих точкам входа E_i в P (рис. 6.3), тогда как l_{e_i} являются указателями на секцию связи, используемыми для установления связей с P во время выполнения другими сегментами. Процедура P сама ссылается символически на слово X в сегменте данных D и на слово E в процедуре Q .

Чистая часть P , показанная на рис. 6.3, содержит указатели секции связи (номера слов) l_{Dx} и l_{QE} для внешних ссылок, при этом команда ADD может быть получена при трансляции из команды

$ADD [D, X]$

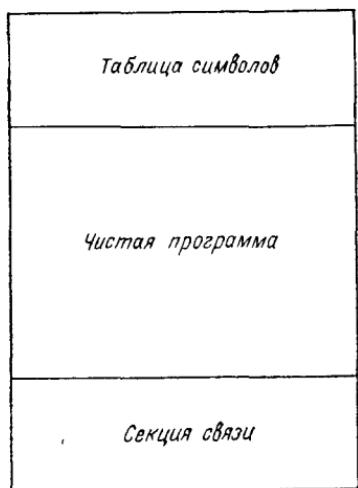
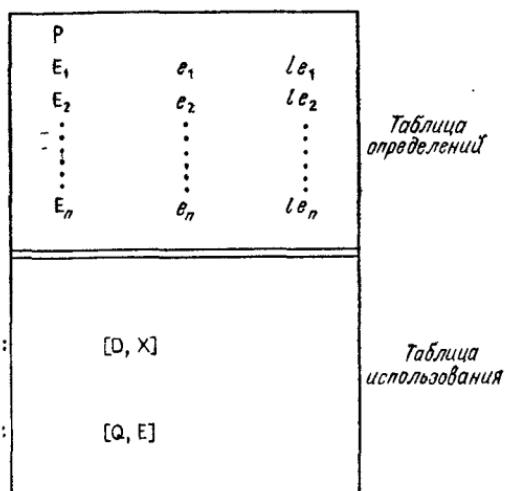


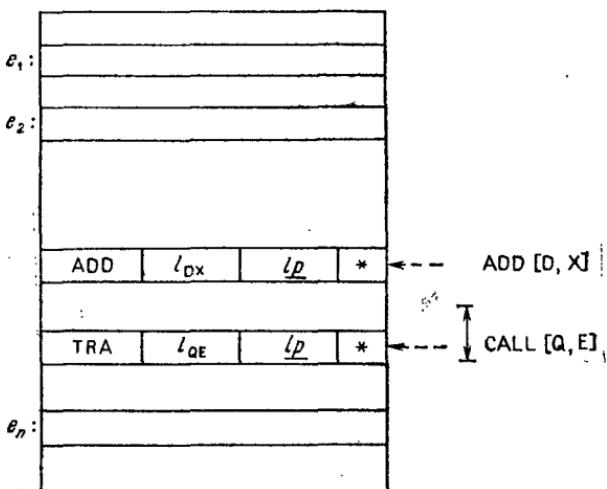
Рис. 6.1. Сегмент процедуры.

Рис. 6.2. Таблица символов для процедуры P .

и команда TRA — трансляцией команды вызова процедуры $CALL [Q, E];$

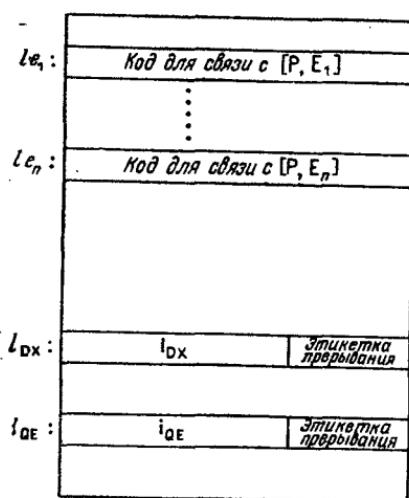
Значок * указывает на косвенную адресацию и lp означает, что эффективный адрес $[s, w]$ каждой команды вычисляется с помощью аппаратуры как

$$\left[segmentnumberin(lp), wordnumber(lp) + \left\{ \begin{array}{l} l_{DX} \\ l_{QE} \end{array} \right\} \right].$$

Рис. 6.3. Секция чистой программы для процедуры P .

Секция связи процедуры P , LS_P показана на рис. 6.4. Ссылки l_{DX} и l_{QE} есть указатели на символы D , X и Q , E в таблице символов. „Флаг прерывания“ есть флаг в поле адреса, который вызывает аппаратное прерывание или вызов супервизора, когда к ячейке обращаются во время косвенной адресации. Программа, соответствующая le_i , будет описана в разд. 6.4.3.

Необходимо подчеркнуть, что все три части P останутся неизменными. Когда процесс α связывается с P , он получает соб-

Рис. 6.4. Секция связи для процедуры P .

ственную копию LS_P , скажем LS_P^a ; адрес $[s, w]$ секции LS_P^a помещается в lp , поэтому все внешние ссылки доступны косвенно из P через LS_P^a .

6.4.2. Связывание данных

Теперь мы изучим метод обработки внешних ссылок к данным. Допустим, что процесс α выполняет сегмент P (рис. 6.2, 6.3 и 6.4) и только что начал выборку операнда по адресу $[D, X]$ в команде ADD ; пусть данная команда выполняется процессом α в первый раз. Регистр pbr содержит номер сегмента P в виртуальном пространстве процесса α , а lp указывает на копию секции связи LS_P^a , связанную с α [рис. 6.5(a)]. Первоначально LS_P^a идентична LS_P .

Операнд l_{DX} совместно с lp сначала вырабатывает адрес a (рис. 6.5(a)). Используя косвенную адресацию через a , механизм обращения по эффективным адресам производит выборку

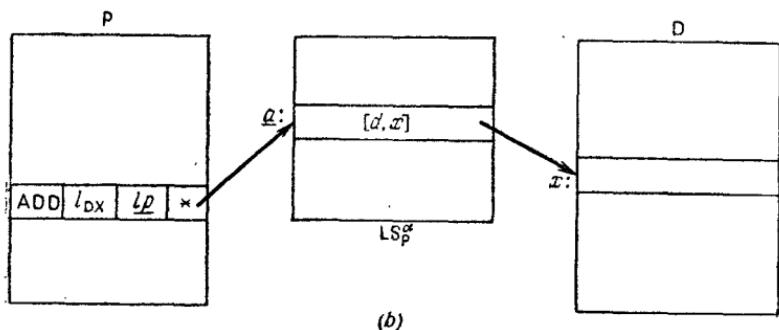
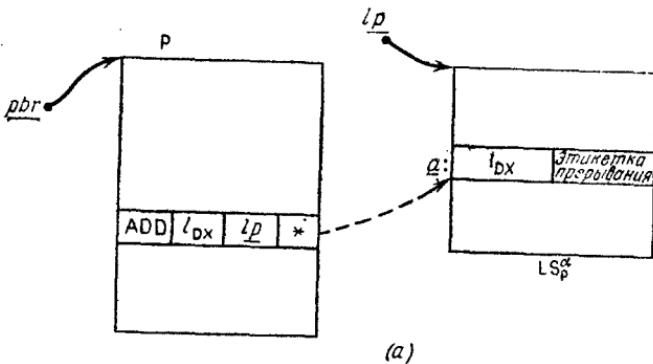


Рис. 6.5. Установление связи с данными $[D, X]$ процессом α в P : (a) — первое обращение к данным с помощью α в P ; (b) — завершающая связь с данными $[D, X]$.

содержимого a ; из-за „флага прерывания“ немедленно возникает внутреннее прерывание, и управление передается части программного обеспечения, называемой *установщик связи*. Основными функциями последнего являются установка правильной связи с $[D, X]$, т. е. выработка (d, x) и обеспечение того, что будущие обращения процесса α по адресу $[D, X]$ в P будут производиться автоматически.

Установщик связи находит символьическое имя данных, $[D, X]$, из секции связи и P , очевидным образом используя указатели. Затем вызывается модуль управления сегментами: он делает D „известным“ процессу α и возвращает номер сегмента D (обозначим его через d). Этот процесс может быть значительно усложнен за счет рассмотрения вызовов файловой системы, распределения памяти для таблиц сегментов (и таблиц страниц), возможно, перераспределения памяти, верификации „допустимости“ доступа и множества действий по изменению таблиц. (В гл. 9 более детально описываются функции файловой системы.) Тогда таблица символов сегмента D позволяет установщику связи получить x — номер слова X . Пара $[d, x]$ помещается в a , заменяя idx , а флаг прерываний сбрасывается. Процесс α готов к продолжению, а цепочка косвенной адресации завершается получением пары $[d, x]$. При всех последующих выполнениях a пара $[d, x]$ вырабатывается без каких-либо прерываний (рис. 6.5(b)). А сегмент P остается прежним для использования другими процессами.

Следует заметить, что методика MULTICS требует двукратного доступа к памяти, чтобы получить внешние данные этого типа (один — для косвенной адресации и второй — для собственно выборки данных) помимо обращений к памяти, необходимых для реализации сегментно-страничной организации. Трудно представить, как устранить косвенную адресацию и вместе с тем поддерживать символьическую информацию в чистой программе. Внешние данные, в форме аргументов и адресов возврата, не нуждаются в символическом представлении, так как стандартные вызывающие последовательности и использование магазина гарантируют то, что относительные положения этой информации известны. Регистр указателя магазина обеспечивает достаточные и эффективные средства для доступа к операнду данных в чистой программе¹⁾.

6.4.3. Обращения к процедурам

Передача управления от одного сегмента процедуры к другому более сложна, чем связывание данных. Система должна

¹⁾ Система MULTICS использует регистр аргументов *ар* также для указания списка аргументов.

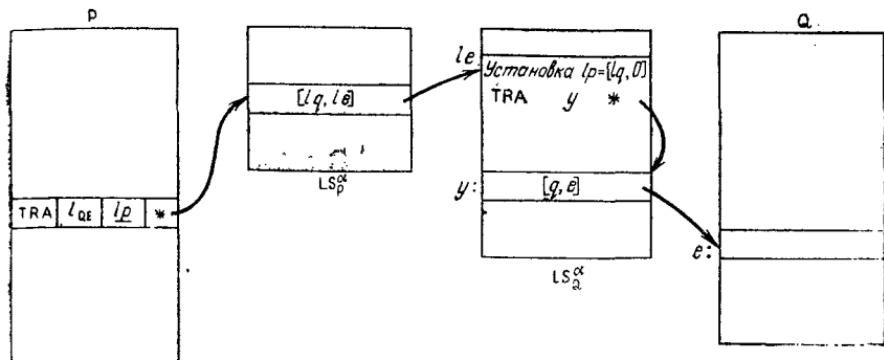


Рис. 6.6. Установление связи с процедурой Q процессом α из P .

не только установить связь, но должна также получить копию секции связи вызываемой процедуры, обеспечить занесение соответствующей новой пары $[s, w]$ в lp и нового номера сегмента в pbr . Мы опять описываем механизм связывания, используя пример, представленный на рис. 6.2, 6.3 и 6.4. Процесс α выполняет P и только начинает выборку операнда по адресу $[Q, E]$; пусть команда TRA выполняется процессом α впервые и в LS_P^a не было никаких изменений. Команды, предшествующие вызову Q , должны, конечно, обеспечивать аргумент и адрес возврата, а также временное пространство для магазина, которое может понадобиться процедуре Q .

Общая стратегия состоит в прохождении от P через LS_P^a и LS_Q^a и, наконец, к Q . Косвенная адресация и lp снова вызывают прерывание в LS_P^a . Установщик связи вызывает модуль управления сегментом, как и прежде передавая ему символическое имя сегмента Q . Сегмент Q становится „известным“ процессу α при создании личной копии, LS_Q^a , его секции связи и назначении номера сегмента Q в пространстве процесса α . Адрес $[s, w]$ для определенного символа E в LS_Q^a (обозначим его lq, le) заносится в LS_P^a в ячейку прерываний, а флаг прерывания сбрасывается; адрес в LS_P^a теперь указывает на внутреннюю часть LS_Q^a (рис. 6.6).

Трудность заключается теперь в том, чтобы обеспечить соответствующую программу в LS_Q^a для завершения передачи управления. На рис. 6.4 эта программа не была изображена. Первая операция заносит в lp пару $[s, w]$ — адрес LS_Q^a . Это выполняется посредством разрешения исходной команде TRA завершить свое выполнение так, чтобы управление действительно передавалось слову le в LS_Q^a . В этот момент prb содержит

номер сегмента новой секции связи;¹⁾ номер сегмента из *lp* может стать содержимым регистра *prb*, а номер слова из *lp* — значением базы LS_Q^{α} ²⁾. Целью следующей команды, косвенного ветвления (*TRA y**), является передача управления в желаемую точку входа *Q*, т. е. $[q, e]$ и значения *q* в *plr*. Адрес $[q, e]$, соответствующий $[Q, E]$, заносится установщиком связи после того, как сегмент *Q* идентифицирован в виртуальном пространстве процесса α . Теперь связь установлена полностью.

Процесс *Q* может теперь вернуться к процессу, вызвавшему его, т. е. к процессу *P*, с помощью ветвления через магазинные сегменты процесса α по адресу возврата. Последующее выполнение команды *CALL[Q, E]* процессом α в процедуре *P* будет автоматически проходить через секции связи без каких-либо прерываний, как показано на рис. 6.6.

Кроме выполнения работы по первоначальному установлению связи за универсальность данного способа разделения процедур приходится „расплачиваться“ двумя дополнительными командами и двумя выборками по косвенному адресу. Следует заметить, что приемы, описанные в этих последних разделах, могут также быть применены в системах со статическим распределением с помощью статически связывающего загрузчика, выполняющего функции динамического установщика связи; эквивалента флага прерывания не требуется.

¹⁾ Косвенные переходы к новым сегментам автоматически устанавливают в *rbr* значения нового номера сегмента.

²⁾ Секция связи часто рассматривается как полный сегмент, благодаря тому, что базовый номер слова устанавливается равным нулю; однако секции связи для единственного процесса могут также быть объединены в сегменты большего размера.

7. Управление процессами и ресурсами

В ответ на запросы и определения, выраженные в командном языке или языке управления заданиями системы, устанавливаются наивысшие уровни процессов пользователей в МС. Мы будем предполагать, что для каждого пользовательского задания J создается „супервизорный“ процесс наивысшего уровня p_J . Во многих системах это действительно имеет место, а там, где это не так, все же удобно теоретически подразумевать существование таких процессов p_J , несмотря на то что их функции могут быть выполнены централизованно системными процессами. Процесс p_J отвечает за контроль и управление прохождением задания J через систему, а также за глобальный учет и поддержание структуры данных для описания ресурсов задания. Структура данных может включать в себя статическую информацию, такую, как идентификация задания, максимальное время и требования к вводу-выводу, приоритет, „класс“ (например, интерактивный, пакетный, с ограниченным вводом-выводом, с ограниченными вычислениями) и потребности в ресурсах (например, потребности в основной памяти, файлах, периферийных устройствах); а также динамические данные, относящиеся к использованию ресурсов и к текущим процессам в задании. Вообще, p_J будет создавать ряд процессов-потомков, причем каждый соответствует некоторой затребованной единице работы, такой, как шаг задания; эти процессы могут в свою очередь создавать другие процессы или последовательно или параллельно. Таким образом, по мере прохождения задания изменяется структура соответствующего иерархического дерева процессов. Процессы в общем случае не независимы, а взаимодействуют друг с другом и с такими частями операционной системы, как распределители¹⁾ ресурсов, планировщики и компоненты файловой системы. Эти идеи иллюстрируются рис. 7.1, где процессом ps операционной системы созданы процессы заданий p_{J1}, \dots, p_{Jn} .

В этой главе мы определим с некоторой детализацией набор структур данных и базовые операции для процессов и ресурсов, рассмотрим обработку прерываний и процессы ввода-

¹⁾ В оригинале „managers“. — Прим. перев.

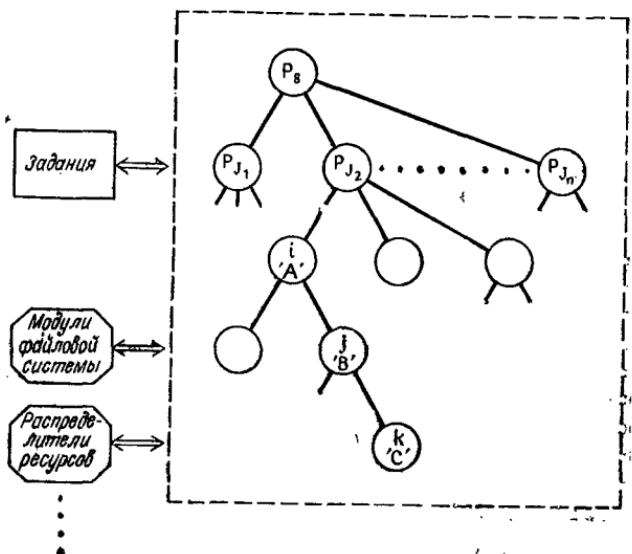


Рис. 7.1. Иерархия процессов для пользовательских заданий.

вывода, а также несколько организаций и методов распределения процессоров процессам, находящимся в состоянии готовности, т. е. методы планирования или диспетчеризации. Нашей целью является: собрать воедино материал и обогатить содержание предыдущих глав, относящихся к примитивам процессов и ресурсов, а также к другим частям ядра системы, представить одну возможную модель организации ядра¹⁾ и использовать эту модель как основу для обсуждения разнообразия методов обработки.

7.1. Структуры данных для процессов и ресурсов

Каждый класс ресурсов и каждый процесс обычно имеет соответствующую структуру данных, отражающих его основное „состояние“, идентификатор и учетную информацию. Эти структуры данных иногда называют системными *управляющими блоками* или *дескрипторами*. Дескрипторы в совокупности представляют состояние операционной системы и обрабатываются системными стандартными программами, которые к ним часто обращаются.

¹⁾ Реализация модели ядра, представленного в этой главе, описана в работе Шоу и др. (1973).

7.1.1. Дескрипторы процесса

Дескриптор процесса строится при создании процесса и представляется процесс во время его существования. Эти структуры данных модифицируются и широко используются основными операциями над процессами и ресурсами, стандартными программами обработки прерываний и планировщиками; к ним обращаются во время анализа производительности, а также для управления ею. В табл. 7.1 перечислен возможный набор дескрипторных данных для процесса в МС общего назначения. Индексированная переменная, определяющая или следующая за компонентом в таблице, будет использоваться для обозначения компонента и его содержимого; данный элемент сам может иметь сложную структуру, как описано ниже.

Таблица 7.1 Структура данных процесса

Процесс i

1. Имя $Id[i]$
2. Виртуальная/реальная машина
 - 2.1. Состояние процессора и полномочия¹⁾ $CPUstate[i]$
 - 2.2. Процессор $[i]$
 - 2.3. План основной памяти $MainStore[i]$
 - 2.4. Другие распределяемые ресурсы $Resources[i]$
 - 2.5. Список созданных ресурсов $CreatedResources[i]$
3. Информация о состоянии
 - 3.1. $Status[i]$
 - 3.2. Данные, связанные с состоянием $Sdata[i]$
4. Прямые родственники
 - 4.1. Отец $[i]$ ($Parent[i]$)
 - 4.2. Сын $[i]$ ($Progeny[i]$)
5. Данные учета и планирования $Priority[i], \dots$

¹⁾ В оригинале capability. В переводе книги: Д. Цикртэнс, Ф. Бернстайн. Операционные системы.—М.: Мир, 1977—этот термин переведен как «возможность». В данном переводе выбран синоним термина «полномочие» как наиболее отвечающий смыслу в дальнейшем контексте.—Прим. перев.

Идентификация

Каждый процесс имеет уникальный внутренний идентификатор, например целое i , обеспечиваемый системой в момент создания процесса, и внешнее имя $Id[i]$. Последнее предназначено для того, чтобы обеспечить удобные явные межпроцессные ссылки. Чтобы устранить конфликты, возникающие при использовании одного и того же внешнего имени более чем одним процессом, мы принимаем соглашение, что „дети“ любого процесса в древовидной иерархии имеют уникальные имена, и полностью определяется возможность обращения процесса i к процессу j .

как упорядоченная конкатенация имен Id процессов на „пути“, соединяющем i с j (включая $Id[j]$); например, на рис. 7.1, процесс i может обратиться к процессу k по имени BC , где $Id[j] = B$ и $Id[k] = C$. (Более подробно об этом типе ссылок сказано в гл. 9, где аналогичная схема используется для именования файлов в справочниках файлов.)

Определение виртуальной/реальной машины

Второй компонент, спецификация виртуальной/реальной машины, определяет вектор состояния процесса в смысле наших предварительных замечаний в разд. 3.2 и 4.3. В данном разделе описаны состояние и полномочия виртуального или реального процессора, на котором развивается процесс i , адресное пространство и любые другие ресурсы, распределенные для i . Отображения виртуальной машины на реальную, если требуется, также включаются в каждую часть этого компонента.

Элемент $CPUstate[i]$ содержит данные о полномочиях процессов и о защите, а также текущее содержимое счетчика команд и регистров в случае, когда процесс i находится в состоянии готовности или заблокирован, но не продолжается. Состояние $CPUstate$ можно считать эквивалентным информации, находящейся в типичном слове состояния машин третьего (или четвертого) поколения, т. е. данным, аппаратурно сохраняемым при прерывании. Полномочие тогда выражается в наборе команд, которые процесс вправе выполнять. Такие команды определяются, например, указанием способа ведущего и ведомого¹) и, возможно, ключа защиты памяти для адресного пространства процесса. Форма самозащиты определяется типами прерываний (внутренних прерываний), которые разрешаются, запрещаются или задерживаются во время выполнения процессов. Полномочия и защита могут быть реализованы программно или аппаратно²), например с помощью „кольцевой“ защиты или матриц путей доступа (см. разд. 5.3) или перечислением ресурсов, которые процесс имеет право запросить и освободить. Если развивается процесс i , то выражение $Processor[i]$ идентифицирует с помощью номера (центральный) процессор, на котором развивается i .

План основной памяти описывает пространство реальных адресов процесса и отображение имен в ячейки, если таковое используется. Компонент $MainStore[i]$ может содержать верхнюю и нижнюю границы памяти или указатель на таблицу сег-

¹⁾ Состояние «Супервизор/задача» в ЭВМ ЕС. — Прим. ред.

²⁾ Здесь речь идет о механизме контроля и настройки на тот или иной набор команд. — Прим. ред.

ментов, или указатель на таблицу страниц. С помощью этого компонента можно найти ресурсы основной памяти, распределенные процессу, включая разделяемые. Следующим компонентом *Resources*[*i*] является указатель на список всех других повторно используемых ресурсов, распределенных процессу *i*. Ими могут быть такие повторно используемые ресурсы, как периферийные устройства, пространство вспомогательной памяти и критические секции; список обычно содержит подробные данные о распределении, например секторные адреса пространства на диске. Каждый элемент списка ресурсов и компонента *Main-Store*[*i*] имеет флаг, показывающий, является ли ресурс разделяемым или используемым монопольно. Список созданных ресурсов есть список всех классов ресурсов, созданных процессом.

Данные о состоянии

Состояние процесса *i* описывается индикатором *Status*[*i*] и элементом данных *Sdata*[*i*], указывающим на связанный с ресурсом список, по которому определяется состояние. Как было определено в предыдущих главах, компонент *Status*[*i*] может принимать следующие значения:

- running*: процесс *i* развивается на процессоре (обозначенном *Processor*[*i*]);
- ready*: процесс *i* готов к выполнению, ожидая процессора;
- blocked*: процесс *i* в принципе не может развиваться, пока не получит конкретный ресурс или сообщение.

В терминах этих типов состояний можно управлять многими ситуациями, но существуют некоторые применения, в которых желательно разделять типы состояний более детально. Рассмотрим следующие два примера:

1. Пользователь интерактивно отлаживает или выполняет программу. У него часто будет возникать желание приостановить вычисления, чтобы определить состояние своего задания, может быть, сделать некоторые изменения и либо продолжить, либо прекратить выполнение.

2. Внутренний процесс может пожелать приостановить работу одного или более других процессов, чтобы определить или, возможно, модифицировать их состояние. Целью приостановки может быть, например: обнаружить или предотвратить некоторые виды тупиков или обнаружить и уничтожить „убежавший” (неконтролируемый) процесс.

В обоих случаях требуется „блокировать“ процесс, чтобы достичнуть приостановки. Однако процесс мог быть уже заблокирован в момент, когда его желательно приостановить. Если мы не хотим разрешать процессу блокироваться более чем по одному „ресурсу“ или быть заблокированным каким-нибудь другим процессом, требуется новое состояние „приостановки“¹⁾.

Процесс будет или *активным* или *приостановленным*. Если он активный, то может быть в одном из состояний: развития, готовности к развитию или в состоянии блокировки, что фиксируется в $Status[i]$ значениями *running*, *readya*, *blockeda* соответственно. В случае приостановки $Status[i]$ есть *readys* или *blockeds*.

$Sdata[i]$ указывает на ресурс и имеет значение:

$Status[i]$	$Sdata[i]$
1. <i>running</i> <i>ready</i> (s_a)	Указатель на список готовности, используемый планировщиком процессов для распределения ресурса процессора
2. <i>blocked</i> (s_a)	Указатель на список ожидания, связанный с ресурсом, вызвавшим блокировку

Детальные списки готовности и ожидания приведены в следующем разделе, в котором идет речь о дескрипторах ресурсов.

Другая информация

В начале этой главы был кратко рассмотрен метод создания иерархий процессов. Каждый процесс i имеет отца $Parent[i]$, который создал процесс и который является „хозяином“ любого из своих потомков и управляет ими²⁾. Аналогично, каждый процесс i может создать другие процессы; компонент $Progeny[i]$ состоит из списка внутренних имен прямых потомков процесса i . Приоритет процесса $Priority[i]$ представляет собой, по существу, статическую или динамическую величину относительной важности процесса i ; мы будем использовать соглашение, что условие $Priority[i] > Priority[j]$ приводит к тому, что процесс i имеет приоритет выше, чем j . Приоритет не обязательно является внешним приоритетом, приписанным заданию; последний может рассматриваться как „долговременный“ приоритет,

¹⁾ Мы будем предполагать, что блокировки являются всегда самоблокировками, которые свойственны всем примитивам, кроме $Block(i)$, рассмотренного в гл. 3.

²⁾ Будет существовать прародитель, созданный когда система впервые загружается.

используемый для определения того, когда задание должно быть загружено и активизировано, в то время как *Priority*[*i*] в основе своей „кратковременный“ и, возможно, динамический приоритет, применяемый планировщиком процессов при выборе процесса для развития. *Priority*[*i*] может быть сложной функцией от внешнего приоритета, от требований процесса к ресурсам и от текущей операционной среды, т. е. других процессов в системе. Последняя часть в структуре данных процесса используется для целей измерения производительности, планирования, учета расходов и распределения. В нее включаются такие типовые элементы, как время ЦОУ, используемое процессами, остающееся время (в соответствии с некоторой оценкой); используемые ресурсы, запрошенные ресурсы, а также число запросов ввода-вывода с момента создания процесса.

7.1.2. Дескрипторы ресурсов

В этом разделе представлен „скелет“ структур данных, с помощью которых можно описать состояние любого большого числа объектов, называемых ресурсами; причем типов ресурсов оказывается гораздо больше, чем обычно предполагают. Структуры данных и приведенные ниже основные разработанные Вейдерманом (1971) операции с ресурсами объединяют широко используемые отдельные нотации.

Термин „ресурс“ обычно применяется по отношению к повторно используемым, относительно стабильным и часто недостающим (дефицитным) объектам, которые запрашиваются, используются и освобождаются процессами в период их активности. Этот термин наиболее часто используется для описания компонентов аппаратуры. Однако очевидно, что многие типы программного обеспечения — данные и программы — также удовлетворяют общему определению. Вышеупомянутые ресурсы называются *повторно используемыми* (*serially reusable*), поскольку они обычно (но не всегда) разделяются процессами на последовательной основе, т. е. в любой момент времени используются одним процессом. Каждый класс ресурсов требует для описания по крайней мере три компонента:

1. *Опись*, включающую число и идентификацию доступных единиц ресурса.

2. *Список ожидания* блокированных процессов с неудовлетворенными запросами на ресурсы.

3. *Распределитель*, ответственный за принятие решения, который из запросов должен быть удовлетворен и когда.

Существует и другой набор объектов, которые разделяют одни и те же средства описания как повторно используемые

ресурсы. Они появляются в ситуациях синхронизации и связи там, где между процессами передаются сообщения, сигналы и данные. Для каждого класса сообщений в данный момент времени существует „опись“ сообщений, которые были выданы, но не были получены (потреблены), список ожидания (возможно, пустой) процессов, которые затребовали некоторую информацию, и некоторый механизм распределения, который распределяет сообщения получателям. Мы будем называть эти ресурсы *потребляемыми*, следуя Холту (1972), так как они используются прежде всего в приложениях типа „производитель — потребитель“, каждый из них имеет короткое время жизни в системе и не „возвращается“ системе после использования. Части повторно используемых ресурсов могут быть освобождены или возвращены в опись ресурса только, если они предварительно были приобретены, тогда как элементы потребляемых ресурсов могут быть освобождены независимо от любых предыдущих приобретений.

Потребляемые и повторно используемые ресурсы имеют одно общее важное свойство: процесс может стать логически блокированным благодаря неудовлетворенному запросу на такой ресурс. Это приводит к общему определению:

Логическим ресурсом, в дальнейшем называемым просто *ресурсом*, является нечто, что может вызвать переход процесса в логически блокированное состояние.

Исторически класс (логических) ресурсов называется *семафором ресурса*; элементы его дескриптора приведены в табл. 7.2. Компонент *Sdata* в дескрипторе процесса будет указывать на семафор ресурса, если процесс заблокирован. Семафоры ресурсов и их дескрипторы устанавливаются *динамически* во время развития процесса, когда процесс создает новый класс ресурсов.

Таблица 7.2. Структура данных ресурса

- Семафор ресурса *i*
1. Идентификация *Id[i]*, *SR[i]*, *Creator[i]*
 2. Опись доступности *Avail[i]*
 3. Список ожидающих процессов *Waitrs[i]*
 - 4 Точка входа в распределитель *Allocator[i]*
 5. Другая информация

Как и в случае дескриптора процесса, каждый семафор будет иметь внутреннее и внешнее имя, определяемое системой, и родителя во время создания (определения) ресурса. Булевый индикатор *SR[i]* обозначает, является ли ресурс повторно используемым (*SR[i] = true*) или потребляемым. В компоненте *Resources[i]* дескриптора процесса описываются только распределенные ресурсы типа повторно используемых. Кроме этого, поле *Creator[i]* содержит внутреннее имя процесса, который создал семафор ресурса.

Указатель описи

Ссылка $Avail[i]$ указывает на начало списка доступности или опись для семафора ресурса i . Сам список может представлять собой единственный бит, указывающий на наличие или отсутствие сигнала прерывания определенного типа, или таблицу блоков основной памяти, или связный список буферных цеплов. Список также может представлять собой некоторые более сложные и структурированные данные, такие, как адреса устройств вспомогательной памяти, которые содержат справочники доступности для конкретного устройства. Заголовок списка содержит также указатели (точки входа) по крайней мере на две стандартные программы, одну — для вставки новых элементов (освобождаемых ресурсов) и другую — для удаления распределенных ресурсов; эти возможности аналогичны тем, которые описаны ниже.

Список ожидающих процессов

В этом списке можно найти все процессы, заблокированные на семафоре ресурса. Запись о каждом процессе будет содержать его уникальный внутренний идентификатор и адрес области, в которую позднее будет помещена подробная информация о распределении, а также информацию о запросе на ресурс. Последняя может включать в себя размер запроса, его тип, например запрос на задание с ограниченными вычислениями из списка ресурсов задания или запрос на 7-дорожечную или 9-дорожечную ленту, и идентификатор процесса (процессов), который может предоставить ресурс (например, послать сообщение) запрашивающему объекту.

Списки ожидания могут быть различного вида. В простейшем случае список содержит один элемент. Это встречается в тех случаях, когда процесс только блокируется и ждет сигнала от другого процесса. Вероятно, наиболее эффективно объединить все ресурсы этого типа в больший класс, так как механизм распределения будет одним и тем же.

Универсальной организацией для списка ожидания является структура обобщенной очереди; заголовок этой структуры адресуется с помощью указателя $Waitrs[i]$. Хотелось бы неформально определить операции вставки и удаления элементов очереди независимо от особенностей ее организации. С этой целью для заголовка очереди предложена структура табл. 7.3. Тогда вызовы

1. $Insert(q, e);$
2. $e := Remove(q);$

могли бы использоваться соответственно для:

1. Вставки e в очередь q (т. е. вызова программы, адресуемой $Insert[q]$ с аргументом e).

2. Удаления элемента из очереди q и помещения его указателя или индекса в e .

Таблица 7.3. Заголовок обобщенной очереди

1. Точка входа в стандартную программу вставки $Insert[q]$
2. Точка входа в стандартную программу удаления $Remove[q]$
3. Первый элемент $First[q]$
4. Последний элемент $Last[q]$
5. Дополнительные данные

Удобно также иметь другую программу удаления (или точку входа в $Remove$), которая удаляет названный элемент из списка очереди; мы будем использовать обращение для изъятия из q элемента, названного i , в виде:

$e := Remove(q, i);$

Другие стандартные программы доступа к очереди, такие, как программы для просмотра очереди, могут быть написаны с использованием программ $Insert$ и $Remove$, или их точки

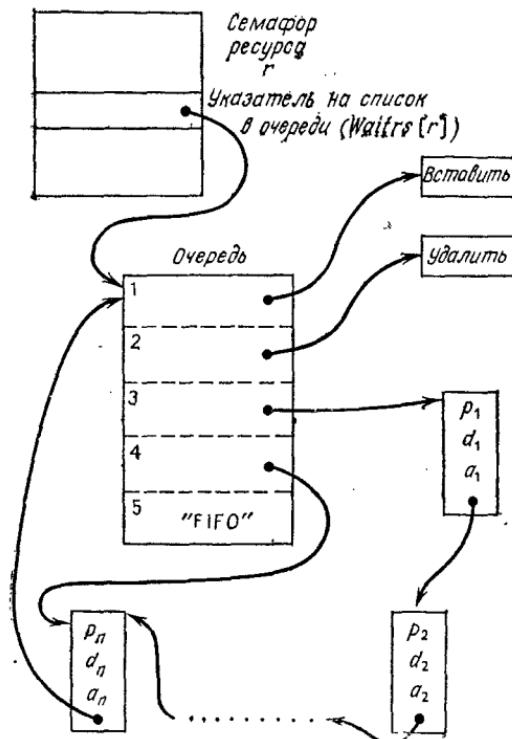


Рис. 7.2. Список ожидания FIFO.

входа могут быть занесены в последний элемент заголовка очереди. Последний компонент заголовка очереди может также применяться для хранения статистических данных об очереди и индикаторов для использования процедурами измерения производительности и распределения ресурсов, а также для идентификации структуры очереди.

Очередь FIFO могла бы иметь такой вид, как показано на рис. 7.2. Идентификатор процесса r_i может быть просто указателем на дескриптор процесса, элемент d_i представляет подробные сведения о запросе, а a_i есть базовая ячейка, куда должна быть помещена информация о распределении, когда запрос будет удовлетворен. Следует заметить, что эта структура FIFO аналогична организации буферного пула, описанного в гл. 2. Структура рис. 7.2 могла бы быть упрощена, если использовать простую сплошную таблицу, рассматриваемую как неявный циклический список. Преимущество явно связанного списка заключается в том, что длинные аргументы не должны удаляться; в записях должны встречаться только указатели на различные элементы.

Часто бывает полезно некоторым способом упорядочить элементы очереди в списке ожидания; упорядочение может быть основано, например, на размере запроса или некотором приоритете запрашивающего процесса. Аналогично очереди FIFO могли бы быть применен связанный в прямом направлении список, и порядок мог бы поддерживаться таким образом, что группы наивысшего приоритета остаются в вершине. Это несколько усложняет вставку, но может явиться самым эффективным методом, если ожидается, что списки будут не слишком длинные. Если списки ожидания потенциально длинные, то полезна структура данных с приоритетом. Пусть приоритеты пронумерованы от 1 до n . Тогда очередь может быть организована следующим образом (Лэмпсон, 1968): компоненты $First[q]$ и $Last[q]$ указывают на первый и последний элементы в таблице приоритетов ($Last[q]$ не всегда является обязательным). Каждый элемент i в таблице возглавляет двусвязный список ожидания типа FIFO процессов с приоритетом i и имеет прямой указатель очереди $fqp[i]$ и обратный указатель, $bqr[i]$, которые адресуют процессы приоритета i в списке, если таковые имеются. На рис. 7.3 приведен пример такой структуры данных; процессы 6, 8 и 5 ожидают с приоритетом 1, ни один процесс с приоритетом 2 не ожидает, а процесс 2 ожидает с приоритетом 3. Прямые и обратные указатели удобны для изменения списка, когда динамически меняются приоритеты процесса или когда процессы удаляются из списка. Такая структура данных также является естественной для списка процессов в состоянии готовности и будет использована для этой цели в разд. 7.4.2.

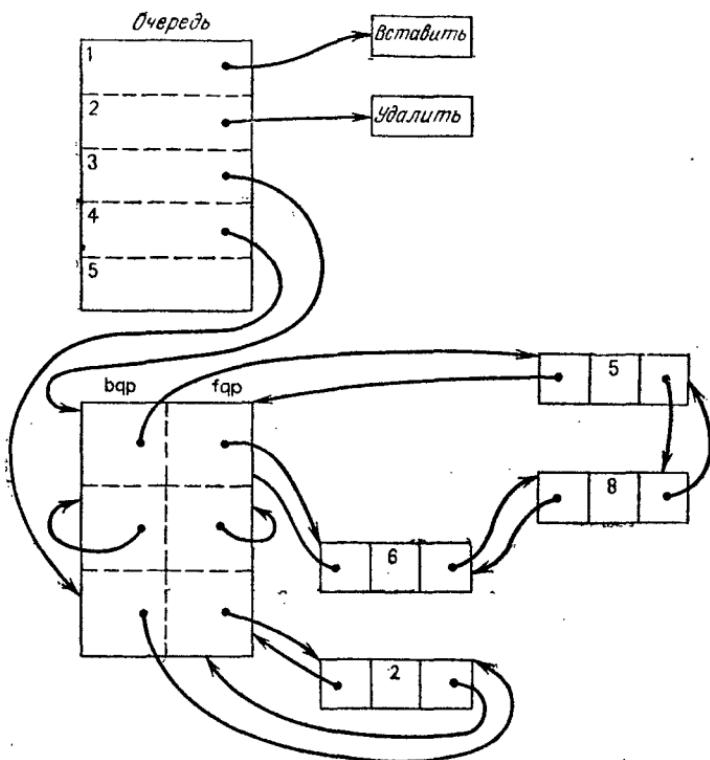


Рис. 7.3. Структура данных приоритетного списка ожидания.

Распределитель

Распределитель, указанный в семафоре ресурса, — это стандартная программа, которая стремится сопоставить доступные ресурсы запросам блокированных процессов. Если ресурсы распределены, он записывает сведения об этом распределении в специально предназначено для этого поле параметров запроса. Будем считать, что распределитель удовлетворяет запросы от одного или более процессов при однократном обращении к нему. Это, конечно, желательно, когда ресурсы добавляются к списку доступности, так как в результате такого действия может быть разблокировано несколько процессов. Менее очевидно, что может быть желательным отсрочить распределения до некоторого момента в будущем, даже если ресурсы доступны в момент запроса. Это может оказаться необходимым, чтобы предотвратить тупик, или в ожидании будущих запросов, или из соображений эффективности. Будем предполагать сле-

дующую форму обращения:

Allocator(*r*, *k*, *L*)

где *k* и *L* есть выходные параметры со значением:

1. *k*: число процессов (незаблокированных), которым распределены ресурсы, $k \geq 0$;
2. *L*[*i*], $i = 1, \dots, k$: список процессов, которым распределены ресурсы (определен только тогда, когда $k > 0$);
3. *r* — внутреннее имя семафора ресурсов.

Дополнительная информация

Последнее поле в дескрипторе ресурса может применяться для нескольких целей. Так как поведение системы в значительной степени зависит от требований на ресурсы и их использование, удобно сохранять в этом поле некоторые измерительные данные. Здесь может также храниться текущее распределение элементов семафора ресурсов.

Ресурс процессора

Существует один ресурс, который нужно рассмотреть отдельно. Это центральный процессор (центральные процессоры). Дело в том, что мы не считаем процесс логически блокированным, если он ожидает только доступности процессора. В системе, где может быть более одного центрального процессора, планировщику для такого ресурса требуется своя структура данных. Для простоты будем предполагать, что *все центральные процессоры эквивалентны* в том отношении, что процессу в состоянии готовности может быть назначен (распределен) любой из них. На практике это может быть не так, потому что процессоры могут отличаться по скорости, объему и типу внутренней доступной памяти и по их возможности взаимодействовать с другими процессами, устройствами памяти и периферийными устройствами. Наш дескриптор процессора состоит из двух компонентов:

1. *np*: число процессоров ($np \geq 1$)
2. *Process*[*i*], $i = 1, \dots, np$: идентификатор процесса, развивающегося на процессоре *i*.

Планирование процессов упрощается, если мы допустим, что каждый процессор всегда активен и на нем развивается некоторый процесс. Чтобы обеспечить эту непрерывную активность, для каждого процессора создается циклический „холостой“ процесс; холостой процесс развивается, если в системе не существует других процессов в состоянии *readya*.

Дескрипторов процесса и ресурса вполне достаточно, чтобы иметь дело с самыми разнообразными мультипрограммными

системами. Однако немногие коммерческие системы имеют такую обобщенную структуру дескрипторов. Обычно каждый класс ресурсов имеет свою собственную структуру данных, которая ограничена (определенна) при генерации системы. Дескриптор процесса, как правило, содержит поля для каждого аппаратурного ресурса, который может быть распределен, и битовые строки для индикации причины ожидания. За наши более универсальные структуры приходится расплачиваться эффективностью: требуется память для хранения указателей и дополнительные вычисления для выполнения основных операций над процессами и ресурсами. Выгоды от такой систематической организации прежде всего вытекают из ее ясности и единства. Можно легко вносить изменения, чтобы приспособиться к новым условиям, система легче для понимания и, следовательно, для анализа и оценки, а также существует возможность реализации примитивов (базовых операций) над этими структурами в аппаратуре, например посредством микропрограммирования.

7.2. Основные операции над процессами и ресурсами

В следующих разделах определяется множество примитивов в составе ядра. С помощью этих примитивов производятся манипуляции по поддержанию состояний нашей гипотетической МС, по обработке дескрипторов процессов и ресурсов. Будем придерживаться точки зрения, что каждый примитив представляет команду, которая выполняется в пределах вызывающего процесса, а не рассматривать каждый примитив как отдельный процесс. Примитивы считаются неделимыми и защищаются общим замком типа „занятое ожидание“; для простоты в последующих описаниях мы опустим рассмотрение замков. Проверка полномочий и ошибок также опускается. Эти вопросы обсуждаются после того, как соответствующие операции будут определены (в разд. 7.2.3).

7.2.1. Управление процессом

Определяются пять операций над процессами

1. *Create*: создать новый процесс.
2. *Destroy*: уничтожить один или более процессов.
3. *Suspend*: изменить состояние процесса на „приостановленный“.
4. *Activate*: изменить состояние на „активный“.
5. *ChangePriority*: установить новый приоритет процесса.

Межпроцессные связи организуются общими операциями над ресурсами, описанными в разд. 7.2.2.

Чтобы создать нового потомка n , процесс-отец будет вызывать примитив *Create* с аргументами: внешнее имя n , начальное состояние ЦОУ S_0 , приоритет k_0 , начальная основная память M_0 , начальная опись других ресурсов R_0 и, возможно, учетная информация и ограничения. Начальное состояние будет *readys* — „готов“ (*ready*), потому что новый процесс n должен быть в состоянии логически продолжающегося, и „приостановленный“ (*suspended*), так как может оказаться желательным создать процесс до его активизации. Системный список готовности процессов RL будет приоритетно организованной структурой в форме, показанной на рис. 7.3. Примитив *Create* описывается следующей процедурой:

```

procedure Create (n, S0, M0, R0, k0, accounting);
begin
  i := GetNewInternalProcessName;
  Id[i] := n; CPUstate[i] := S0;
  MainStore[i] := M0; Resources[i] := R0; Priority[i] := k0;
  Status[i] := 'readys'; Sdata[i] := RL;
  Parenl[i] := *; Progeny[i] := Λ;
  Insertl(Progeny[*], i);
  Set Accounting Data;
  Insertl(RL, i)
end Create

```

Первая команда устанавливает внутреннее имя i для n , и i становится известным системе; при распределении передаваемых в процедуру атрибутов для имени i предполагается также и распределение пространства для дескрипторов процесса. Звездочкой (*) обозначается внутреннее имязывающего процесса. Оно может быть получено или из регистра, или программным путем. Последнее предложение *Create* вносит новый процесс в список готовности. Начальные ресурсы, включая основную память, определяются как *разделяемые* ресурсы для нового процесса и должны быть подмножеством ресурсов процесса-предка; процесс-предок может аналогичным способом разделять любые из своих ресурсов с другими процессами-потомками. Созданный процесс в свою очередь может разделять свои ресурсы со своими „детьми“ и может также получать в дальнейшем ресурсы, которые будут его *собственными*.

Процессу будет разрешено приостанавливать только своих „потомков“. Операцию приостановки можно было бы тракто-

вать двумя различными способами. По вызову *Suspend(n, a)*, где *n* есть имя процесса и *a* указывает на область памяти возврата, можно было бы приостановить только *n* или *n* вместе с его „потомками“; могут потребоваться обе эти возможности. Последняя возможность является отчасти излишеством, так как некоторые „потомки“ могли быть приостановлены своими „предками“ (которые являются „потомками“ процесса *n*). Система RC4000 (Бринч Хансен, 1969) приостанавливает процессы таким способом, ибо как следствие требует более сложных типов состояний процесса (см. упражнение). Наше решение должно разрешать за один раз приостановку только одного процесса, но должно передавать достаточно информации приостанавливающему процессу, с тем чтобы некоторые или все остальные его „потомки“ могли быть также приостановлены, если это желательно. Информация для достижения этой цели легко доступна, так как чтобы позволить приостанавливающему процессу определить состояние приостанавливаемого, копия последнего возвращается в области памяти, указанной параметром *a* в операции приостановки. (Можно было бы, напротив, разрешить доступ к дескриптору процесса только для чтения и вернуть указатель дескриптору, но это может оказаться трудно-реализуемым.)

```

procedure Suspend(n, a);
begin
  i := GetInternalName(n); s := Status[i];
  if s = 'running' then Stop(i);
  a := CopyDescriptor(i);
  Status[i] := if s = 'blockeda' ∨ 'blockeds' then 'blockeds'
                else 'readys';
  if s = 'running' then Scheduler
end Suspend

```

Операция *Stop(i)* предназначена для прерывания *Processor(i)*, сохранения соответствующей части состояния процессора в *CPUstate[i]* и установки состояния процесса *Process[Processor[i]]* в состояние Ω (неопределенное); тем самым операция делает процессор доступным. Тогда в конце операции приостановки вызывается планировщик процессов, чтобы распределить процессор некоторому другому процессу в состоянии *readya*. Приостановленный процесс остается в составе списка, который он занимал до приостановки.

Активизация процесса проста и естественна. Она предполагает изменение состояния процесса на активное и возможное об-

ращение к планировщику. Последний допускает вариант планирования с приостановкой распределения, если состояние активизируемого процесса становится *readya*. Процесс может активизировать любого из известных ему „потомков“, в частности своего „сына“. Таким образом, нормальная последовательность действий для введения нового процесса в систему состоит из операции *Create*, за которой следует *Activate*.

```

procedure Activate(n);
begin
  i := GetInternalName(n);
  Status[i] := if Status[i] == 'readys' then 'readya' else 'blockeda';
  if Status[i] == 'readya' then Scheduler
end Activate

```

Для уничтожения процесса предусмотрены те же самые альтернативы, что и в операции *Suspend*, — можно или удалить единственный процесс (потомок), или удалить этот процесс и всех его потомков¹⁾. Если бы была выбрана первая альтернатива, то иерархия процессов могла бы легко раздробиться и разрушиться, потенциально оставляя в системе изолированные процессы без контроля за их поведением. Мы, следовательно, требуем, чтобы процедура *Destroy* удаляла процесс и *всех* его потомков. Остается открытый вопрос, что делать с ресурсами уничтоженных процессов. Простейший путь в данном случае (который мы выбираем) уничтожить все семафоры ресурсов, созданные каждым уничтожаемым процессом, и вернуть все *собственные* повторно используемые ресурсы, которые были созданы уничтожающим процессом или его предшественниками, в соответствующие им описи. Если уничтожающий процесс желает иметь подробную информацию из дескрипторов уничтожаемых процессов, он может перед *Destroy* выполнить одну или более операций *Suspend*.

Операция *Destroy* вызывается обращением *Destroy(*n*)*, где *n* — это имя „корневого“ процесса того дерева, которое должно быть удалено. Для каждого процесса *i* в этом дереве стандартная программа удаляет *i* из его списка *Sdata*, возвращает все собственные ресурсы в их списки доступности, уничтожает все семафоры, созданные *i*, и, наконец, ликвидирует дескриптор процесса. Если один из уничтожаемых процессов находился в

¹⁾ Мы не будем разрешать процессу уничтожать самого себя непосредственно; он может поручить это породившему его процессу, посыпая ему соответствующее сообщение.

состоянии развития, то вызывается планировщик процессов.

```

procedure Destroy(n);
begin
  sched := false;
  i := GetInternalName(n);
  Kill(i);
  if sched then Scheduler
end Destroy;

procedure Kill(i);
begin
  if Status[i] = 'running' then
    begin Stop(i); sched := true end;
  Remove(Sdata[i], i);
  for all s ∈ Progeny[i] do Kill(s);
  for all r ∈ {MainStore[i] ∪ Resoources[i]} do
    if owned(r) then Insert(Avail[Semaphore(r)], data(r));
  for all R ∈ CreatedResources[i] do
    RemoveResourceDescriptor(R);
  RemoveProcessDescriptor(i)
end Kill

```

Последний примитив, *ChangePriority*(*n, k*), изменяет приоритет процесса *n* на *k*. Обычно при этом также требуется, чтобы *n*, в каком бы списке он ни находился, был передвинут на другое место, отражающее новый приоритет. Если желателен перехват процессора, то при изменении приоритета вызывается планировщик. Оставляем программу *ChangePriority* в качестве упражнения.

Во всех примитивах важно, чтобы в их состав входила проверка на ошибки. Например, последние четыре операции требуют проверки, чтобы убедиться, что процесс с именем *n* является действительно „потомком“ вызывающего процесса; эта проверка могла бы быть введена в *GetInternalName*. Существует также проблема, относящаяся к дублированию внешних имен: каждый непосредственный „потомок“ каждого процесса должен иметь свое отличное от других процессов внешнее имя, а если возможна неоднозначность, то должны быть заданы имена путей как аргументы этих примитивов. Проблемы реализации критической секции и вопросы, обсуждавшиеся в разд. 3.5, также применимы к примитивам. Мы предложили для всех примитивов реализацию „занятого ожидания“ в качестве общего замка вместо возможной блокировки процесса; блокирование

может произойти только как результат запросов на семафоры ресурсов (см. след. раздел). Следовательно, требуется, чтобы эти критические секции, т. е. операции, были эффективно реализованы.

Упражнения

1. Предположим, что операции *Suspend* и *Activate* подобны *Destroy* в том, что они также применяются ко всем «потомкам» названного процесса. Допустим также, что *Suspend* и *Activate* могут называть только прямого «потомка» в качестве их процесса-аргумента. Модифицируйте *Activate* и *Suspend* так, чтобы процесс *n*, вызывающий *Activate*, активизировал только тех потомков процесса, которые были предварительно приостановлены *n*, а *Suspend* приостанавливал только те процессы, которые активны. (Указание: придумайте дополнительные типы состояний приостановленного процесса для указания источника приостановки.)

2. Напишите на алголоподобном языке программу для операции *ChangePriority(n, k)*. Считайте, что в результате этой операции соответствующий процесс вытесняется, если приоритет процесса в состоянии готовности выше, чем развивающегося процесса.

7.2.2. Примитивы ресурсов

Как подчеркивалось ранее, все взаимодействия процессов и распределение ресурсов будут происходить посредством операций над семафором ресурса. Определены четыре примитива, работающие с ресурсами:

1. *Create RS*: создать новый семафор ресурса.
2. *Destroy RS*: уничтожить семафор ресурса.
3. *Request*: запросить некоторые элементы семафора ресурса.
4. *Release*: освободить некоторые элементы семафора ресурса.

Придерживаясь описания этих операций, мы приведем несколько примеров их использования и рассмотрим эффективность их применения.

Процесс, запрашивающий ресурсы, выдает команду *Request(RS, data, ans)*, где *RS* — имя класса ресурсов (семафор ресурсов), *ans* указывает на область памяти для запоминания результатов удовлетворенного запроса, т. е. описания конкретного распределения, и *data* определяет детали запроса. Процессы могут быть заблокированы только своими собственными действиями (самоблокировки) и только через эту операцию. Блокирование происходит в том случае, если распределитель для этого семафора ресурсов не может или не будет удовлетворять запрос немедленно. Действия примитива *Request* состоят в том, чтобы занести процесс в список ожидания *Waitrs*,

связанный с RS , а затем вызвать распределитель; это приводит к тому, что распределителю требуется только проверять свою описание и список ожидания или в *Request* или в *Release*, не делая различий между этими двумя видами обращений. Распределитель возвращает список, состоящий из нуля или более процессов (разд. 7.1.2). Все процессы в этом списке, за исключением одного, вызывающего *Request*, устанавливаются в состояние *readya* или *readys* и заносятся в список готовности RL с помощью операции *Request*. Если запрос вызывающего процесса * не удовлетворяется, то процесс * устанавливается в состояние *blockeda* и процессор становится доступным. В конце *Request* всегда вызывается планировщик процессов, чтобы выявить любые новые распределения процессора; это, возможно, слишком общее решение для некоторых ситуаций, в которых запрос может быть немедленно удовлетворен, но цена реализации не слишком высока, а система в то же время обладает дополнительной гибкостью в принятии новых решений о распределении.

```

procedure Request( $RS, data, ans$ );
begin
   $r := GetInternalNameRS(RS)$ ;
  Insert(Waitrs[ $r$ ], (*,  $ans, data$ ));
  Allocator( $r, k, L$ );
  self := true;
  for  $j := 1$  step 1 until  $k$  do
    if  $L[j] \neq *$  then
      begin
         $i := L[j]$ ; Insert( $RL, i$ ); Sdata[i] :=  $RL$ ;
        Status[i] := if Status[i] = 'blockeda' then 'readya' else
                      'readys';
      end
    else self := false;
    if self then begin Status[*] := 'blockeda';
                  Sdata[*] := Waitrs[r];
                  Process[Processor[*]] :=  $\Omega$ 
                end;
  Scheduler
end Request

```

Когда распределитель приписывает ресурсы ожидающему процессу, он запоминает информацию о распределении в ans ; если ресурс повторно используемый, то информация о распределении запоминается также в списке ресурсов дескриптора процес-

сов. Информация в области *ans* необходима для того, чтобы за-
прашивающий процесс мог идентифицировать ресурс, тогда как
информация в списке необходима для распределения ресурсов,
удаления процесса и ведения списка.

Если процессу больше не требуется ранее полученный по-
вторно используемый ресурс или если он хочет добавить еди-
ницы потребляемого ресурса к описи, он делает это посредством
примитива *Release*. Вызов кодируется в форме *Release(RS, data)*, где *RS* — семафор ресурса, а *data* определяет характе-
ристики свободного ресурса; *data* обычно содержит тот же тип
информации, что и параметр *ans* в *Request*. *Release* добавляет
data к описи ресурса и вызывает распределитель, который пы-
тается удовлетворить самые приоритетные запросы блокиро-
ванных процессов. Если распределения выполнены, то пробуж-
даются соответствующие процессы и вызывается планировщик.

```

procedure Release(RS, data);
begin
  r := GetInternalNameRS(RS);
  Insert(Avail[r], data);
  Allocator(r, k, L);
  for j := 1 step 1 until k do
    begin
      i := L[j]; Insert(RL, i); Sdata[i] := RL;
      Status[i] := if Status[i] = 'blockeds' then 'readys'
                    else 'readya'
    end;
    if k ≠ 0 then Scheduler
  end Release

```

Таким образом, состояние процесса может быть изменено в ре-
зультате выполнения операций *Request/Release* или любых при-
митивов процесса. На рис. 7.4 представлены в совокупности все
возможные изменения состояний процесса и операции, вызы-
вающие эти изменения.

Процедуры *CreateRS* и *DestroyRS* имеют дело с динами-
чески создаваемыми и удаляемыми дескрипторами ресурсов в
системе. (Мы неявно предполагаем, что для всех дескрипторов
процессов и ресурсов существует область памяти.) Большинство
аппаратурных ресурсов „создаются“ при генерации системы и
во время ее загрузки, тогда как другие ресурсы создаются ди-
намически системными и пользовательскими процессами. Про-
цедура *CreateRS* требует инициализации и определения как
описи, так и списков ожидания, спецификации стандартных про-
грамм занесения и удаления для этих списков, а также опреде-

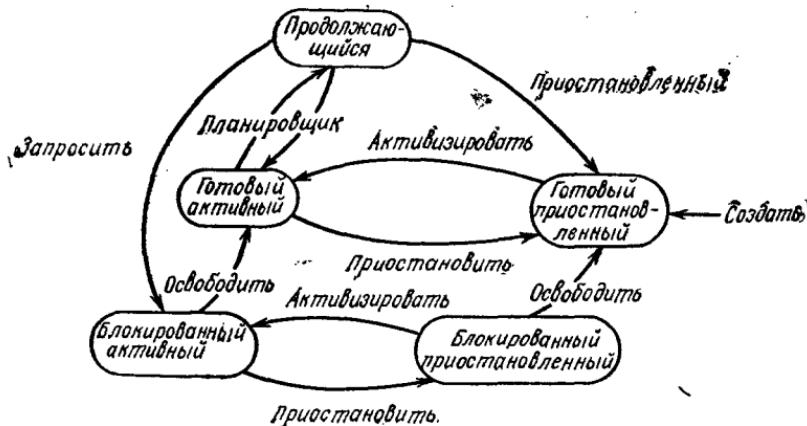


Рис. 7.4. Изменения состояний процесса.

ления распределителя для ресурса. Вызов кодируется следующим образом:

CreateRS(RS, sr, avail, waitrs, allocator, otherinfo);

где

RS есть внешнее имя семафора ресурса;

sr показывает, является ли *n* повторно используемым;

avail и *waitrs* — указатели на головные элементы соответственно описи и очереди ожидания (табл. 7.3) с требуемыми установками в начальное состояние, принятыми для этих очередей;

allocator — точка входа в стандартную программу-распределитель;

otherinfo указывает на произвольный список дополнительных полей в дескрипторе.

Мы опустим подробности и только заметим, что появляется также запись в ресурсной части дескриптора процесса-отца, которая идентифицирует процесс как создателя ресурса.

Как и в случае процесса ресурс может быть уничтожен только его процессом-отцом или предком процесса-отца. Обращение к уничтожающему примитиву имеет вид: *DestroyRS*, где *RS* — есть внешнее имя семафора ресурса. Процедура *DestroyRS* удаляет дескриптор ресурса из системы, пробуждает (изменяет состояние процесса „блокированный“ на состояние „готовый“ и включает в список *RL*) с выдачей соответствующего сообщения все процессы, ожидающие ресурс.

Процедуры *Request* и *Release* могут управлять всеми ресурсами и ситуациями синхронизации, которые мы перечислили.

Наиболее изменяемая часть их внутренней структуры заложена в стандартную программу распределителя. Следующие примеры иллюстрируют некоторые общие применения примитивов:

1. Семафоры Дейкстры

Процедуры *Request* и *Release* могут быть сделаны изоморфными по отношению к *P*- и *V*-операциям. Для семафора *S* соответствующие обращения определяются как *Request*(*S*, Ω , Ω) и *Release*(*S*, Ω), где Ω указывает на неопределенные параметры. Опись есть значение *S*; программы *Insert* и *Remove* для списка доступности просто инкрементируют и декрементируют соответственно значение *S*. Распределитель определяет, возможно ли распределение путем сравнения длины списка ожидания с *S*. *P* и *V* могут быть обобщены для запроса и освобождения любого положительного числа устройств, т. е. *Request*(*S*, n , Ω), и *Release*(*S*, n), $n \geq 1$; в этом случае распределитель получается более сложным (см. упражнение).

2. Прохождение сообщения

Пусть справедливо предположение, что производитель сообщений посылает их конкретным именованным процессам, а потребители сообщений (получатели) могут запрашивать сообщения от известных процессов. Тогда сообщение *m* может быть передано процессу *p* с помощью обращения

Release(*M*,(*p*, *m*));

где *M* — семафор ресурса сообщений. Получатель запрашивает сообщение от процесса *p*, выдавая

Request(*M*, *p*, *m*),

где *m* — область данных, которая будет содержать сообщение.

Распределитель направляет сообщения получателям. Так же легко могут быть реализованы другие возможности, например при которых получатель не определяет посылающий процесс.

3. Распределение основной памяти

Для распределения и освобождения непрерывных блоков переменной длины в основной памяти вызовы могут кодироваться следующим образом:

Request(*MAINSTORE*, *NUMBEROFWORDS*, *BASEADDRESS*);

и

Release(*MAINSTORE*,(*BASEADDRESS*, *NUMBEROFWORDS*));

Структура распределителя и данных может быть одной из таких, какие рассматривались в гл. 5.

4. Буферы ввода и вывода

Пусть *IFB*, *OFB* и *EM* — семафоры ресурсов, связанные соответственно с заполненными при вводе, заполненными для вывода, и пустыми буферами в буфериом пуле. Операции для *IFB* могут кодироваться, например, так:

Request(IFB, Ω, current);

и

Release(IFB, current);

где *current* — это индекс буфера в пуле, используемый как выходной параметр для *Request* и входной параметр для *Release*.

При таком основном наборе примитивов можно исследовать вопросы эффективности и реализации. Чтобы сохранить наше соглашение о том, что процесс может логически блокироваться только посредством примитива *Request*, операции защищаются общим замком типа „занятое ожидание“ — такой же замок используется для всех примитивов, которые обращаются к системным дескрипторам. Помимо того что критические секции должны быть короткими, требуется, с целью обеспечения корректности и эффективности, наличие централизованного контроля над распределителями и стандартными программами манипуляции со списками. Такой контроль может быть осуществлен ведением системной библиотеки отложенных стандартных программ и структур данных, к которым могут обращаться пользователи, когда создаются ресурсы. Вейдерман (1971) сравнил реализации операций, подобных *Request* и *Release* на языке высокого уровня и на Ассемблере, а также общих распределителей и структур данных с операциями передачи сообщений Бринч Хансена и примитивами IBM OS/360 (разд. 3.6); величины времени выполнения имели один и тот же порядок. В этом разделе вновь опущены детали обнаружения и анализа ошибок. Соглашения о доступе и структуре дерева процессов значительно упрощают эту проблему. Однако обычно требуются более расширенные тесты, чтобы обосновать и усилить ограничения на запросы ресурса; например, было бы желательно во многих случаях ограничить объем ресурса (например, памяти), который может запрашивать конкретный процесс или даже запрещать определенным процессам доступ к некоторым ресурсам. (Мы коснемся этих вопросов в следующем разделе.)

Наши последние замечания относятся к некритическим и отложенным запросам. Процесс может заказать более одного элемента ресурса, если ресурс доступен, но в случае недоступности ресурса он может продолжить обработку; аналогично, процесс может в принципе выдать запрос, который не нуждается в не-

медленном удовлетворении. Первый случай может быть выражен парой $(0, n)$ в поле данных запроса; тогда распределитель проектируется так, чтобы распределять нуль элементов (что всегда возможно), если n элементов недоступны. Простейшей реализацией второго типа запроса является посылка сообщений другому процессу, который запрашивает ресурс обычным способом и выдает ответ исходному процессу, когда ресурс распределен.

Упражнение

- Реализуйте примитивы P и V с помощью операций *Request* и *Release* соответственно; в частности, определите структуры данных соответствующих списков, программы *Insert* и *Remove* и распределитель.
- Повторите (а) для обобщенных примитивов P и V , описанных в конце первого примера этого раздела.

7.2.3. Полномочия процесса

Чтобы защитить процесс от вредного воздействия на самого себя, а также защитить другие процессы и системные ресурсы, требуются некоторые механизмы защиты. Они могут быть сформулированы как *полномочия* процесса — действия, которые процесс может или не может производить над собой и другими объектами в системе (см. также разд. 5.3 о защите реальной и виртуальной памяти).

Ядро, описанное в предыдущих разделах, включает элементарную схему полномочий в виде дерева процессов. Процесс может только приостанавливать (*Suspend*), активизировать (*Activate*), изменять приоритет (*ChangePriority*) и уничтожать (*Destroy*) только своих прямых или удаленных потомков. Аналогично, процесс может передать только подмножество своих ресурсов своему потомку и может разрушить, используя *DestroyRS*, только те семафоры ресурсов, которые создали он или его потомки. Мы не накладываем каких-либо ограничений на использование *Request* и *Release*, но, очевидно, наши другие соглашения о процессах распространяются и на этот случай (разрешение для процесса вызвать эти примитивы только через семафоры ресурсов, созданные им или его предшественниками). К сожалению, эта стратегия обычно наделяет наибольшими полномочиями выполнять примитивы *Request/Release* процессы самого нижнего уровня в иерархии дерева, тогда как полномочия в других областях больше для процессов, находящихся на высоких уровнях. Более последовательная и мощная схема полномочий получается, если явно определять, какие ресурсы могут быть запрошены и освобождены каждым процессом. Мы можем легко включить эту возможность в наше ядро добавлением

нового поля, *Capability*[*i*], к дескриптору для каждого процесса *i*. В поле *Capability*[*i*] перечисляются те семафоры ресурсов, которые процесс *i* может запросить или освободить согласно следующим правилам:

1. Создатель семафора ресурса автоматически получает полномочие выполнять *Request/Release* для этого ресурса.
2. При создании потомка процесс может разделять любое подмножество его полномочий с новым процессом.

Этот метод защиты может быть полностью реализован в программном обеспечении. Более развитые полномочия, такие, как описанные в конце разд. 5.3, требуют специальной аппаратуры для практической реализации.

Когда происходит нарушение полномочий, должно быть предпринято некоторое действие. Можно было бы просто разрушить ошибочный процесс (и всех его потомков), но это средство кажется чрезмерным. Более удовлетворительный подход заключается в том, чтобы передать управление стандартной программе обработки ошибок или процессу, порожденному или текущим процессом, или операционной системой; тогда ошибка оказывает такое же действие, как и внутреннее прерывание или программное прерывание. (Обработка прерываний обсуждается в следующем разделе.) Второй приемлемой возможностью является возврат индикатора ошибки прямо в процесс; ошибочный процесс может затем предпринять любое действие, которое он пожелает. Эта последняя стратегия была выбрана для одной из реализаций ядра (Шоу и др., 1970) прежде всего из-за своей простоты.

7.3. Прерывания и процессы ввода-вывода

В предыдущих разделах рассматривалось управление процессами и ресурсами для внутренних процессов ЦОУ. Целью настоящего раздела является изучение процессов, связанных с операциями ввода-вывода, и показ того, как аппаратурные прерывания (особенно те, которые порождаются вводом-выводом и таймерами) могут быть согласованы со структурами процессов и ресурсов.

Наша стратегия состоит в том, чтобы определить для каждого устройства ввода-вывода *D*, т. е. для каждого устройства периферийной вспомогательной памяти, процесс ввода-вывода *rd*, который управляет работой этого устройства. Процесс *rd* получает запросы на ввод-вывод, вызывает соответствующие распределители каналов и контроллеров, инициирует операции ввода-вывода, осуществляет передачи сообщений об ошибках, расшифровывает прерывания и посылает сообщения о завершении запрашивающему процессу. Самая низкоуровневая „коман-

да" ввода-вывода состоит из последовательности сообщений — ответов, передаваемых по отношению к выбранному p_D , и может принимать общую форму:

```
Release(Dio, ioprog);
Request!(Dioend, Ω, completionmessage);
```

Dio — это семафор ресурса, соответствующий сообщениям, направляемым к p_D , тогда как семафор $Dioend$ связан с ответами от p_D . Параметр $ioprog$ определяет природу запроса — чтение, запись, управление или некоторую их последовательность. Параметр $ioprog$ может непосредственно содержать команду канала или приказы контроллеру, возможно, с параметризованными адресами канала или контроллера, если они могут быть переменными; или же этот параметр может соответствовать более высокому уровню управления: например, он может содержать только адреса устройства памяти и число единиц, которые нужно передать. Обычно для каждого устройства и системы устанавливается различный набор соглашений относительно $ioprog$. Результаты выполнения запроса ввода-вывода обычно возвращаются в $completionmessage$; это поле содержит, как правило, число единиц информации, переданных в действительности, и сообщения о том, произошли или нет какие-либо ошибки во время операции.

Между устройством и памятью может быть более одного физического „пути“ в зависимости от аппаратурных связей (рис. 7.5)¹⁾. В общем случае, изображенном на рис. 7.5(с), чтобы установить такой путь, должны быть заданы канал и контроллер. Процесс p_D будет, конечно, изменяться для каждого устройства, но типичная организация могла бы быть такой:

```
 $p_D:$  begin
    L: Request!(Dio, Ω, (p, ioprog));
        Request(PATN, D, path);
        Generate IO instructions from path and ioprog;
        Initiate IO;
        Request(IOint, path, message);
        Decode interrupt; Possible further IO operations;
        Produce completion message;
        Release(PATH, path);
        Release(Dioend, (p, completionmessage));
        go to L
end
```

¹⁾ Некоторые мультипроцессорные системы имеют также канальные контроллеры, позволяющие соединять процессоры с различными каналами.

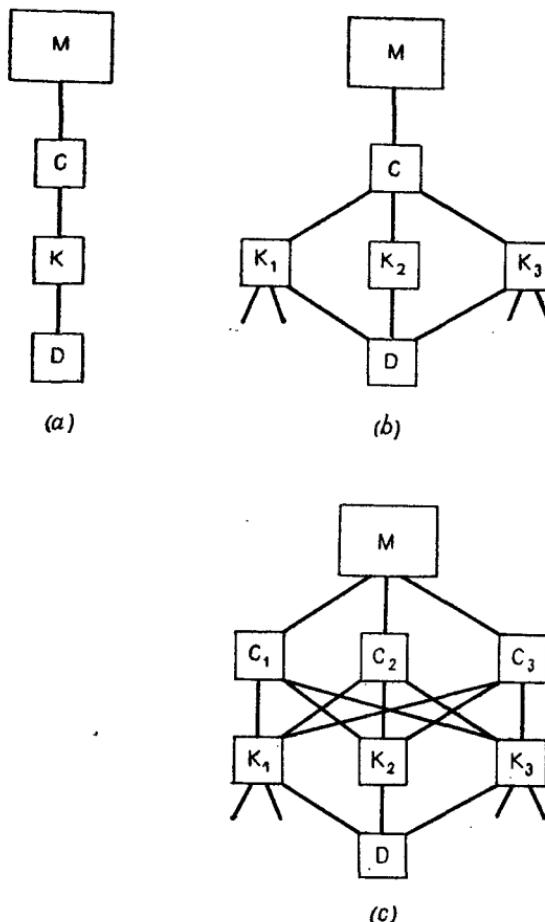


Рис. 7.5. Пути из памяти к устройству ввода-вывода D: (а) — единственный путь; (б) — единственный канал; (с) — множество путей.

M: основная память,

C: канал,

K: контроллер,

D: устройство.

Здесь *p* обозначает процесс, который выдал запрос на ввод-вывод через *Dio*. Распределитель ресурсов для *PATH* назначает канал и контроллер процессу *p_D*, таким образом определяя для операции физический путь передачи. „Дальнейшие операции ввода-вывода“ (*further IO operations*) могут быть необходимы потому, что начальный запрос генерирует несколько последовательностей ввода-вывода (например, за поиском на диске следует чтение), или потому, что возможны определенные

ошибки ввода-вывода, когда система пытается несколько раз правильно выполнить команды ввода-вывода, прежде чем посыпает сигнал оператору машины. Параметр $IOint$ связан с прерываниями ввода-вывода. Приведенный выше алгоритм предполагает, что только одно прерывание происходит в результате каждого инициирования ввода-вывода. Это справедливо в некоторых случаях, но существуют и другие общие возможности. В общем случае сигнал прерывания может возникать в результате завершения *каждого* действия канала, контроллера и устройства. Эти три сигнала могут быть выработаны, например, при выполнении операции перемотки магнитной ленты; для периферийных карточных и печатающих устройств часторабатываются два сигнала: один — при завершении работы канала и другой — в конце работы контроллера или устройства. Вместо выдачи запроса и освобождения ресурса $PATH$ (пути) могут быть выданы индивидуальные команды *Request* и *Release* для каждого компонента пути, т. е. канала и контроллера.

В этой модели обработчики прерываний рассматриваются с одинаковых позиций. Их основной функцией является преобразование аппаратурных сигналов в программные сообщения. Абстрактно мы рассматриваем каждый обработчик прерываний IH_i как циклический процесс, который ожидает прерывания, определяет внутренний процесс ps , ответственный за обслуживание этого прерывания, и посредством *Release* выдает соответствующее сообщение для ps . Чтобы осуществить это, обработчик сначала сохраняет состояние прерванного процесса $*$, изменяет его состояние с *running* на *readya* и устанавливает элемент дескриптора процессора *Process* [*Processor*[*]] в Ω :

```
 $IH_i : SaveStatein(CPUstate[*]); Status[*] := 'readya';
Process[Processor[*]] := \Omega;
Determine service process ps;
Release(Interrupt Class, (ps, interruptdetails));$ 
```

Когда *планировщик* вызывается внутри *Release* в IH_i , он может или реактивизировать прерванный процесс, или отдать ЦОУ процессу, состояние которого в результате выполнения *Release* изменилось с *blocked* на *readya*. Во время работы IH_i запрещены прерывания, так как модифицируются системные дескрипторы.

Для прерываний ввода-вывода класс прерываний *IOint* может быть семафором ресурса, а ps обычно соответствует процессу устройства pr , ответственному за инициирование ввода-вывода. Внутренние прерывания, обусловленные программными ошибками, такими, как арифметическое переполнение или нарушение адресации памяти, обычно вызывают посылку сообщения соответствующему процессу обработки ошибок, который

может проигнорировать ошибку, прекратить процесс или, возможно, исправить ошибку после расшифровки сообщения об ошибке. Таймерные прерывания могут привести, например, к завершению процесса, если он превысил общее выделенное ему время, помещению процесса в конец списка готовности, если истек его квант времени в системе разделения времени, или пробуждению некоторого процесса, если процесс запросил сигнал разбудить его в некоторый определенный момент времени.

Пример

Предположим, что при таймерном прерывании исполняющийся процесс p должен отказаться от ЦОУ в пользу процесса q с более высоким приоритетом, таким, что $\text{Priority}[q] \geq \text{Priority}[p]$, если такой q существует. Операция *Release* в обработчике таймерных прерываний могла бы тогда упроститься следующим образом:

*Release(TIMEOUT, (p_T, *));*

Процесс таймера p_T , связанный с этим прерыванием, может быть представлен в форме:

p_T : **begin**

L: Request(TIMEOUT, Ω, p);

ChangePriority(p, Priority[p]);

go to L

end

Предполагается, что p_T имеет более высокий приоритет, чем p , для того чтобы p_T быстро получал ЦОУ. Операция *ChangePriority* помещает p в конец списка готовых процессов по приоритету $\text{Priority}[p]$. Процесс p_T мог бы достигнуть того же самого эффекта без использования *ChangePriority* с помощью приостановки (посредством *Suspend*) развивающегося процесса и активизации (посредством *Activate*) процесса в состоянии готовности с наивысшим приоритетом. Процесс p_T , по существу, является в этом примере планировщиком процессов, приводимым в действие таймером. Другие примеры использования таймеров и их отношения к планирующим механизмам рассматриваются в следующем разделе.

7.4. Организация планировщиков процессов

Планировщик процессов, или диспетчер, представляет собой часть операционной системы, отвечающую за распределение обрабатывающих устройств (процессоров) процессам, находящим-

ся в состоянии готовности. В этом разделе рассматривается прежде всего *организация* планировщиков, а не частные *стратегии* диспетчирования. Часто бывает желательно, например, применять одни стратегии для процессов, связанных с пакетными заданиями, а другие — для процессов, генерируемых интерактивными пользователями, одни — для системных процессов, а другие — для пользовательских, одни — для процессов, имеющих дело непосредственно с вводом-выводом, а другие — для чисто внутренних действий. С помощью общего механизма приоритетов и аккуратного использования таймеров может быть спроектирована единая система, реализующая широкий спектр таких стратегий.

Как было сказано ранее, в общей мультипроцессорной МС существует ряд ситуаций, когда должен вызываться планировщик. Новое распределение процессора в принципе возможно, когда какой-либо процесс блокируется, пробуждается (т. е. становится готовым к развитию) или изменяет свой приоритет (в приоритетной системе); любое изменение числа процессоров обычно также приводит к вызову диспетчера. Одним из особо важных источников сигналов блокировки и пробуждения, делающим возможным жесткое управление распределением, является *интервальный таймер*, который будет вырабатывать сигнал прерывания после произвольного, заданного пользователем временного интервала.

Существует необходимость установить пределы времени непрерывного развития каждого процесса — для того чтобы предохранить систему от монополизации некоторыми процессами, гарантировать пользователям приемлемое время ответа, реагировать на зависящие от времени события и производить восстановление при некоторых типах программных ошибок. С этой целью диспетчер может устанавливать таймер непосредственно перед передачей управления выбранному процессу; напротив, процесс r_t , получающий сигнал в результате таймерного прерывания, можно считать ответственным за установку таймера. В обоих случаях главным является вопрос: „На какое значение t должен быть установлен таймер?“ Использование r_t возможно, если t — это константа для всех процессов, находящихся в состоянии готовности, например в простой системе разделения времени с одинаковым для всех квантом. Однако более гибким решением будет динамически присваивать различные кванты различным классам процессов, отражая, например, их важность, их поведение в ближайшем прошлом и текущую загрузку системы. Это может быть выполнено связыванием интервала времени Δt_p с каждым процессом p в списке готовности (Зальцер, 1966); когда планировщик выбирает процесс p из списка готовности, он может затем установить таймер на

интервал Δt_p . Сам планировщик может оставаться при этом простым и не иметь дела с вычислением и изменением этих временных интервалов. Эти задачи может выполнять системный процесс, который пробуждается в определенные моменты времени. Для управления такими зависящими от времени действиями удобна некоторая форма списка фиксированных моментов (Лэмпсон, 1968), аналогичная в принципе спискам, используемым в программах для моделирования.

Существуют другие случаи, в которых процесс будет самоблокироваться запросом „разбудить“ его в некоторое определенное время в будущем. Одним из примеров является случай в области измерения систем, когда желательно периодически измерять характеристики загрузки МС. Другой случай встречается при разделении времени, когда для пользователя, работающего в линию, в пределах данного времени гарантируется некоторый тип связи, возможно, тривиальной. На многих больших коммерческих системах прикладные программы часто выполняются периодически, например программа (процесс) может быть запущена (разбужен) один раз в день, чтобы произвести инвентаризацию или суммарный отчет о продаже. Эти виды зависящих от времени задач могут управляться ведением списка процессов, блокированных по ресурсу „время“ и упорядоченных по времени пробуждения. В рамках нашего ядра может быть определен ресурс, называемый *TIME*. Процесс может запросить сигнал пробуждения в некоторое время в будущем с помощью обращения

Request(TIME, t, Q);

где t есть интервал времени, или, возможно, абсолютное время. Высокоприоритетный временной процесс может пробуждаться периодически¹⁾ для выполнения операции

Release (TIME, currenttime);

Распределитель ресурса *TIME* будет управлять списком фиксированных времен, т. е. списком блокированных процессов, ожидающих *TIME*, и пробуждать любые ожидающие процессы при условии *wakeuptime < currenttime*.

Сказанное иллюстрирует тесную взаимосвязь между планированием и ресурсом времени. Это, конечно, является справедливым также для случая других критических ресурсов, в особенности основной памяти. Некоторые страницные системы с вызовом страниц по требованию, например, устанавливают ограничения на число процессов, которые можно выбрать для

¹⁾ Это может оказаться удобным сделать, когда происходит прерывание по таймеру, когда загружается новое задание или в общем случае, когда происходит некоторое событие, о котором известно, что оно происходит регулярно с приемлемой частотой.

выполнения, или на число страниц, одновременно присутствующих в очереди для ввода-вывода, или на количество пространства памяти для рабочего множества (Александер, 1970). Эту задачу обычно выполняют супервизорные процессы, которые контролируют текущее распределение ресурсов и представленные требования.

7.4.1. Ведущие и разделяемые планировщики

Планировщик процессов может разделяться процессами в системе подобно ядру, рассмотренному в этой главе, — т. е. планировщик вызывается путем обращения к подпрограмме, что является косвенным результатом операции ядра. Ядро и планировщик тогда потенциально содержатся в адресном пространстве всех процессов и выполняются в составе любого процесса. Вторым методом является централизация планировщика (и возможно, ядра). Концептуально этот тип планировщика считается отдельным процессом, развивающимся на своем собственном процессоре; он может непрерывно проверять заявки системы на проведение работ или может приводиться в действие сигналами пробуждения. Альтернативы ведущего и разделяемого планировщика иллюстрируются на рис. 7.6,

Организация ведущего была успешно использована в мультипроцессорных системах, где один процессор постоянно предназначен для планирования „ядерных“ или других супервизорных действий, таких, как спуллинг и загрузка. Операционная система тогда строго отделена от заданий пользователей, по крайней мере в принципе. Эта схема применяется в ОС CDC 6400/6600 SCOPE (CDC, 1971) и до некоторой степени в системах ASP¹⁾, „присоединенной поддержки“ IBM/360. Даже в том случае, если единственный процессор не предназначен для выполнения части операционной системы, принцип ведущего все же применим в одно- и мультипроцессорных конфигурациях. Однако в этих ситуациях возникает логическая трудность, а именно: кто диспетчирует процесс планировщика? Одним из решений является передача управления планировщику всякий раз, когда процесс блокируется или пробуждается; причем это будет планировщик более высокого уровня, чем остальные процессы.

Лэмпсон (1968) предложил другой способ реализации принципа ведущего, который кажется особенно привлекательным при современной технологии. Небольшой быстрый процессор предназначен для микропрограммного планировщика, который непрерывно опрашивает определенное число линий ввода на наличие сигналов от других процессоров и от внешних источников

¹⁾ ASP — сокращение от associative support processor (присоединенный вспомогательный процессор). — Прим. перев.

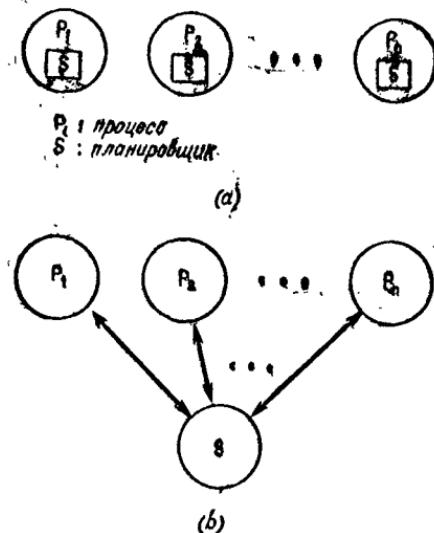


Рис. 7.6 (a) — разделяемый планировщик; (b) — ведущий планировщик.

ввода-вывода. Каждый сигнал обычно представляет собой сообщение о пробуждении, сопровождаемое именем и приоритетом процесса, который необходимо разбудить. Функциями планировщика являются изменение состояния процесса на „головой“ (если он в настоящее время не находится в состоянии готовности или не развивается), занесение его в список готовности и передача процессору сигнала „переключить“, если существует развивающийся процесс более низкого приоритета.

Принцип разделения был выбран в нашей модели ядра по следующим причинам. Так как компоненты ядра автоматически включаются в каждый процесс, они не зависят от числа доступных процессоров и не нуждаются в постоянной привязке к некоторому из них. К такой разделенной организации естественно приводит точка зрения, состоящая в том, что ядро действительно определяет множество машинных операций, необходимых для преобразования стандартной ЭВМ в виртуальную ЭВМ с функциями ОС. Точка зрения разработчиков системы MULTICS, которые применяют этот метод, состоит в том, что процессы могут в принципе иметь различные типы операционных систем, управляющих ими; например, могут одновременно существовать разнообразные планировщики, причем каждый из них привязан к различному множеству процессов. Когда программа ядра является чистой процедурой и единственной разделяемой копией, разница между разделяемой организацией и ведущей с неспециализированными процессорами является скорее разницей точек зрения и в меньшей степени разницей стратегий реализации.

7.4.2. Приоритетное планирование

Сначала будет представлен общий и относительно сложный планировщик. Имея его в виду, мы рассмотрим ряд упрощений, которые могут быть сделаны в общих практических ситуациях.

Предположим, что планировщик, модуль или подпрограмма, называемая *Scheduler*, вызывается только в примитивах ядра этой главы, т. е. в *Suspend*, *Activate*, *Destroy*, *ChangePriority*, *Request* и *Release*. Список готовности *RL* построен на приоритетном базисе, как на рис. 7.3, и содержит все процессы *p*, такие, что $Status[p] \in \{running, readya, readyd\}$. Используется дисциплина диспетчеризации с вытеснением. Она предполагает, что планировщик гарантирует, что в любой момент приоритет любого процесса, находящегося в состоянии *running*, больше или равен приоритету любого процесса, находящегося в состоянии *readya*. На рис. 7.7 показан общий вид такого планировщика и его отношение к структурам системных данных и другим примитивам.

Могут быть кратко определены задачи планировщика. Пусть в любой момент S_r есть множество процессов, находящихся в состоянии *running*, и S_{ar} есть множество процессов в системе, находящихся в состоянии *readya* и *running*. Определим S_a как

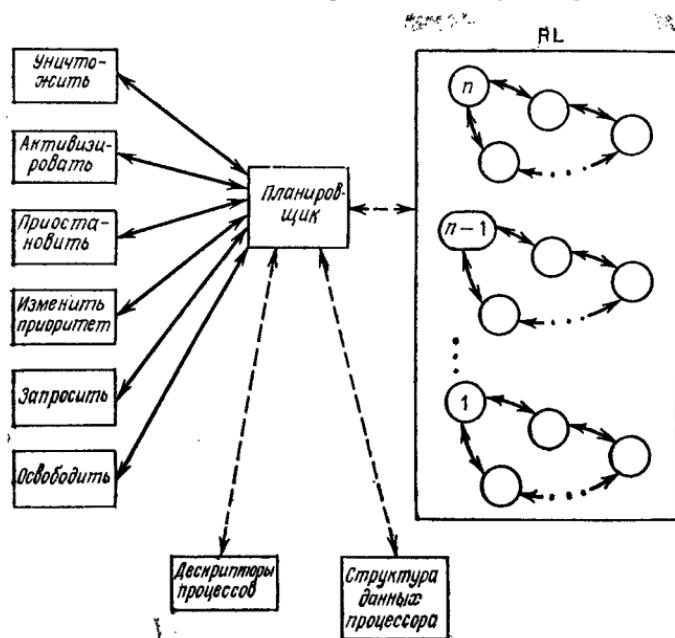


Рис. 7.7. Взаимодействие планировщика со структурами системных данных и примитивами ядра.

множество процессов, которые должны развиваться в данное время (находиться в состоянии *running*). S_a есть наибольшее подмножество S_{ar} со свойствами:

$|S_a| = np$ (число процессоров)¹⁾,

и для всех $p \in S_a$ и $q \in (S_{ar} - S_a)$, $p \neq q$, выполняется:

1. $\text{Priority}[p] \geq \text{Priority}[q]$.

2. Если $\text{Priority}[p] = \text{Priority}[q]$, то p предшествует q в RL .

Планировщик гарантирует, что $S_a = S_r$ путем распределения всех членов множества процессов $S_a - S_a \cap S_r$ процессорам, а именно перехваченным процессорам всех членов множества процессов $S_r - S_r \cap S_a$ и неактивным процессорам, когда $|S_r| < np$.

Реализация, однако, требует некоторой осторожности. Одним из источников трудностей является процесс, обозначаемый *, внутри которого вызывается планировщик. Если процесс * блокируется — в результате выдачи *Request* — или если он вытесняется со своего процессора, то необходимо отсрочить передачу управления некоторому другому процессу, пока не будут выполнены все другие распределения; в противном случае планировщик не сможет завершить последний, так как он развивается на процессоре процесса *. Когда процессор c , включая *Processor*[*], перераспределяется, состояние c должно быть сохранено в компоненте дескриптора процесса *CPUstate*[*]. В случае * это можно считать небольшой хитростью: счетчик команд, записанный в *CPUstate*[*], должен указывать на команду, которая позволяет процессу * соответствующим образом возвратиться из *Scheduler* (и, следовательно, выйти из ядра, так как *Scheduler* вызывается только в конце примитива) при его последующем возобновлении. Мы выполняем переключение с процесса p на процесс q на процессоре c с помощью процедуры

```
procedure Switch(p, q, c);
begin
    Interrupt(c);
    StoreState(c, CPUstate[p]);
    if p == * then StoreInstrCntr(RA, CPUstate[p]);
    LoadState(c, CPUstate[q])
end Switch
```

¹⁾ Согласно разд. 7.1.2, существует «пустой» процесс для каждого процессора; приоритет пустого процесса меньше, чем у любого «работающего» процесса.

Операция *Interrupt(c)* прерывает процессор *c*, если *Processor[*] ≠ c*; в противном случае она рассматривается как пустая операция ("по-оп"). *RA* есть адрес возврата процесса, вызывающего *Switch*. Кроме передачи управления новому процессу, *LoadState* может также устанавливать интервальный таймер. Особенности процедуры *Switch* будут, конечно, сильно зависеть от характеристик конкретной ЭВМ.

Чтобы упростить описание манипуляций с *RL* в процедуре *Scheduler*, прямой указатель каждой записи процесса *p* также будет обозначаться *fqp[p]*. Предполагается, что номера приоритетов *n, n - 1, ..., 1* отличаются от номеров процессов, чтобы не возникло путаницы. Универсальный планировщик приводится ниже. Общая последовательность действий такова:

1. Взять процесс *p*, находящийся в состоянии *readya*, с наивысшим приоритетом.
2. Если существуют неактивные процессоры, распределить *p* одному из них и перейти к шагу 1.
3. Попытаться вытеснить развивающийся процесс. Если попытка оказалась успешной, перейти к шагу 1.

Специальная обработка для процесса * выполняется в конце алгоритма.

```

procedure Scheduler;
begin
  p := Π := n; cpu := 1;
  comment Найти процесс „readya“ с наивысшим приоритетом;
  Nextreadya;
  b := true;
  while b ∧ Π ≠ 0 do
    begin
      p := fqp[p];
      if p = Π then Π := Π - 1
      else b := Status[p] ≠ 'readya'
    end;
    if Π = 0 then go to Wrapup *;
  comment Найти неактивный процессор для p;
  while cpu ≤ np do
    if Process[cpu] ≠ Ω then cpu := cpu + 1
    else
      begin
        Process[cpu] := p; Processor[p] := cpu;

```

```

Status[p] := 'running';
if cpu ≠ Processor[*] then Loadstate(cpu, CPUstate[p])
else p* := p;
cpu := cpu + 1;
go to Nextreadya
end;

```

comment Все процессы распределены. Попытайтесь добиться вытеснения. Существует ли развивающийся процесс с priority < Priority[*p*];

```

Πmin := Π;
for c := 1 step 1 until np do
begin
  q := Process[c];
  if Priority[q] < Πmin then
    begin Πmin := Priority[q]; cp := c end
  end;
  if Π ≠ Πmin then
    begin
      comment Выполнить вытеснение (preemption) на процессоре cp;
      q := Process[cp]; Status[q] := 'readya';
      Processor[p] := cp; Process[cp] := p; Status[p] := 'running';
      if q = * then p* := p else Switch(p, q, cp);
      go to Nextreadya
    end;
  comment Окончательное фиксирование вызывающего процесса *;
  Wrapup *:
  if Status[*] ≠ 'running' then Switch (*, p*, Processor [*])
end Scheduler

```

Планировщик, оставаясь управляемым, может тем не менее быть значительно упрощен, если ослаблены некоторые основополагающие предположения. Рассмотрим сначала устранение вытеснения. Планировщик в этом случае не нужно вызывать по операции *ChangePriority*, и около половины программы (и времени) устраивается. Самая общая ситуация, когда доступен только один центральный процессор (*np* = 1), приводит к значительно менее сложному алгоритму планирования. В этом случае *Suspend* никогда не вызывает *Scheduler* и может произойти самое большое одно переключение процесса. Другой возмож-

ностью является ограничение планировщиков, так что в любое время из списка блокировки (пробуждения) выбирается не более чем один процесс. Широко используются также различные комбинации приведенных выше способов.

Упражнения

1. Составьте полный планировщик для каждого случая, когда на систему накладываются следующие ограничения:
 - а) процессы никогда непосредственно не вытесняются со своих процессоров;
 - (b) $pr = 1$ (однопроцессорная система);
 - (c) операция *Release* пробуждает не более одного процесса;
 - (d) объединяются ограничения (a), (b), (c).
2. Рассмотрите систему с отдельным процессором, предназначенным целиком для распределения других процессоров. Пусть планировщик циклически опрашивает другие процессоры о наличии у них работы. Предположим, что планировщик вызывается косвенно через наши примитивы ядра. Однако вместо вызова подпрограммы сначала устанавливается с помощью примитива специальный процессорный регистр, скажем R_i для i -го процессора, и затем, по отдельной операции, цикл занятости на R_i выполняется до тех пор, пока регистр не сбрасывается планировщиком. Приведите алгоритм для общего приоритетного планировщика с вытеснением в этом типе вычислительной системы.

7.5. Методы планирования

В рамках нашего ядра представлен ряд общих стратегий планирования. После обсуждения вопроса о вытеснении и двух основных методов планирования рассматриваются критерии статического и динамического определения приоритетов.

Вытеснение, невытеснение и выборочное вытеснение

В последнем разделе определен планировщик однородного вытеснения, основанный на приоритетах процесса. Вытеснение в этом методе представляет собой один из способов гарантировать, что „важные“ процессы, например процессы, ответственные за прием и рассылку сигналов реального времени, — быстро получают процессор, когда они активны и готовы к развитию. (Менее общий подход, применимый к высокоприоритетным процессам, которые приводятся в действие прерываниями, состоит в том, чтобы неявно инициировать вытеснение через аппаратурные прерывания с помощью прямой передачи управления процессу с более высоким приоритетом, связанному с прерыванием.) Часто, однако, может быть приемлемой более экономичная дисциплина невытеснения. Стоимость вытеснения включает время на переключение процессов и затраты на дополнительную логику в планировщике; вытеснение может также

привести к другим потерям эффективности, таким, как потери на свопинг данных или программ во вспомогательную память. Между этими двумя крайностями лежит другой набор возможностей, который мы назовем *выборочное вытеснение*.

Одна из таких „выборочных“ дисциплин приписывает каждому процессу p битовую пару (u_p, v_p) со следующим значением:

$$u_p = \begin{cases} 1, & \text{если } p \text{ может вытеснить другой процесс;} \\ 0 & \text{в противном случае.} \end{cases}$$

$$v_p = \begin{cases} 1, & \text{если } p \text{ может быть вытеснен некоторым другим} \\ & \text{процессом;} \\ 0 & \text{в противном случае.} \end{cases}$$

Таким образом, u_p определяет полномочия процесса p в отношении вытеснения, и v_p в действительности есть текущий приоритет. Пара (u_p, v_p) является обычно еще одним компонентом вектора состояния процесса, используемого планировщиком. Для увеличения степени общности приоритет процесса p может быть заменен на пару (Π_p, ρ_p) , где Π_p представляет приоритет процесса p , находящегося в состоянии готовности, и ρ_p есть приоритет процесса p , когда он развивается. Тогда можно осуществлять некоторое управление степенью вытеснения одного процесса относительно другого. Наконец, универсальная и, вероятно, непрактичная схема требует матрицы вытеснения с элементами (i, j) , показывающими, может ли нет процесс i получить предпочтение перед процессом j . Тем не менее многие из вышеупомянутых эффектов могут быть получены с помощью разумного использования операций *Suspend* и *ChangePriority* в высокоприоритетных „управляющих“ процессах в совокупности с механизмами вытеснения, представленными ранее.

Дисциплины планирования типа FIFO и Round-Robin

Дисциплина типа FIFO диспетчирует процессы в соответствии с их временем поступления в список готовности RL — чем раньше поступление, тем выше приоритет при планировании. Это легко может быть реализовано в нашем ядре с помощью установления для всех процессов одного и того же приоритета. Стандартная программа *Insert* для RL просто помещает процесс-аргумент в конец очереди с единственным приоритетом. Несмотря на то что все множество процессов в МС редко рассматривается на основе дисциплины FIFO, она является, конечно, наиболее общей (а при отсутствии другой информации — рациональной) дисциплиной для управления очередью готовности в пределах каждого приоритета.

Важным вариантом метода FIFO является дисциплина типа *round-robin*¹⁾, при которой процессы диспетчируются в соответствии с правилом FIFO, но для величины *непрерывного* процессорного времени, которое может быть использовано любым процессом, устанавливается фиксированный *квант времени* t . Если данный процесс развивается в течение t непрерывных единиц времени, он вытесняется со своего процессора и помещается в конец RL , причем процессор распределяется следующему (первому) процессу в RL снова не больше чем на t единиц времени. Интервал времени t может быть постоянным в течение длительного периода времени или может меняться по мере загрузки процессов. Если время T для одного полного цикла всех активных процессов поддерживается постоянным, то t может быть вычислено динамически каждые T секунд с помощью выражения $t = T/n$, где n есть число активных процессов в RL . Пример в конце разд. 7.3 реализует дисциплину типа *round-robin(RR)*, если все процессы, за исключением p , имеют один и тот же приоритет. Методы, основанные на схеме *RR*, используются большинством систем с разделением времени для управления „интерактивными“ процессами, т. е. процессами, которые прямо или косвенно связаны с интерактивными терминалами.

Обычные дисциплины планирования в средних и больших МС включают использование множества различных приоритетов (также называемых приоритетными уровнями). Как и когда выбираются эти приоритеты — это наиболее важные вопросы, относящиеся к дисциплинам планирования.

Статические приоритеты

При использовании статического приоритета его значение остается постоянным и равным тому, которое приписано процессу в момент его создания. Для определения значений приоритетов применялись самые разнообразные критерии и методы. Одним из широко распространенных методов, который используется в пакетных мультипрограммных системах, является *внешняя генерация* значения приоритета. Задания пользователей помещаются на заданный приоритетный уровень, и процессам, последовательно порождаемым каждым заданием, приписываются приоритеты задания. Обычно с каждым уровнем связана стоимость, основанная на алгоритме учета ресурсов; пользователь выбирает тот приоритет для своего задания, который дает ему удовлетворительное обслуживание (ответ) за цену, которую он может заплатить. Например, задание с приорите-

¹⁾ В отечественной литературе используются термины «циклический алгоритм», «круговой циклический алгоритм». — Прим. ред.

том Π может быть оценен $f_1(\Pi)$ долларов/с за процессорное время, $f_2(\Pi)$ долларов/слово/с за основную память, $f_3(\Pi)$ долларов/карта за ввод и $f_4(\Pi)$ долларов/строка за вывод. Системным процессам приписываются фиксированные приоритеты разработчиками и (или) администрацией вычислительного центра.

Более удовлетворительные результаты получаются, если статический приоритет основан на характеристиках заданий пользователей и процессов, включая, возможно, внешний приоритет; эти характеристики объявляются пользователем. Далее следуют два примера дисциплин приоритетного планирования.

1. Время выполнения

Принцип, принятый во многих системах, состоит в том, что чем короче задание, тем лучше должно быть его обслуживание. Мы приведем два аргумента в пользу этого принципа. Первый состоит в том, что пользователи не будут удовлетворены, если они не получат быстрого обслуживания коротких заданий; в противном случае, если возможно, они обратятся за услугами в другое место¹⁾. Второй аргумент вытекает из теории. Если каждое задание обрабатывается последовательно до завершения (без мультипрограммирования), среднее время выполнения задания минимизируется при диспетчеризации заданий в порядке „самое короткое задание — первым“ (см. упражнения). Таким образом, есть основание надеяться, что в более сложной мультипрограммной среде, включающей интерактивные процессы, среднее время прохождения задания будет минимизировано при предоставлении самым коротким заданиям наивысшего приоритета.

2. Тип процессов

Задание или процесс классифицируется в зависимости от его относительных требований к вводу-выводу и ЦОУ и от настоящей необходимости получения быстрого ответа, например пакетное или интерактивное задание, фоновый или основной процесс в системе реального времени, задание с ограниченными вычислениями или с ограниченным вводом-выводом. Часто наивысшие приоритеты (в отношении ЦОУ) приписываются процессам с повышенными требованиями к вводу-выводу; в то же самое время ввод-вывод, затребованный процессами с ограниченным вводом-выводом, получает высокий приоритет. Эта дисциплина обеспечивает хорошее время ответа для интерактив-

¹⁾ В универсалах давно это поняли, введя специальные кассы для одной-двух покупок.

ных пользователей и пользователей в реальном времени, а также поддерживает высокую степень параллелизма между вводом-выводом и центральной обработкой.

Динамические приоритеты

Как время ответа, так и производительность ЭВМ могут регулироваться более точно, когда приоритет процесса может изменяться во время его существования; эта черта особенно ценна в системах разделения времени. Критерии для выбора новых приоритетов аналогичны критериям, используемым в случае статического приоритета, но более часто базируются на измерениях поведения процесса, а не на *априорных* декларациях. Большинство динамических схем спроектированы так, чтобы помещать интерактивные процессы и процессы с ограниченными вычислениями в начало приоритетных очередей и чтобы разрешить процессам с ограниченным вводом-выводом перемещаться на более низкие приоритетные уровни. В пределах каждого уровня процессы обслуживаются согласно дисциплине FIFO или RR. Степень активности ввода-вывода может быть измерена непосредственным подсчетом числа запросов от процесса операций ввода-вывода на протяжении данного периода времени. И наоборот, приоритет будет прямо отражать частоту ввода-вывода, если приоритетный уровень процесса автоматически увеличивается после запроса ввода-вывода (при условии, конечно, что приоритет уже не достаточно высокий).

Почти классическим примером динамического метода этого типа является многоуровневая схема, использованная первоначально в системе разделения времени CTSS (Корбато и др., 1962). Каждому приоритетному уровню Π соответствует время T_Π , которое является максимальным количеством процессорного времени, которое любой процесс может получить на этом уровне; если процесс превышает T_Π , его приоритет уменьшается до $\Pi - 1$. Удовлетворительным правилом определения T_n (в основном используемым в CTSS) является $T_n = 2^{\Pi-n} \cdot t_n$, где t_n — максимальное время в очереди с наивысшим приоритетом; t_n кратно основному кванту времени. Многие существующие системы используют подобный алгоритм. Как правило, процесс помещается на самый высокий уровень, когда он разбужен при завершении ввода-вывода.

Существует много других вариантов. Данное множество процессов может быть ограничено в некотором диапазоне приоритетов. Процессу может также автоматически приписываться более высокий приоритет, если он ожидал в списке *RL* без получения обслуживания в течение большого интервала времени,

Более подробные рассмотрения методов планирования могут быть найдены у Коффмана и Клейнрока (1968) и Александера (1970).

Упражнение

Для данного набора независимых последовательных процессов $\{p_1, \dots, p_n\}$ с соответствующими временами выполнения $\{t_1, \dots, t_n\}$ докажите, что *среднее время существования* процесса минимизируется при выполнении их последовательно в порядке «самый короткий процесс — первым», т. е. для всех $i, j, i \neq j$ p_i выполняется раньше p_j , если $t_i < t_j$. Предположите, что только один процессор доступен и что процессы не взаимодействуют.

8. Проблема тупиков

Понятие тупика было неформально введено в примере об объединении буферов в разд. 2.4.4; неограниченная конкуренция за получение свободных буферов приводит к нежелательному состоянию, при котором все процессы, способные освободить пустые буфера, заблокированы по одному и тому же ресурсу. Когда некоторые процессы заблокированы в результате таких запросов на ресурсы, которые *никогда* не могут быть удовлетворены, если не будут предприняты *чрезвычайные* системные меры, то о таких процессах говорят, что они *зашли в тупик*. Такая ситуация обычно возникает потому, что поставщики требуемых ресурсов сами блокируются по тем же самым ресурсам. „Чрезвычайные“ системные меры могут заключаться в принудительном изъятии ресурсов или ликвидации процессов.

Почему изучение тупиков имеет важное значение? Эта проблема, конечно, рассматривалась при проектировании многих существующих систем, но значительное внимание ей было удалено только в нескольких системах. Тупиками мало интересовались потому, что в большинстве МС средства параллельного программирования и взаимодействия процессов имели дело только с системными процессами (и то в незначительной степени) и потому, что процессам пользователей ресурсы распределялись статически во время создания. Однако в будущем ожидается, что динамическое распределение ресурсов, параллельное программирование и взаимодействующие процессы будут рабочими характеристиками многих средних и больших систем на всех уровнях системных и пользовательских программ. Будут соответственно расти вероятности возникновения тупиков. Проблема может иметь также первостепенное значение в ряде систем реального времени, таких, как управление ракетами с помощью ЭВМ или управление и контроль систем жизнеобеспечения, например, во время хирургических операций. Существуют и другие причины, почему мы должны подробно изучить тупики. Работа в этой области представляет один из немногих удачных примеров теоретического и практического изучения некоторых сторон операционных систем и дает представление о возможном подходе к теории операционных систем. Наконец, при изучении тупиков пытаются частично найти ответ на один из самых важных вопросов в вычислительной технике: „Сможет ли процесс развиваться до завершения?“

В этой главе содержится достаточно формальное рассмотрение проблемы тупиков, основанное прежде всего на исследований и модели Холта (1971b, 1972)¹). Далее приводятся некоторые примеры и определения, обсуждаются способы распознавания тупиков, их предотвращения и восстановления по отношению как к стандартным ресурсам (повторно используемым), так и ресурсам типа сообщений (потребляемым).

8.1. Примеры тупиков в вычислительных системах

Далее представлены примеры различных типов тупиков, которые могут возникнуть с повторно используемыми (SR) и потребляемыми (CR) ресурсами. Цель заключается в том, чтобы проиллюстрировать разнообразие ситуаций, в которых возможны тупики, и обосновать формальную модель.

1. Разделение файла

Предположим, что как процесс p_1 , так и p_2 модифицируют файл D во время обновления и требуют рабочую ленту. Пусть T — единственное лентопротяжное устройство, имеющееся для рабочих лент. Предположим далее, что процесс p_2 непосредственно перед модификацией нуждается в рабочей ленте по некоторой другой причине, например, чтобы организовать входные данные для модификации. Пусть D и T — ресурсы типа SR, позволяющие модифицировать файл и использовать лентопротяжное устройство соответственно. Тогда процессы p_1 и p_2 могут иметь следующий вид:

$p_1:$	\vdots	$p_2:$	\vdots
$Request(D);$ ²		$Request(T);$	
$r_1: Request(T);$			\vdots
\vdots		$r_2: Request(D);$	
\vdots			\vdots
$Release(T);$		$Release(D);$	
$Release(D);$		$Release(T);$	
\vdots		\vdots	

¹) Некоторые начальные сведения содержатся в статье Коффмана и др. (1971). Дейкстры (1965a, 1968b) принадлежит одно из первых наиболее серьезных исследований в этой области.

²) Процесс блокируется при выполнении операции *Request* (Запрос) до тех пор, пока не будет распределен ресурс по операции *Release* (Освобождение) ресурс возвращается в систему.

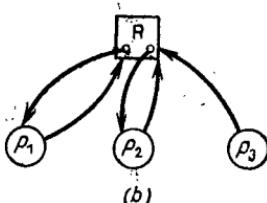
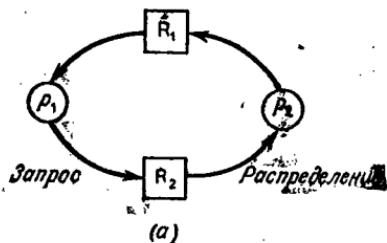


Рис. 8.1. Тупик с ресурсами типа SR:
(а) — простой пример тупика; (б) — разделение единственного ресурса.

Процессы p_1 и p_2 могут достичь меток r_1 и r_2 соответственно в одно и то же „время“; например, строго последовательно p_2 сначала получает T , затем p_1 получает D , затем p_1 доходит до r_1 , и, наконец, p_2 достигает r_2 . Сразу же после этого момента два наших процесса войдут в тупик. Процесс p_1 будет блокироваться по T , сохраняя за собой D , тогда как p_2 блокируется по D , сохраняя T ; p_1 не может продолжаться, если не продолжает p_2 , и наоборот; процессы сомкнулись в „смертельном объятии“. Эта тупиковая ситуация графически показана на рисунке 8.1(а). Квадратами обозначены ресурсы, а кружки представляют процессы. Стрелка (направленная дуга) от ресурса к процессу указывает на распределение, тогда как стрелка от процесса к ресурсу есть запрос. Подставьте D для R_1 и T для R_2 . (Следует заметить, что этот пример был бы даже более реалистичным, если бы T был другим файлом данных; p_1 и p_2 тогда запрашивают одни и те же файлы данных, но в обратном порядке).

2. Разделение единственного ресурса

Единственный ресурс R типа SR, такой, как основная или вспомогательная память, содержит m единиц, достаточных для распределения, и разделяется n процессами p_1, \dots, p_n , $2 \leq m \leq n$. Пусть каждый процесс использует элементы ресурса R в последовательности

$\text{Request}(R); \text{Request}(R); \text{Release}(R); \text{Release}(R);$

где каждая из операций Request и Release действуют на единицу ресурса R . Когда распределены все единицы ресурса,

может легко возникнуть тупик; все процессы, держащие по единице ресурса R , могут навсегда заблокироваться на второй операции *Request*, тогда как некоторые процессы могут аналогично заблокироваться на первой операции *Request*. Такое состояние системы показано на рис. 8.1(b) для случая $n = 3$ и $m = 2$. Это относительно общий тип тупика. Он может возникнуть, например, в подсистеме спулинга, в которой несколько вводных и выводных процессов конкурируют за получение пространства вспомогательной памяти; если пространство становится полностью распределенным (a) входным файлам заданий, ожидающих загрузки, и (b) выходным записям частично выполненных заданий, то система попадает в тупик.

3. Статическое распределение ресурсов для заданий и шагов заданий

Рассмотрим пакетную МС, которая распределяет память статически (во время создания процесса) для процессов, соответствующих заданиям и шагам заданий. Предположим, что двухшаговое задание J_1 требует файл данных D для всего задания, магнитную ленту T_1 для своего первого шага и ленту T_2 для последнего шага; T_1 может быть 7-дорожечной, а T_2 — 9-дорожечной лентой. Пусть существует также одношаговое задание J_2 , которое использует для своего выполнения как D , так и T_2 . Считается, что ресурсы D , T_1 и T_2 являются ресурсами типа SR. Процессы планировщика заданий p_1 и p_2 , сопоставленные заданиям J_1 и J_2 , могут оказаться следующими:

p_1	$Request(T_1);$	$p_2:$	$Request(T_2);$
	$Request(D);$		$Request(D);$
	$Execute Step1;$		$Execute Step1;$
	$Release(T_1);$		$Release(T_2);$
$r_1:$	$Request(T_2);$		$Release(D);$
	$Execute Step2;$		\vdots
	$Release(T_2);$		
	$Release(D);$		
	\vdots		

Если процессы p_1 и p_2 достигают меток r_1 и r_2 в одно и то же „время“, то снова возникает тупиковая ситуация, изображенная на рис. 8.1(a). (Следует подставить D вместо R_1 и T_1 вместо R_2 .) Таким образом, распределение ресурсов в произвольном порядке, например в порядке, в котором они определяются в управляющих картах задания, может быть очень опасным. Первоначальная дисциплина распределения в системе IBM

OS/360 так же не ограничивала порядок следования запросов до тех пор, пока эти опасности не были обнаружены (Хавендер, 1968).

4. Ресурсы типа SR и CR

Вычислительный процесс p_c сообщается с процессом ввода-вывода p_{io} . Предположим, что время развития p_c обратно пропорционально размеру пространства памяти, которое он может приобретать динамически; например, p_c может быть задачей обработки сложного списка. Следовательно, p_c запрашивает у системы, например, с помощью операции „ $\text{Request}(M, \text{all})$;” все доступные блоки памяти. А процесс ввода-вывода время от времени требует для целей буферизации один блок памяти, который он запрашивает посредством операции „ $\text{Request}(M)$;”. Эти два процесса p_c и p_{io} могут быть синхронизированы следующим образом:

p_c :	$\text{Request}(M, \text{all});$	p_{io} :	$\text{Request}(\text{IOrequest});$
	$\text{Release}(\text{IOrequest});$.
r_c :	$\text{Request}(\text{IOcompletion});$	r_{io} :	$\text{Request}(M);$
	.		.
	.		.
	$\text{Release}(M, \text{all});$		$\text{Perform } IO;$
			$\text{Release}(\text{IOcompletion});$
			.
			.

Эти два процесса будут *всегда* попадать в тупик при выполнении операции Request с метками r_c и r_{io} . Вычислительный процесс p_c не может получить сигнал „завершение ввода-вывода“ до тех пор, пока p_{io} не получит свой буфер, а система не может выделить память по запросу от p_{io} , пока p_c не получит уведомление „завершение ввода-вывода“ и не освободит память M . Это тупиковое состояние показано на рис. 8.2(а); для ресурсов типа CR (IOcompletion и IOrequest) стрелка направлена от ресурса к производителю. (Производитель „владеет“ неограниченным количеством единиц ресурса.)

5. Обмен сообщениями

Пусть процесс p_1 вырабатывает сообщения S_1 , p_2 вырабатывает сообщения S_2 и p_3 вырабатывает сообщения S_3 ; S_1 , S_2 и S_3 — это ресурсы типа CR. Предположим, что p_1 получает сообщения от p_3 , p_2 от p_1 и p_3 от p_2 . Если связь с помощью этих

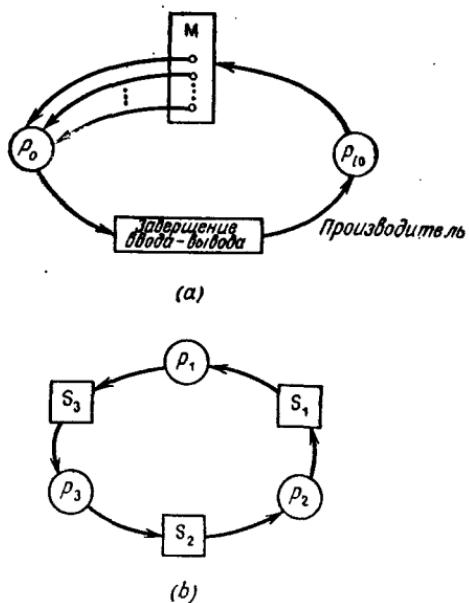


Рис. 8.2. Тупик с ресурсами типа CR:
(a) — разделение ресурсов типа SR
и CR; (b) — прохождение сообще-
ния.

сообщений со стороны каждого процесса устанавливается в следующем порядке:

$p_i: \dots \text{Release}(S_i); \text{Request}(S_j); \dots$

(где для $i = 1, 2, 3$ $j = 3, 1, 2$ соответственно), то никаких трудностей не возникает. Однако перестановка этих двух операций вызывает тупик (рис. 8.2(b)):

$p_1: \dots \text{Request}(S_3); \text{Release}(S_1); \dots$

$p_2: \dots \text{Request}(S_1); \text{Release}(S_2); \dots$

$p_3: \dots \text{Request}(S_2); \text{Release}(S_3); \dots$

6. Крайний случай в языке PL/1

Холт (1971b) приводит следующую программу для иллюстрации того, как пользователь легко может завести процесс в тупик:

```
REVENGE: PROCEDURE OPTIONS(MAIN, TASK);
    WAIT(EVENT);
    END REVENGE;
```

„WAIT(EVENT)” эквивалентно „Request(EVENT)” в нашей нотации. Теоретически $EVENT$ никогда не происходит и $REVENGE$ навсегда блокируется.

7. Полезный тупик (Холт, 1971а)

Предположим, что система отводит 200К основной памяти заданиям пользователей и что задание потребует или 100К или 200К памяти для своего выполнения. Допустим, что с самого начала загружены два 100К-байтовых задания, и очередь заданий *всегда* содержит некоторые 100К-байтовые задания. Тогда, если новое задание выбирается для загрузки сразу же после завершения какого-то задания и в качестве критерия выборки используется доступность памяти, 200К-байтовое задание никогда не будет выполняться. Это можно предотвратить без „чрезвычайных мер“ с помощью задержки распределения памяти для 100К-байтовых заданий, если 200К-байтовое задание уже ожидало в течение достаточно долгого времени. Такой вид ситуации, в которой планировщик или распределитель может препятствовать процессу продолжаться до завершения, даже если это логически возможно сделать, был назван Холтом „полезным тупиком“. Полезный тупик может также встретиться, например, в системе, которая присваивает наивысший приоритет самым коротким заданиям и поддерживает длинные (с большим объемом вычислений) задания в состоянии ожидания до тех пор, пока не завершатся все короткие задания. Этот тип тупика мы будем рассматривать в оставшейся части главы.

Как показано на примерах, проблема тупика предполагает взаимодействие потенциально многих процессов и ресурсов; поэтому она должна рассматриваться как глобальная системная проблема. Проблема тупика может быть разделена на три основные части:

1. *Распознавание*. Каким образом можно определить при данной системе процессов, разделяющих ресурсы и взаимодействующих друг с другом, вошло ли некоторое множество из них в тупик? Одним из ключей к решению этой проблемы, полученной из рассмотрения первых пяти примеров, является существование в графах типа „процесс — ресурс“ цикла (направленного пути от некоторой вершины к ней самой).

2. *Восстановление*. Каковы „наилучшие“ способы выхода из тупика? „Чрезвычайные“ системные меры необходимы, но мы хотим как можно меньше вторгаться в ход выполнения работ.

3. *Предотвращение*. Главный вопрос — как можно избежать тупика? Входной спулер в разд. 3.4.3 содержал простой аппарат для предотвращения тупика в случае одного частного ресурса. Изменение порядка операций *Request* и *Release* в примерах 1 и 5 делает тупик невозможным. Нас интересуют некоторые общие методы предотвращения. Примеры от 1 до 6 состоят из простой последовательной программы без условных пере-

дов; следовательно, существует фиксированная последовательность операции с ресурсами для каждого процесса, которая может быть легко определена путем пробы. Однако в общем (и вполне реальном) случае, лучшее, что можно сделать, — это пронумеровать все „пути“ в программе; один из этих путей может привести к тупику в комбинации с другими процессами, но для произвольного процесса невозможно знать *aприори*, будет ли выбран когда-либо в действительности этот частный путь. [Это замаскированная проблема остановки (например, см. Мински, 1967).]

Интересно заметить, что примеры тупиков можно найти не только в вычислительных, но и в других системах, таких, например, как транспортные и экологические системы. Причем те же самые три проблемы (предотвращение, распознавание и восстановление) присутствуют в любых системах.

Упражнение

Перечислите некоторые примеры тупика в других типах систем, отличных от вычислительных.

8.2. Модель системы

Для модели, рассматриваемой в этой главе, состояние операционной системы будет сводиться к состоянию различных ресурсов в системе (свободны они или распределены). Состояние системы изменяется процессами, когда они запрашивают, приобретают или освобождают ресурсы; это будут единственно возможные действия процессов. Если процесс не блокирован в данном состоянии системы, он может изменить это состояние на новое. Однако, так как в общем случае невозможно (это неразрешимая проблема) знать *aприори*, какой путь может избрать произвольный процесс в своей программе, то новое состояние может быть любым из конечного числа возможных; следовательно, процессы моделируются как недетерминированные объекты. Введенные понятия приводят к следующим формальным определениям:

1. *Система* есть пара $\langle \sigma, \pi \rangle$, где σ — множество состояний системы $\{S, T, U, V, \dots\}$ и π — множество процессов $\{p_1, p_2, \dots\}$.

2. *Процесс* p_i есть частичная функция, отображающая состояния системы в непустые подмножества ее же состояний; это обозначается так:

$$p_i: \sigma \rightarrow \{\sigma\}$$

Если p_i определен на состоянии S , то область значений p_i обозначается как $p_i(S)$. Если $T \in p_i(S)$, то мы говорим, что p_i мо-

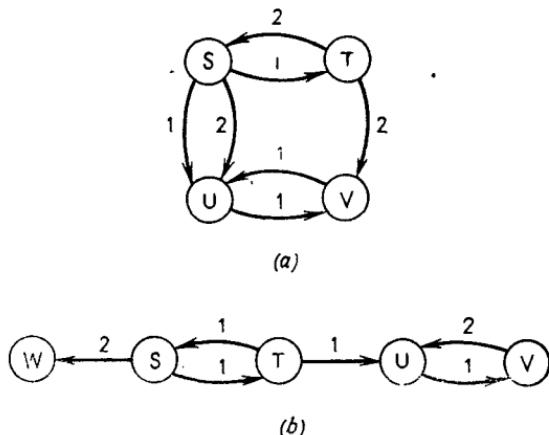


Рис. 8.3. Примеры систем: (а) — система с тупиковыми состояниями; (б) — система с тупиковым и безопасным состояниями.

жет изменить состояние из S в T посредством *операции*, и используем обозначение $S \xrightarrow{i} T$. Наконец, нотация $S \xrightarrow{*} W$ обозначает, что

- (а) $S = W$ или
- (б) $S \xrightarrow{i} W$ для некоторого p_i или
- (с) $S \xrightarrow{i} T$ для некоторого p_i и T , и $T \xrightarrow{*} W$.

Другими словами, система может быть переведена посредством $n \geq 0$ операций с помощью $m \geq 0$ различных процессов из состояния S в состояние W .

Пример

Определим систему $\langle \sigma, \pi \rangle$: $\sigma = \{S, T, U, V\}$, $\pi = \{\rho_1, \rho_2\}$, $\rho_1(S) = \{T, U\}$, $\rho_1(T) = \Omega$, $\rho_1(U) = \{V\}$, $\rho_2(S) = \{U\}$, $\rho_2(T) = \{S, V\}$, $\rho_2(U) = \Omega$, $\rho_2(V) = \{U\}$, $\rho_2(V) = \Omega$, где Ω обозначает неопределенность. На рис. 8.3(а) показана эта система.

Некоторые последовательности изменений состояния таковы:

$$S \xrightarrow{1} U, T \xrightarrow{2} V, S \xrightarrow{*} V \text{ (например, } S \xrightarrow{1} T \xrightarrow{2} V \text{ или } S \xrightarrow{2} U \xrightarrow{1} V).$$

Процесс заблокирован в данном состоянии, если он не может изменить состояние, т. е. в этом состоянии процесс не может ни затребовать, ни получать, ни освобождать ресурсы.

3. Процесс p_i заблокирован в состоянии S , если не существует ни одного T , такого, что $S \xrightarrow{i} T$.

Процесс находится в тупике в данном состоянии S , если он заблокирован в состоянии S и если вне зависимости от того, какие операции (изменения состояний) произойдут в будущем, процесс остается заблокированным.

4. Процесс p_i находится в тупике в состоянии S , если для всех T , таких, что $S \xrightarrow{*} T$, p_i блокирован в T .

На рис. 8.3(а) p_2 находится в тупике как в U , так и в V , тогда как p_1 блокирован, но не в тупике в T ; для последнего случая $T \xrightarrow{2} V$ или $T \xrightarrow{2} S$, и p_1 не становится заблокированным ни в V ни в S .

5. Состояние называется *тупиковым*, если существует процесс p_i , находящийся в тупике в S . Тупик предотвращается, по определению, при введении ограничения на систему, такого, чтобы каждое возможное состояние не было тупиковым состоянием.

6. Состояние S есть *безопасное состояние*, если для всех T , таких, что $S \xrightarrow{*} T$, T не является тупиковым состоянием.

Система на рис. 8.3(б), где $\sigma = \{S, T, U, V, W\}$ и $\Pi = \{p_1, p_2\}$, имеет как тупиковые, так и безопасные состояния: U и V безопасны; S , T и W не являются безопасными (почему?); и W есть состояние тупика.

Пример

Рассмотрим первый пример последнего раздела („Разделение файла“) и предположим, что p_1 и p_2 — циклические процессы. Если мы последовательно просматрим программы процессов p_1 и p_2 , то можно легко определить следующие состояния *процессов*:

Состояния для p_1	Состояния для p_2
0. Не держит никаких ресурсов	0. Не держит никаких ресурсов
1. Запросил D , не держит никаких ресурсов	1. Запросил T , не держит никаких ресурсов
2. Держит D	2. Держит T
3. Запросил T , держит D	3. Запросил D , держит T
4. Держит T , держит D	4. Держит D , держит T
5. Держит D (после освобождения T)	5. Держит T (после освобождения D)

Пусть состояние системы S_{ij} таково, что p_1 находится в состоянии i и p_2 находится в состоянии j . Тогда система может

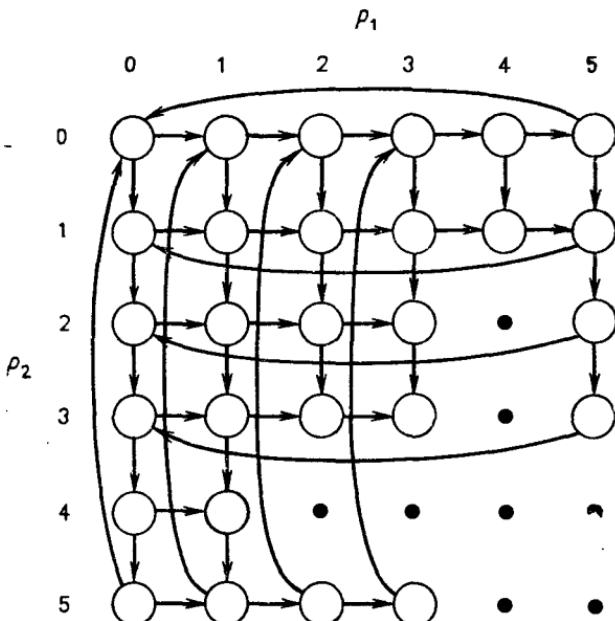


Рис. 8.4. Изменения состояния системы в примере разделения файла.

быть представлена диаграммой, изображенной на рис. 8.4. Изменения состояний (операции) процессом 1 показаны горизонтальными стрелками, тогда как изменения состояний процессом p_2 — вертикальными. Процессы p_1 и p_2 находятся в тупике в $S_{3,3}$; p_1 заблокирован, но не находится в тупике в $S_{1,4}$, $S_{3,2}$ и $S_{3,5}$; p_2 заблокирован, но не в тупике в $S_{2,3}$, $S_{4,1}$ и $S_{5,3}$.

На этой модели системы мы изучим компоненты состояния системы, операции изменения состояния и проблему тупика, когда ресурсы только типа SR.

Упражнения

1. Используя определения этого раздела, докажите, что из утверждения « S не есть состояние тупика» не следует утверждение « S есть безопасное состояние». Безопасность и тупик не являются дополняющими друг друга понятиями.

2. (Холт, 1971а). Пусть два процесса p_1 и p_2 разделяют ресурс R , содержащий две идентичные единицы. Каждый процесс может быть в одном из следующих состояний:

0. Не держит никаких ресурсов.
1. Запросил 1 единицу R , не держит ресурсов.
2. Держит 1 единицу R .
3. Держит 1 единицу R , запросил 1 единицу ресурса.
4. Держит 2 единицы.

Каждый процесс может перейти из состояния 0 в 1 по запросу, из 1 в 2 в результате приобретения ресурса R , из 2 в 3 по запросу, из 3 в 4 в результате приобретения, из 4 в 2, освобождая единицу, и из 2 в 0, освобождая единицу; это единственны возможные изменения состояния. Пусть состояние системы есть S_i , где i — состояние p_1 и j — состояние p_2 . Постройте диаграмму состояний для этой системы, аналогичную изображенной на рис. 8.4. В каких состояниях, если такие существуют, система находится в тупике и(или) заблокирована?

8.3. Тупик в случае повторно используемых ресурсов

Повторно используемый ресурс (SR) есть конечное множество идентичных единиц со следующими свойствами:

1. Число единиц ресурса постоянно.
2. Каждая единица ресурса или доступна или распределена одному и только одному процессу (разделение отсутствует).
3. Процесс может освободить единицу ресурса (сделать ее доступной), только если он ранее получил эту единицу.

Данное определение выделяет существенные характеристики обычных ресурсов, необходимые нам для изучения тупика; в частности, нет необходимости явно идентифицировать каждую единицу такого ресурса. Примерами ресурсов типа SR являются компоненты аппаратуры, такие, как основная память, вспомогательная память, периферийные устройства и, возможно, процессоры, а также программное обеспечение, такое, как файлы данных, таблицы и „разрешение войти в критическую секцию“.

8.3.1. Графы повторно используемых ресурсов

Состояние операционной системы представляется графом типа „процесс — ресурс“, называемым в случае ресурсов типа SR графом *повторно используемых ресурсов*. *Направленный граф* определяется как пара $\langle N, E \rangle$, где N есть множество вершин и E есть множество упорядоченных пар (a, b) , $a, b \in N$, называемых *ребрами*. Граф повторно используемых ресурсов есть направленный граф со следующей интерпретацией и ограничениями.

1. Множество N разделено на два взаимно исключающих подмножества, множество вершин для представления процессов $\pi = \{p_1, p_2, \dots, p_n\}$ и множество вершин для представления ресурсов $\rho = \{R_1, R_2, \dots, R_m\}$. Каждая вершина для ресурса $R_i \in \rho$ обозначает ресурс типа SR.

2. Граф является „двудольным“ по отношению к π и ρ . Каждое ребро $e \in E$ соединяет вершину π с вершиной ρ . Если $e = (p_i, R_j)$, то e есть ребро запроса и интерпретируется как запрос от процесса p_i на единицу ресурса R_j . Если $e = (R_j, p_i)$,

то e есть ребро назначения и выражает назначение единицы ресурса R_i процессу p_i .

3. Для каждого ресурса $R_i \in \rho$ существует неотрицательное целое t_i , обозначающее количество единиц ресурса R_i .

4. Пусть $|(a, b)|$ — число ребер, направленных от вершины a к вершине b . Тогда система должна всегда работать при следующих ограничениях:

а) Может быть сделано не более чем t_i назначений (распределений) для R_i , т. е. $\sum_j |(R_i, p_j)| \leq t_i$ для всех i .

б) Сумма запросов и распределений относительно любого процесса для конкретного ресурса не может превышать количества доступных единиц, т. е. $|(R_i, p_i)| + |(p_i, R_i)| \leq t_i$ для всех i и j .

Примеры

1. На рис. 8.1(а) последнего раздела изображен граф повторно используемых ресурсов.

$$\rho = \{R_1, R_2\}; \pi = \{p_1, p_2\}; N = \rho \cup \pi; t_1 = t_2 = 1;$$

$$E = \{(p_1, R_2), (R_2, p_2), (p_2, R_1), (R_1, p_1)\}.$$

2. Пример 2 последнего раздела представлен графом на рис. 8.1(б).

$$\rho = \{R\}; t = 2; \pi = \{p_1, p_2, p_3\};$$

$$E = \{(p_i, R) | i = 1, 2, 3\} \cup \{(R, p_1), (R, p_2)\}.$$

Состояние системы изменяется на новое только в результате запросов, освобождений или приобретений ресурсов одним процессом.

1. *Запрос*. Если система находится в состоянии S и процесс p_i не выдал запросов (нет ребер запроса), то p_i может запросить любое число ресурсов, указывая при этом количество единиц конкретного ресурса в пределах указанного выше ограничения 4. Тогда система попадает в состояние T , например $(S \xrightarrow{i} T)$. T отличается от S только дополнительными ребрами запроса от p_i к затребованным ресурсам.

2. *Приобретение*. Система может изменить состояние S на состояние T в результате операции *приобретения* процесса p_i (т. е. определить $S \xrightarrow{i} T$) тогда и только тогда, когда p_i выдал запросы и все такие запросы могут быть удовлетворены; т. е. для всех ресурсов R_j , таких, что $(p_i, R_j) \in E$, мы имеем

$$|(p_i, R_j)| + \sum_k |(R_j, p_k)| \leq t_j.$$

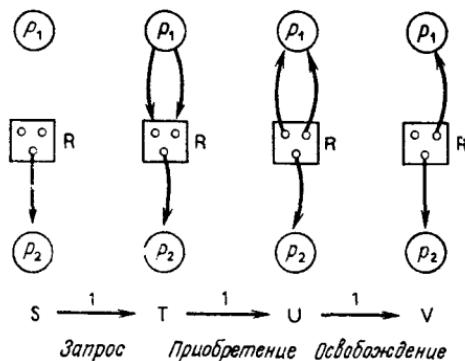


Рис. 8.5. Операции процесса.

Граф T идентичен S , за исключением того, что все ребра запроса (p_i, R_j) для p_i обратны ребрам (R_j, p_i) , что отражает произведенное распределение ресурсов.

3. *Освобождение*. Процесс p_i может вызвать переход из состояния S в состояние T с помощью освобождения (*release*) тогда и только тогда, когда p_i не имеет запросов, а имеет некоторые распределенные ресурсы; т. е. если для *любого* j не существует ребер (p_i, R_j) и для некоторого j существуют ребра (R_j, p_i) в графе повторно используемых ресурсов для состояния S . В этой операции p_i может освободить *любое* непустое подмножество своих ресурсов. Результирующий граф состояния идентичен графу состояния S , за исключением того, что удалены некоторые ребра приобретения; удаляется ребро (R_j, p_i) каждой освобожденной единицы ресурса R_j .

На рис. 8.5 показаны эти операции в системе с одним ресурсом (3 единицы) и двумя процессами.

Следует заметить, что процессы являются недетерминированными; при соблюдении приведенных выше ограничений выполнение *любой* операции *любого* процесса возможно *в любое* время. Если невозможно отслеживать время выполнения операций или на процессоры не накладывается ограничений по операциям и ресурсам, то процессы следует считать недетерминированными, так как не существует общего способа узнать заранее, какой ресурс запросит процесс или освободит в конкретный момент времени. В мультипрограммной среде также не существует общего способа узнать *a priori*, какому процессу будет распределен данный ресурс в конкретный момент времени и какой процесс развивается на ЦОУ и, следовательно, может повлиять на изменение состояния. После рассмотрения общего случая мы изучим результаты ограничений, накладываемых на некоторые операции и ресурсы процесса.

8.3.2. Распознавание тупика

Чтобы распознать состояние тупика, необходимо для каждого процесса определить, *сможет ли он когда-либо снова развиваться*. (Для процесса может существовать возможность выполнить операцию в некотором настоящем и будущем состоянии, но это не гарантирует нам, что процесс когда-нибудь действительно снова оживет; распределитель ресурсов или планировщик процессов может не допустить этого либо намеренно, либо случайно.) Такой полезный тупик был рассмотрен в примере 7 разд. 8.1. Так как мы рассматриваем возможность продвижения процесса, а не сам факт продвижения процесса, то достаточно изучить только самые „благоприятные“ изменения состояния. Стратегия распознавания тупика, как показано далее, состоит в моделировании наиболее благоприятного развития для каждого незаблокированного процесса при немультипрограммном (последовательном) режиме работы.

Незаблокированный процесс приобретает любые ресурсы, в которых он нуждается, освобождает *все* свои ресурсы и затем „засыпает“; освобождение ресурсов может „разбудить“ некоторые ранее заблокированные процессы. Это продолжается до тех пор, пока не останется незаблокированных процессов. Если существуют некоторые заблокированные процессы при завершении этой последовательности действий, то начальное состояние S является состоянием тупика, а оставшиеся процессы находятся в тупике в S ; в противном случае S не есть состояние тупика.

Теперь мы формально разработаем и докажем эти положения.

Процесс p_i заблокирован, если он не способен выполнить ни одну из трех операций, определенных в последнем разделе. Это может возникнуть только тогда, когда p_i выдал запросы, которые не могут быть удовлетворены за счет имеющихся ресурсов; т. е. должен существовать по крайней мере один ресурс R_j , такой, что

$$|(p_i, R_j)| + \sum_k |(R_j, p_k)| > t_j.$$

Наиболее благоприятные действия для незаблокированного процесса p_i могут быть представлены *редукцией* (сокращением) графа повторно используемых ресурсов:

1. Граф повторно используемых ресурсов *сокращается* процессом p_i , который не является ни заблокированной, ни изолированной вершиной, с помощью удаления всех ребер, входящих в p_i и выходящих из p_i . Эта процедура является эквивалентной приобретению процессом p_i неких ресурсов, на которые он

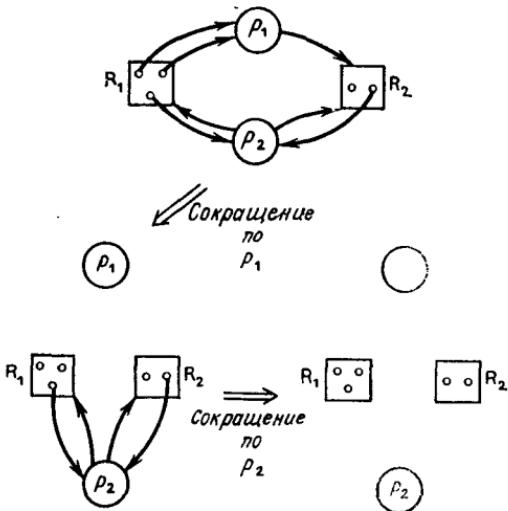


Рис. 8.6. Последовательность сокращений.

выдал ранее запросы, а затем освобождению всех его ресурсов; тогда p_i становится изолированной вершиной.

2. Граф повторно используемых ресурсов *несокращаем*, если он не может быть сокращен ни одним процессом.

3. Граф повторно используемых ресурсов является *полностью сокращаемым*, если существует последовательность сокращений, которые устраниют *все* ребра графа.

На рис. 8.6 показана последовательность сокращений. Граф в состоянии S полностью сокращаем, так как $S \rightarrow U$, а U не содержит ребер. Следует заметить, что p_2 заблокирован в S по ресурсу R_1 , но становится незаблокированным после сокращения процессом p_1 .

Для ресурсов типа SR порядок сокращений несуществен; все последовательности ведут к одному и тому же несокращаемому графу. Это доказывается следующей леммой:

Лемма 1

Все последовательности сокращения данного графа повторно используемых ресурсов приводят к одному и тому же несокращаемому графу.

Доказательство. Допустим, что лемма неверна. Тогда должно существовать некоторое состояние S , которое сокращается до несокращаемого состояния T_1 с помощью последовательности seq_1 и до несокращаемого состояния T_2 с помощью seq_2 , так

что $T_1 \neq T_2$ (т. е. все процессы в T_1 и T_2 или заблокированы или изолированы). Мы придем тогда к противоречию, которое устраняется только при $T_1 = T_2$. Предположим, что seq_1 состоит из упорядоченного списка процессов (q_1, q_2, \dots, q_k) . Тогда seq_1 должна содержать процесс q , который не содержится в seq_2 ; в противном случае $T_1 = T_2$, потому что сокращение только устраниет ребра, уже существующие в состоянии S , а если seq_1 и seq_2 содержат одно и то же множество процессов (в различном порядке), то должно быть удалено одно и то же множество ребер. Теперь доказательство по индукции покажет, что $q \neq q_i$, $i = 1, 2, \dots, k$, приводит к нашему противоречию:

1. $q \neq q_1$, так как S может быть сокращено процессом q_1 , а T_2 должно, следовательно, также содержать q_1 . (Почему?)

2. Пусть $q \neq q_i$, $i = 1, 2, \dots, j$. Но поскольку после сокращения процессами q_i , $i = 1, \dots, j$, возможно сокращение процессом q_{i+1} , это же самое должно быть справедливым для seq_2 независимо от порядка следования процессов; то же самое множество ребер удаляется с помощью q_i . Таким образом, $q \neq q_{i+1}$.

Следовательно, $q \neq q_i$ для $i = 1, 2, \dots, k$ и q не может существовать, а это противоречит нашему предположению, что $T_1 \neq T_2$. Следовательно, $T_1 = T_2$.

Существует простое необходимое и достаточное условие наличия тупика, и оно будет доказано с помощью леммы 1.

Теорема о тупике

Теорема 1

Состояние S есть состояние тупика тогда и только тогда, когда граф повторно используемых ресурсов в состоянии S не является полностью сокращаемым.

Доказательство: (a) Предположим, что S есть состояние тупика и процесс r_i находится в тупике в S . Тогда для всех T , таких, что $S \xrightarrow{*} T$, r_i заблокирован в T . Так как сокращения графа идентичны для серий операций процессов, то конечное несокращаемое состояние в последовательности сокращений должно оставить процесс r_i блокированным. Следовательно, граф не является полностью сокращаемым.

(b) Предположим, что S не является полностью сокращаемым. Тогда существует процесс r_i , который остается заблокированным при всех возможных последовательностях сокращений по лемме 1. Так как любая последовательность сокращений, оканчивающаяся несокращаемым состоянием, гарантирует, что все ресурсы типа SR, которые могут когда-либо стать доступными в действительности освобождены (почему?), то r_i навсегда заблокирован и, следовательно, находится в тупике.

Следствие 1.1

Процесс p_i не находится в тупике тогда и только тогда, когда серия сокращений приводит к состоянию, в котором p_i не заблокирован.

Следствие 1.2

Если S есть состояние тупика (по ресурсам типа SR), то по крайней мере два процесса находятся в тупике в S .

Доказательства следствий оставляются в качестве упражнений.

Из теоремы о тупике непосредственно следуют алгоритмы обнаружения тупиков. Нужно просто попытаться сократить граф эффективным способом; если граф не полностью сокращаемый, то начальное состояние было состоянием тупика. Лемма 1 позволяет удобным образом упорядочить сокращения. Граф повторно используемых ресурсов может быть представлен или матрицами или списками; в обоих случаях экономия памяти достигается слиянием определенных ребер приобретения или ребер запроса между конкретным ресурсом и данным процессом в одно ребро с соответствующим ему весом, определяющим количество единиц ресурса.

1. Матричное представление

Граф представляется двумя матрицами размерности $n \times m$:

а) Матрицей распределения A , $i = 1, \dots, n$, $j = 1, \dots, m$, в которой элемент A_{ij} , выражает количество единиц ресурса R_j , распределенного процессу p_i , т. е.

$$A_{ij} = |(R_j, p_i)|.$$

б) Матрицей запросов B , где $B_{ij} = |(p_i, R_j)|$.

2. Структура связанного списка

Ресурсы, распределенные любому p_i , связаны с p_i указателями

$$p_i \rightarrow (R_x, a_x) \rightarrow (R_y, a_y) \rightarrow \dots \rightarrow (R_z, a_z),$$

где R_j — вершина, представляющая ресурс, и a_j — вес, т. е. $a_j = |(R_j, p_i)|$. Подобным образом и ресурсы, запрошенные процессом p_i , связаны вместе. Аналогичные списки используют для ресурсов. При распределениях ресурса R_i формируется список:

$$R_i \rightarrow (p_u, b_u) \rightarrow (p_v, b_v) \rightarrow \dots \rightarrow (p_w, b_w),$$

где $b_j = |(p_j, R_i)|$; аналогичный список составляют запросы на R_i .

Для обоих представлений удобно также иметь вектор доступных единиц ресурсов (r_1, \dots, r_m) , где r_i задает число доступных (нераспределенных) единиц ресурса R_i , т. е.

$$r_i = t_i - \sum_k |(R_i, p_k)|.$$

Метод прямого обнаружения заключается в просмотре по порядку списка (или матрицы) запросов, причем, где возможно, производятся сокращения до тех пор, пока нельзя будет сделать более ни одного сокращения. Самая плохая ситуация возникает, когда процессы упорядочены в последовательности p_1, \dots, p_n , а единственным возможным порядком сокращений является последовательность p_n, \dots, p_1 , а также когда процесс запрашивает все m ресурсов. Тогда число проверок процессов равно $n + (n - 1) + \dots + 1 = n(n + 1)/2$, причем каждая проверка требует испытания m ресурсов; таким образом, время выполнения в наихудшем случае пропорционально mn^2 .

Более эффективный алгоритм может быть получен за счет хранения некоторой дополнительной информации о запросах. Для каждой вершины процесса p_i определяется счетчик ожиданий w_i , содержащий число ресурсов (не единиц ресурса), которые в какое-то время вызывают блокировку процесса; мы также храним для каждого ресурса запросы, упорядоченные по размеру. Тогда следующий алгоритм сокращений имеет максимальное время выполнения, пропорциональное mn :

```

 $L := \{p_i \mid w_i = 0\};$ 
for all  $p \in L$  do
  begin
    for all  $R_j \ni |(R_j, p)| > 0$  do
      begin
         $r_j := r_j + |(R_j, p)|;$ 
        for all  $p_i \ni 0 < |(p_i, R_j)| \leq r_j$  do
          begin
             $w_i := w_i - 1;$ 
            if  $w_i = 0$  then  $L := L \cup \{p_i\}$ 
          end
        end
      end
    end;
  Deadlock :=  $\neg(L = \{p_1, \dots, p_n\})$ ;
```

L — это текущий список процессов, которые могут выполнять сокращение графа. Программа выбирает процесс p из L , процесс p сокращает граф, увеличивая число доступных единиц r_j .

всех ресурсов R_i , распределенных процессу p , обновляет счетчик ожидания w_i каждого процесса p_i , который сможет удовлетворить свой запрос на частный ресурс R_i и добавляет p_i к L , если счетчик ожидания становится нулевым.

Структура графа обеспечивает простое необходимое, но недостаточное условие для тупика. Для любого графа $G = \langle N, E \rangle$ и вершины $a \in N$ пусть $P(a)$ обозначает множество вершин, „достижимых“ из a , т. е.

$$P(a) = \{b \mid (a, b) \in E\} \cup \{c \mid (b, c) \in E \wedge b \in P(a)\}.$$

Тогда в G имеется цикл, если существует некоторая вершина $a \in N$, такая, что $a \in P(a)$.

Теорема 2

Цикл в графе повторно используемых ресурсов является необходимым условием тупика.

Доказательство. См. упражнение 3.

Теорема о тупике и теорема 2 о цикле являются наиболее общими утверждениями, которые можно сделать об обнаружении тупика, когда не наложено никаких ограничений на использование ресурсов типа SR. Однако проверка на тупик может быть выполнена более эффективно, если она проводится *непрерывно*¹). Следующая теорема является основополагающей при обнаружении тупика.

Теорема 3

Если S не является состоянием тупика и $S \xrightarrow{i} T$, то T есть состояние тупика в том и только в том случае, когда операция процесса p_i есть запрос и p_i находится в тупике в T .

Доказательство оставляем в качестве упражнения.

Смысл теоремы 3 состоит в том, что тупик может быть вызван только при запросе, который не может быть удовлетворен немедленно. В терминах непрерывного распознавания тупика с помощью сокращений графа это означает, что надо применить сокращения только после запроса от некоторого p_i и что на любой стадии необходимо сначала попытаться сократить с помощью процесса p_i ; если процесс p_i смог провести сокращение, то никакие дальнейшие сокращения не являются необходимыми.

¹) Подразумевается обнаружение тупика по мере развития процессов, что в дальнейшем автор называет непрерывным обнаружением. — *Прим. ред.*

Специальные случаи

Ограничения, накладываемые на распределители, на число ресурсов, запрошенных одновременно, и количество единиц ресурсов, приводят к более простым условиям для тупика. Для некоторых из этих специальных случаев полезно другое теоретическое представление графа:

Узел в ориентированном графе $\langle N, E \rangle$ есть подмножество вершин $M \subseteq N$, таких, что для всех $a \in M$, $P(a) = M$.

Каждая вершина в узле достижима из каждой другой вершины этого узла, и узел есть максимальное подмножество с этим свойством. Рис. 8.7 иллюстрирует эту идею. Следует заметить, что цикл есть необходимое, но недостаточное условие для узла.

1. Непосредственные распределения

Если состояние системы таково, что удовлетворены все запросы, которые могут быть удовлетворены, то существует простое достаточное условие существования для тупика. Эта ситуация возникает, если распределители ресурсов не откладывают запросы, которые могут быть удовлетворены, а выполняют их по возможности немедленно; большинство распределителей следует этой стратегии. Результатирующие состояния называются „выгодными“¹⁾.

Выгодное состояние есть состояние, в котором все процессы, имеющие запросы, заблокированы.

Теорема 4

Если состояние системы выгодное, то наличие узла в соответствующем графе повторно используемых ресурсов является достаточным условием тупика.

Доказательство. Предположим, что граф содержит узел K . Тогда все процессы в узле должны быть заблокированы только по ресурсам, принадлежащим K , так как никакие ребра не могут выходить из K по определению. Аналогично, по той же самой причине все распределенные единицы ресурсов в K принадлежат процессам в K . Наконец, все процессы в K должны быть заблокированы согласно условию выгодности и определению узла. Следовательно, все процессы в K должны быть в тупике.

Алгоритм распознавания узла приведен в конце этого раздела при рассмотрении запросов на единицу ресурса.

¹⁾ «Выгодность», вероятно, понимается автором в смысле удобства для последующего обнаруживания тупика, а не выгодность в работе системы. — Прим. ред.

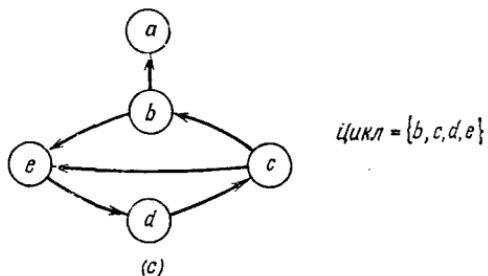
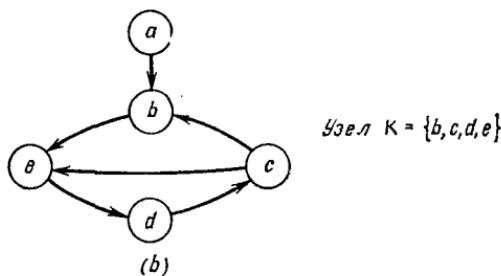
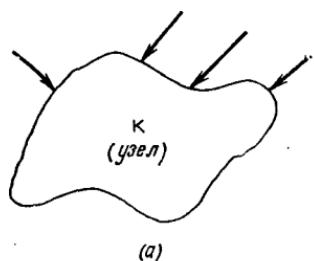


Рис. 8.7. Иллюстрация узлов: (a) — структура узла; (b) — пример узла; (c) — цикл, но не узел.

2. Ресурсы единичной емкости

Допустим, что каждый ресурс имеет единичную емкость, т. е. $t_i = 1$, $i = 1, \dots, m$. При этом ограничении наличие цикла также становится достаточным условием тупика.

Теорема 5

Граф повторно используемых ресурсов с единичной емкостью указывает на состояние тупика тогда и только тогда когда он содержит цикл.

Доказательство. Теорема 2 доказывает необходимость цикла. Для доказательства достаточности допустим, что граф содержит цикл, и рассмотрим только лишь процессы и ресурсы, при

надлежащие циклу. Так как каждая такая вершина-процесс должна иметь входящее и исходящее ребро, она должна иметь выданный запрос на некоторый ресурс, принадлежащий циклу, и должна держать некоторые ресурсы, принадлежащие тому же циклу; аналогично, каждый ресурс единичной емкости в цикле должен быть захвачен некоторым процессом. Следовательно, каждый процесс в цикле блокируется на этом цикле на ресурсе, который может быть освобожден только некоторым процессом из этого цикла. Следовательно, процессы в цикле находятся в тупике.

Чтобы обнаружить тупик в случае ресурса единичной емкости, мы должны просто проверить граф повторно используемых ресурсов на наличие циклов. Пусть *сток* — это любая вершина, которая не имеет ребер, выходящих из нее. Один из эффективных алгоритмов обнаружения цикла основан на последовательном исключении ребер, направленных в сток; если результирующий граф содержит только стоки, то в исходном графе нет ни одного цикла.

Пусть $w_i = \sum_k |(i, k)|$, число ребер, выходящих из вершины i .

Определение цикла для графа $\langle N, E \rangle$:

```

 $S := \{i \mid \text{node } i \text{ is a sink}\};$ 
for all  $i \in S$  do
  begin
    for all  $j \ni (j, i)$  is an edge do
      begin
         $w_j := w_j - 1;$ 
        if  $w_j = 0$  then  $S = S \cup \{j\}$ 
      end
    end;
  Cycle :=  $\neg(S = N);$ 

```

Время выполнения пропорционально числу ребер в графе, которое в худшем случае равно $m n$.

Для непрерывного обнаружения мы можем использовать теорему 3. Тогда необходимо только проверить, находится ли запрашивающий процесс p_i в цикле. Это можно выполнить прослеживанием всех путей, начинаяющихся с p_i , выискивая новое появление p_i .

Ресурсы единичной емкости не столь уж редки в существующих системах; например, макрокоманды *ENQ/DEQ* в системе IBM OS/360 (см. разд. 3.6) имеют дело исключительно с ресурсами этого типа.

3. Запросы на единицу ресурса

Допустим, что за один раз процесс может запросить только единицу ресурса; это означает, что с любой вершиной-процессом может быть связано не больше одного ребра запроса. Тогда для выгодных состояний наличие узла также становится достаточным условием для тупика.

Теорема 6

Граф повторно используемых ресурсов для запросов на единицу ресурса в выгодном состоянии указывает на состояние тупика тогда и только тогда, когда он содержит узел.

Доказательство. По теореме 4 требуется только доказать необходимость узла. Допустим, что граф не содержит узла; мы покажем, что для любой вершины-процесса p_i существует последовательность сокращений, включающая p_i . Рассмотрим любую вершину-процесс p_i , которая не является изолированной. (Последнюю легко можно проигнорировать.) Так как в графе не существует узлов, то p_i или (a) есть сток, или (b) расположен на пути, ведущем к вершине-процессу, которая является стоком. Вершина типа стока в (b) не может быть ресурсом, так как состояние выгодное. (Почему?) В случае (a), сокращение по p_i , очевидно, возможно. Для случая (b) пусть процессы на пути суть $p_i, p_j, p_k, \dots, p_x$, где p_x — вершина типа стока. Граф может быть сокращен процессами $p_x, \dots, p_k, p_j, p_i$ в указанном порядке; этот факт устанавливается простым доказательством по индукции, основанным на том, что каждый запрос требует только единицу ресурса. Следовательно, при известных обстоятельствах сокращение произвольным процессом p_i может быть сделано всегда, и по лемме 1 граф полностью сократим.

Обнаружение узла может быть выполнено аналогичным, но более простым способом, чем обнаружение цикла. Для любой вершины i , связанной со стоком, переменная-счетчик w_i не обязательна, так как независимо от числа ребер, выходящих из i , вершина i не может быть частью узла. Алгоритм графа $\langle N, E \rangle$ есть

```

 $S := \{i \mid \text{node } i \text{ is a sink}\};$ 
for all  $i \in S$  do
    for all  $j \in (i, i)$  is an edge do
        if  $j \notin S$  then  $S := S \cup \{j\}$ ;
    Knot :=  $\neg(N = S)$ ;

```

Если желательно проводить непрерывное обнаружение, снова может быть использована теорема 3. Необходимо только проверить, принадлежит или нет запрашивающий процесс p_i узлу. Мы можем сделать это путем просмотра всех путей из p_i , выискивая сток.

Упражнения

1. Докажите следствие 1.2 теоремы 1.
2. Докажите, что «эффективный» алгоритм обнаружения тупика, использующий счетчик ожидания, имеет максимальное время выполнения, пропорциональное *тп.*
3. Докажите теорему 2. (*Указание.* Используйте следующее свойство направленных графов: если направленный граф не содержит цикла, то существует линейное упорядочение вершин, такое, что если существует путь от вершины *i* к вершине *j*, то *i* появляется перед *j* в этом упорядочении.)
4. Докажите теорему 3.
5. Докажите с помощью контрпримеров, что
 - (а) цикл не есть достаточное условие для тупика,
 - (б) узел не есть необходимое условие для тупика в графах с выгодным состоянием.
6. Докажите, что успешное последовательное устранение ребер, направленных в стоки, приведет к графу, содержащему только вершины типа стока, тогда и только тогда, когда граф не содержит циклов.
7. Опишите алгоритм непрерывного обнаружения тупика в системе с
 - (а) ресурсами единичной емкости,
 - (б) запросами на единицу ресурса.

8.3.3. Выход из тупика

Существуют два общих подхода для восстановления из тупиковых состояний. Первый основан на *прекращении* процессов. Процессы в тупике последовательно прекращаются (уничтожаются) в некотором систематическом порядке до тех пор, пока не станет доступным достаточное количество ресурсов для устранения тупика; в худшем случае уничтожаются все процессы, первоначально находившиеся в тупике, кроме одного. (Почему?) Второй выход основан на *перехвате* ресурсов. У процессов отнимается достаточное количество ресурсов и отдается процессам, находящимся в тупике, чтобы ликвидировать тупик; процессы в первом множестве остаются с выданными запросами на перехваченные у них ресурсы.

Возможно, самым практическим и простым методом является стратегия завершения, по которой первыми уничтожаются процессы с наименьшей ценой прекращения. „Ценой“ прекращения процесса может быть, например:

1. Приоритет процесса.
2. Цена повторного запуска процесса и выполнения до текущей точки согласно обычным нормальным системным учетным процедурам.
3. Простая гештальтная цена, основанная на типе задания, связанного с процессом; например, задания студентов, административные задания, производственные задания, научные задания, а также задания системного программирования, причем каждое может иметь фиксированные цены прекращения.

Алгоритм выхода обычно завершает наименее дорогой процесс, уничтожая его и освобождая его ресурсы, а затем, насколько возможно, сокращает граф; эти шаги повторяются до тех пор, пока граф не станет содержать только изолированные вершины. После выхода из тупика начальное состояние системы — это исходное тупиковое состояние без тех процессов, которые были прекращены. Достоинство этого метода — простота, но, к сожалению, он без разбора разрушает также те процессы, которые оказывают незначительное влияние на тупик¹⁾.

Более удовлетворительная, но менее эффективная схема прекращения процесса выполняет выход из тупика за минимальную цену. Пусть C_i — цена уничтожения процесса p_i . Тогда стратегия с минимальной ценой завершает соответствующее подмножество π' из множества процессов, такое, что

1. Прекращение всех элементов подмножества π' устраняет тупик.

2. Для всех других подмножеств $\tilde{\pi} \subset \pi$, чья ликвидация устраняет условие тупика,

$$\sum_{p_i \in \pi'} C_i \leq \sum_{p_i \in \tilde{\pi}} C_i.$$

Минимальная цена $rcmin(S)$ выхода из тупикового состояния S удовлетворяет отношению

$$rcmin(S) = \min_{(p_i)} (C_i + rcmin(U_i)),$$

(причем
minimum
выбирается
по всем p_i
находящимся
в тупике S)

где U_i — состояние, следующее за S при завершении p_i . Из приведенного выше соотношения непосредственно может быть выведен алгоритм вычисления $rcmin(S)$ и определены соответствующие уничтожаемые процессы. По этому алгоритму последовательно получаются новые состояния до тех пор, пока не возникнет нетупиковое состояние. Пусть $d(T)$ — множество процессов, уничтоженных для достижения состояния T , исходя из начального тупикового состояния S , и пусть L — список новых состояний, полученных из S и упорядоченных по ценам (частич-

¹⁾ Мы не обратили внимания на одну важную практическую проблему: процесс, который выполняется внутри критической секции, должен в общем случае иметь возможность выйти из критической секции прежде, чем он будет прекращен; в противном случае общие области данных могут быть оставлены в некорректных или «нестабильных» состояниях.

ного) выхода из тупика. При завершении работы следующего алгоритма T является начальным состоянием с минимальной ценой выхода, $d(T)$ перечисляет прекращенные процессы и $rc(T)$ — минимальная цена выхода из тупика:

```

 $rc(S) := 0; L := \{ \}; T := S;$ 
while deadlocked( $T$ ) do
  begin
    for all  $p_i$  deadlocked in  $T$  do
      begin
        Terminate  $p_i$  to create state  $U_i$  from  $T$ ;
         $d(U_i) := d(T) \cup \{p_i\}$ ;
         $rc(U_i) := rc(T) + C_i$ ;
        Insert in recovery cost (rc) order( $U_i, L$ )
      end;
       $T := firststatein(L)$ 
      Remove( $T, L$ );
    end;
  
```

Процедура выявления тупика *deadlocked*(T) может быть выполнена путем сокращения графа. Так как элементы T в списке L упорядочены по $rc(T)$ и рассматриваются в этом порядке, то первое нетупиковое состояние есть состояние с минимальной ценой выхода.

Природа „исчерпывающего поиска“ в алгоритме минимальной цены делает его практически неэффективным для сложных тупиковых ситуаций. Например, если C_i — константа, все m ресурсов — это ресурсы с единичной емкостью, $n = m$, каждый ресурс R_i назначен процессу p_i и каждый p_i запрашивает все ресурсы, за исключением R_i , то при выходе из тупика с минимальной ценой затраты времени и памяти пропорциональны по крайней мере $n!$ Однако если в системе запросы могут быть только на единицу ресурса, то существует эффективный и более практический метод выхода.

Допустим, что состояния системы являются „выгодными“ и что запросы могут быть только на единицу ресурса. Тогда по теореме 6 для выхода с минимальной ценой требуется только устраниć минимальной ценой узлы в графе повторно используемых ресурсов. Чтобы устраниć узел, достаточно устранить все ребра, исходящие из любой выбранной вершины в узле, превращая таким образом вершину в сток. Это приводит к следующему алгоритму:

1. Найти все узлы в графе.
2. Для каждого узла выбрать вершину-ресурс R_i , такую, что для всех $R_j \neq R_i$, в узле

$$\sum_{|(R_i, p_k)| > 0} C_k \leq \sum_{|(R_j, p_k)| > 0} C_k.$$

3. УстраниТЬ все узлы, прекращая все процессы, которым распределен выбранный R_i , т. е. прекратить все p_k , такие, что $|(R_i, p_k)| > 0$.

По одному из способов определения узлов в графе (шаг 1 выше) вначале создаются классы эквивалентности всех вершин в общих циклах. Все пути из данной вершины последовательно просматриваются до тех пор пока не повторится вершина или не закончится анализ графа. В первом случае цикл обнаружен и все вершины в цикле заменяются единственной вершиной, представляющей класс эквивалентности; ребра, входящие или выходящие из любой вершины в цикле, аналогично входят или выходят в вершину класса эквивалентности. Этот процесс повторяется до тех пор, пока не будет найдено больше ни одного цикла. Тогда узлы в исходном графе соответствуют классам эквивалентности, являющимся стоками, и содержат по крайней мере две вершины. Для графа повторно используемых ресурсов с запросами на единицу ресурса идентификация узла и остальная часть алгоритма выполняются за время, в худшем случае пропорциональное $(n + m)^2$.

Два алгоритма с минимальной ценой, представленные выше, также могут быть построены на основе перехвата ресурсов. Цена перехвата может быть связана с каждой единицей ресурса. Вместо прекращения процессов, находящихся в тупике, производится перехват ресурсов с минимальной ценой, чтобы ликвидировать тупик. Когда ресурс перехватывается процессом, он передается этому процессу, а у процесса, у которого перехватили ресурс, назначение заменяется запросом на ресурс.

Если обнаружение тупика выполняется непрерывно, то на основании теоремы 3 предлагается простая методика выхода из тупика. Тупик может быть устранен или завершением запрашивающего (находящегося в тупике) процесса, или перехватом ресурсов и назначением их запрашивающему процессу.

Упражнения

1. Проанализируйте алгоритм восстановления с минимальной ценой, используя прекращения процессов в случае, когда C_i — константа, все ресурсы единичной емкости, $n = m$, а каждый процесс p_i держит один ресурс R_i и запрашивает остающиеся ресурсы. Определите, сколько времени и пространства требуется этому алгоритму.

2. Разработайте алгоритм выхода из тупика с минимальной ценой, который использует перехват ресурса, а не прекращение процесса.

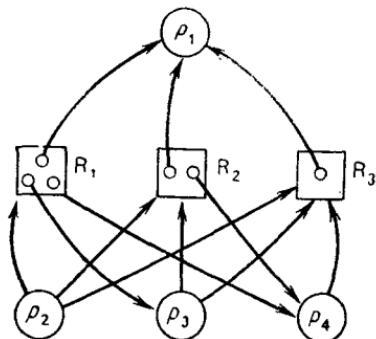
3. Попытайтесь разработать общий алгоритм выхода из тупика с минимальной ценой, основанный на манипуляциях с узлами и циклами повторно используемых ресурсов.

8.3.4. Методы предотвращения тупиков

Вспомним, что безопасное состояние было определено в разд. 8.2 как состояние, которое никогда не может привести к тупику. Общим подходом к задаче предотвращения тупика является решение ограничить систему так, чтобы все состояния были безопасными. Для повторно используемых ресурсов это может быть сделано очень просто — разрешить в любое время владеть ресурсами только одному процессу; на практике стратегия этого типа по существу приводит к однопрограммному, а не мультипрограммному режиму работы, и поэтому, за исключением отдельных ситуаций, может быть отброшена как непрактичная.

Более практическое ограничение состоит в том, чтобы каждый процесс *сразу* (однократно) запрашивал и приобретал ресурсы, в которых он может нуждаться; как правило, это будет самой первой операцией, которую выполняет процесс. Процессы с распределенными для них ресурсами никогда не будут заблокированы, потому что они не могут делать каких-либо дальнейших запросов, а со временем освободят все свои ресурсы не обязательно в один и тот же момент. Таким образом, у данного процесса будут или назначения, или запросы, но никогда и то и другое вместе. Следовательно, тупик невозможен, и каждое состояние безопасно. Другими словами, в графе повторно используемых ресурсов не может содержаться цикл — необходимое условие для тупика (теорема 2 из разд. 8.3.2). Главные недостатки такой стратегии состоят в том, что ресурсы могут быть запрошены задолго до того, как они действительно используются, и что ресурсы могут быть затребованы без необходимости в предвидении возможного использования, которое не состоится.

Стратегия *упорядоченных ресурсов*, первоначально предложенная Хавендером (1968) для IBM OS/360, предотвращает тупик с менее жесткими ограничениями, чем приведенная выше схема „коллективного“ запроса. Повторно используемые ресурсы разделяются на k классов K_1, K_2, \dots, K_k . Процессу разрешается запрашивать ресурсы из любого класса K_i , если только ему не распределены ресурсы из классов K_i, K_{i+1}, \dots, K_k (если $k = 1$, то последний метод и этот идентичны). Можно доказать индукцией по k , что ни одно состояние с указанными выше ограничениями не может быть тупиковым. Ключевая часть дока-



$$K_1 = \{R_1\} \quad K_2 = \{R_2\} \quad K_3 = \{R_3\}$$

Рис. 8.8. Состояние, использующее стратегию упорядоченных ресурсов.

зательства заключается в том, что запрос, если он имеется для наивысшего класса K_k , будет обязательно удовлетворяться. Это должно быть справедливым, так как ни один процесс, которому распределены ресурсы из K_k , не сможет сделать дальнейших запросов до тех пор, пока он не освободит свои ресурсы из класса K_k ; таким образом, имеется гарантия, что все ресурсы из класса K_k будут освобождены. На рис. 8.8 показано состояние системы, соответствующее этой стратегии.

Пример

В IBM OS/360 процессы-инициаторы заданий получают и освобождают ресурсы для задания пользователя и каждого шага в пределах задания, используя стратегию упорядоченных ресурсов. Число классов $k = 3$, и к ним относятся

$$K_1 = \{\text{наборы или файлы}\}, \quad K_2 = \{\text{основная память}\}, \\ K_3 = \{\text{устройства ввода-вывода}\}.$$

Посредством включения самых дорогих или дефицитных ресурсов в наивысшие классы можно будет отсрочить запросы на наиболее ценные ресурсы до тех пор, пока они не станут действительно нужны. Однако здесь имеются те же недостатки, что и в первой „коллективной“ схеме; некоторые ресурсы должны быть распределены заранее, до того как они понадобятся.

Предотвращение тупиков возможно также в таком важном случае, когда известна *максимальная потребность* во всех ресурсах *aприори* для каждого процесса. Максимальная потребность для любого процесса есть наибольшее число единиц каждого ресурса, в которых когда-либо будет нуждаться процесс. Она может быть представлена как матрица потребностей c_{ij} , $i = 1, \dots, n$; $j = 1, \dots, m$, где c_{ij} задает максимальную потребность процесса i в ресурсе j . Система этого типа имеет

следующее дополнительное ограничение на граф повторно используемых ресурсов.

Для каждого процесса p_i и ресурса R_j

$$|(p_i, R_j)| + |(R_j, p_i)| \leq c_{ij} \leq t_j.$$

Это означает, что сумма распределений и запросов любого процесса не может превышать его максимальных потребностей.

Граф потребностей для состояния S определяется как граф повторно используемых ресурсов состояния S , дополненный ребрами запросов (p_i, R_j) в количестве n_{ij} для каждого процесса p_i и ресурса R_j , где

$$n_{ij} = c_{ij} - (|(p_i, R_j)| + |(R_j, p_i)|).$$

Граф потребностей, полученный из S , представляет будущее состояние T в худшем случае, такое, что $S \rightarrow T$ посредством последовательной выдачи запросов всеми процессами на максимально допустимое количество ресурсов. На рис. 8.9 изображен граф потребностей нескольких состояний; дополнительные ребра отмечены пунктиром.

Тупик может быть предотвращен в системах, учитывающих максимальную потребность путем запрещения любых операций приобретения ресурсов до тех пор, пока не будет полностью сократим результирующий граф потребностей. Полная сокращаемость графа потребностей означает, что наихудшим будущим состоянием можно будет управлять; если все процессы последовательно запрашивают остаток своих потребностей, эти запросы могут быть удовлетворены. Эта стратегия предотвращения может быть сформулирована как теорема.

Теорема 7

Если устраниены все операции приобретения ресурсов, которые не приводят к полностью сокращаемому графу потребностей, то любое состояние системы, которое не является тупи-

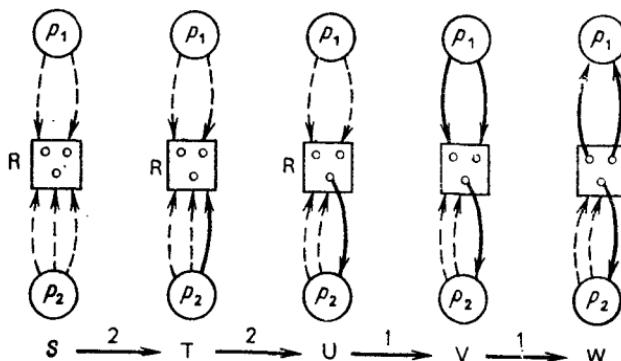


Рис. 8.9. Примеры графов потребностей.

ковым, безопасно. Более того, любая стратегия, которая исключает не все указанные операции приобретения (и не накладывает дальнейших ограничений на операции запроса или освобождения), производит некоторые системные состояния, которые не являются безопасными.

Доказательство. См. упражнение 2.

Процедура сокращения, которая определяет, является ли граф потребностей полностью сокращаемым, должна сначала пытаться сделать сокращение по отношению к приобретающему процессу r_i . Если граф потребностей был полностью сокращаемым до приобретения, то он является полностью сокращаемым после этой операции тогда и только тогда, когда возможно окончательное сокращение по r_i . Этот результат аналогичен теореме 3 о непрерывном обнаружении из разд. 8.3.2. Когда имеются ресурсы *единичной емкости*, существует более эффективный метод проверки сократимости графа потребностей. В результате приобретения получается полностью сокращаемый граф потребностей тогда и только тогда, когда приобретающий процесс r_i не находится в цикле в графе потребностей; следовательно, может быть применен простой алгоритм прослеживания пути из r_i .

На первый взгляд кажется, что не существует какого-нибудь простого способа определения, является ли безопасным состояние в общей системе без ограничений на максимальную потребность. Состояние можно проверить на безопасность с помощью процедуры полного поиска, но этот подход, несмотря на то что он гарантирует получение результата, является слишком расточительным по времени и по памяти (см. упражнение 4). Однако система с известной максимальной потребностью в ресурсах *единичной емкости* может быть легко проверена на безопасные состояния с помощью следующих результатов.

Рассмотрим *ненаправленный* граф потребностей, полученный при представлении каждого ребра графа потребностей как *неупорядоченной пары*¹). *Ненаправленный цикл* определяется как последовательность неупорядоченных ребер (a, b) , (b, c) , ..., ..., (r, s) , (s, a) , в которой ни одно ребро не встречается более одного раза; следует заметить, что в ненаправленном графе ребро (a, b) — это то же ребро, что и (b, a) . Тогда безопасность может быть определена проверкой на циклы.

Теорема 8

В системе с известной максимальной потребностью в ресурсах единичной емкости состояние безопасно тогда и только

¹) То есть ребра не указывают порядок соединения двух вершин, а указывают только факт их соединения. — Прим. перев.

тогда, когда ненаправленный граф не содержит иенаправленных циклов.

Доказательство: (а) Предположим, что ненаправленный граф потребностей не содержит циклов. Тогда, очевидно, граф потребностей (направленный граф) не содержит циклов; тем самым подразумевается, что ни одно будущее состояние не является тупиковым. (Почему?)

(б) Пусть ненаправленный граф потребностей содержит цикл. Тогда ребра могут быть заменены на направленные ребра так, что ненаправленный цикл становится направленным циклом. Таким образом, граф повторно используемых ресурсов, содержащий цикл (достаточное условие для возникновения тупика в системах с ресурсами единичной емкости), может быть создан из ненаправленного графа потребностей. В конечном счете или начальное состояние может быть преобразовано в состояние (тупиковое), представляемое приведенным выше графом, или система входит в тупик до достижения этого состояния. В обоих случаях начальное состояние не является безопасным.

На самом деле нами доказан более сильный результат, так как ненаправленный граф потребностей идентичен для всех состояний.

Теорема 9

Все состояния в системе с известной максимальной потребностью в ресурсах единичной емкости являются безопасными тогда и только тогда, когда (единственный) ненаправленный граф потребностей не содержит ненаправленных циклов.

Системы с известной максимальной потребностью были введены и изучены Дейкстрой (1965а) и Хаберманом (1969). Когда от пользователей требуют специфицировать их потребности в ресурсах (повторно используемых) на картах управления заданием, информация о максимальной потребности в ресурсах полностью доступна. Однако предотвращение тупиков, основанное на знании максимальных потребностей, все еще является потенциально расточительным в отношении ресурсов, так как может оказаться, что процессу никогда не нужно удовлетворять свою максимальную потребность во всех ресурсах сразу.

Таким образом, системный проектировщик имеет две возможности по отношению к ресурсам типа SR. Он может или разрешить возникновение тупика и применить некоторые способы обнаружения и выхода из него, или он может решить предотвратить тупик за счет отказа от, быть может, более полного использования ресурсов. Реалистический подход, принятый во многих системах, заключается в предотвращении тупиков в некоторых классах ресурсов и процессов, для которых выйти из

тупика трудно, и в разрешении его появления в других классах; например, стратегия предотвращения тупика может быть реализована для системных процессов, тогда как обнаружение и выход будут применены только к процессам пользователей.

Проблема тупика по отношению к повторно используемым ресурсам была широко изучена. Другие подходы, которые мы не рассмотрели в этой главе, были разработаны Шошани и Коффманом (1969), Хебалкаром (1970) и Расселом (1972).

Упражнения

1. Докажите, что в следующих двух случаях графы повторно используемых ресурсов никогда не содержат циклы.
 - а) В системе принятая стратегия «коллективных» запросов.
 - б) В системе принятая стратегия упорядоченных ресурсов.
2. Докажите теорему 7. Заметьте, что безопасность или отсутствие безопасности главным образом вытекает только из ограничения на приобретения; запросы и освобождения могут быть сделаны в любом порядке и количестве, будучи ограничены только заданной максимальной потребностью.
3. Для любой системы с известной максимальной потребностью в ресурсах единичного объема докажите, что операция приобретения приводит к безопасному состоянию тогда и только тогда, когда приобретающий процесс не находится в цикле в результирующем графе потребностей.
4. Составьте алгоритм, который определяет, безопасно ли данное состояние в общей системе с известной максимальной потребностью. Оцените, какие максимальные требования ко времени и пространству у вашего алгоритма.

8.4. Системы с потребляемыми ресурсами

Потребляемый ресурс (тип CR) отличается от ресурса типа SR в нескольких важных отношениях:

1. Число доступных единиц ресурса типа CR изменяется по мере того как приобретаются (расходуются) и освобождаются (производятся) отдельные элементы процессами, и такое число единиц ресурса является потенциально неограниченным.
2. Процесс (производитель) увеличивает число единиц ресурса освобождая одну или более единиц, которые он „создал“.
3. Процесс (потребитель) уменьшает число единиц ресурса, сначала запрашивая и затем приобретая одну или более единиц. Единицы ресурса, которые приобретены, в общем случае не возвращаются ресурсу, а потребляются приобретающим процессом.

Эти характеристики присущи многим сигналам, сообщениям и данным, порождаемым как аппаратурой, так и программным обеспечением, и могут рассматриваться как ресурсы типа CR при изучении тупиков. В их число входят: прерывания от таймера и ввода-вывода; сигналы синхронизации процессов; сооб-

щения, содержащие запросы на различные виды обслуживания или данные и также соответствующие ответы; данные, такие, как „дескриптор“ задания, который может быть передан от одного процесса другому по мере прохождения задания пользователя через МС.

Сначала неформально рассмотрим системы с *неограниченными* ресурсами типа CR. Как и в случае ресурсов типа SR система изменяет состояние, когда какой-нибудь *незаблокированный* процесс p_i выполняет одну из трех операций — *затребовать* (*Request*), *приобрести* (*Acquisition*) или *освободить* (*Release*) над ресурсами типа CR. Заметьте, что теоретически не существует конечных пределов на число запрашиваемых и освобождаемых единиц ресурса при условии, что этих единиц остается конечное множество. Свойства, которыми обладает система в отношении тупиков, выводятся легко. Состояние является тупиковым тогда и только тогда, когда все процессы заблокированы. Если есть один незаблокированный процесс в данном состоянии, то он может приобрести затребованные ресурсы, если такие имеются, а затем освободить достаточное количество единиц всех ресурсов, чтобы разбудить каждый заблокированный процесс. Второе свойство состоит в том, что *ни одно* состояние не является безопасным; все незаблокированные процессы могут всегда последовательно запрашивать количество ресурсов, превышающее текущий запас. Более практические и интересные результаты могут быть получены, когда доступна некоторая дополнительная информация о взаимодействиях процессов. Системы ресурсов типа CR могут тогда быть смоделированы и проанализированы с использованием методов, аналогичных разработанных для ресурсов типа SR.

Производители известны

Допустим, что процессы-производители для каждого ресурса известны *a priori*; это значит, что множество процессов, которые могут выполнить операции типа *освобождение*, определено. Тогда состояние системы, содержащее только ресурсы типа CR, может быть представлено направленным графом $\langle N, E \rangle$, называемым *графом потребляемых ресурсов*, со следующими характеристиками:

1. $N = \pi \cup \rho$, где π — множество вершин-процессов $\{p_1, \dots, p_n\}$ и ρ — множество вершин-ресурсов типа CR $\{R_1, \dots, R_m\}$.
2. Граф является двудольным по отношению к π и ρ ; каждое ребро направлено между вершиной π и одним из ρ .
3. Если ребро $e = (p_i, R_j)$, то e есть *ребро запроса* и интерпретируется как запрос процесса p_i на единицу ресурса R_j .

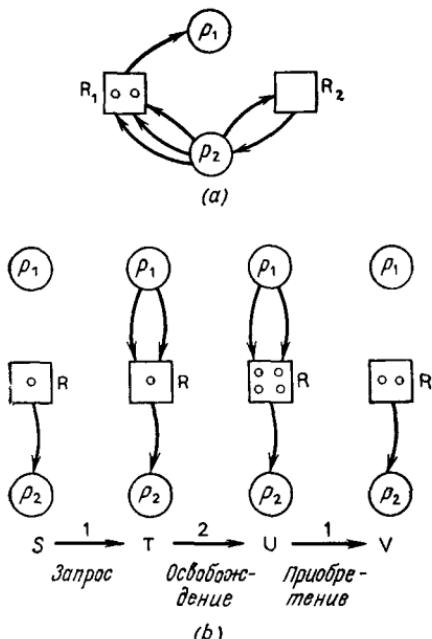


Рис. 8.10. Графы потребляемых ресурсов: (а) — граф потребляемых ресурсов; (б) — операции над графиками типа CR.

4. Для каждого $R \in \rho$ существует непустое множество процессов-производителей $\pi_R \subseteq \pi$. Граф содержит ребро производителя (R, p) , направленное от R к p для всех $p \in \pi_R$. (Ребра производителя являются постоянными ребрами графа и никогда не удаляются.)

5. С каждым $R_i \in \rho$ связано неотрицательное целое r_i , обозначающее число доступных единиц.

На рис. 8.2(б) показан граф потребляемых ресурсов, $\pi_{S_i} = \{p_i\}$ и $r_i = 0$, $i = 1, 2, 3$. Другой пример приведен на рис. 8.10(а).

Процесс p_i , у которого нет выданных запросов, может выполнять операцию типа *запрос* по отношению к любому конечному числу единиц любого числа ресурсов; в новом состоянии добавляются соответствующие ребра запроса. Если p_i имеет выданные запросы, то он может выполнить операцию типа *приобретение*, при условии что он не является заблокированным; т. е. при условии, что для всех R_j , таких, что $(p_i, R_j) \in E$, $|(p_i, R_j)| \leq r_j$; количество доступных единиц каждого приобретаемого ресурса R_j модифицируется:

$$r_j := r_j - |(p_i, R_j)|.$$

Процесс p_i , у которого нет выданных запросов и который является производителем одного или более ресурсов, может выполнить операцию типа *освобождение*, в результате которой к

одному или более ресурсу добавляется любое конечное количество единиц; по мере этого увеличиваются соответствующие счетчики доступных единиц (r_i). На рис. 8.10(b) отражены эти операции и результирующие состояния.

Сокращения графа для ресурсов типа CR также играют основную роль при нашем изучении тупиков. Граф ресурсов типа CR может быть *сокращен* процессом p_i , который не является ни заблокированным, ни изолированной вершиной. Сокращение состоит из следующих двух операций:

1. Удовлетворить все выданные запросы для процесса p_i устранением его ребер запроса и уменьшением соответствующих счетчиков доступных единиц.

2. Для каждого R_j , такого, что p_i есть производитель R_j , освободить достаточное количество единиц ресурса, чтобы удовлетворить все выданные запросы на ресурс R_j и удалить ребро производителя (R_j, p_i). Мы избегаем слежения за доступными единицами ресурса R_j путем присваивания r_j специального числа ω со следующим свойством: $\omega > i$ и $\omega + i = \omega - i = \omega$ для любого целого i ¹⁾.

На рис. 8.11 отображена последовательность сокращений процессами $\langle p_1, p_2, p_3 \rangle$; исходное состояние S было полностью сокращаемым, так как все ребра устраниены в *несокращаемом* состоянии V . (См. определение в разд. 8.3.2.) Заметьте, что в то время как сокращение в графе ресурсов типа SR никогда не уменьшает число доступных единиц, сокращение графа ресурсов типа CR может в действительности оставить после себя меньшее значение r_j ; так на рисунке $r_2 = 1$ в состоянии S и $r_2 = 0$ в состоянии T . Пример на рис. 8.11 иллюстрирует также другое важное отличие систем с ресурсами типа CR — *порядок* сокращений является существенным. Состояние S может быть сокращено процессом p_1 , но если S сначала было сокращено процессом p_2 , то сокращать его процессом p_1 уже нельзя. Лемма 1 из разд. 8.3.2 здесь не справедлива.

Теорема 1 из разд. 8.3.2, которая утверждает, что S есть тупиковое состояние тогда и только тогда, когда граф S не является полностью сокращаемым, здесь также не справедлива, как показано на рис. 8.12; приведенное состояние не является тупиковым, хотя граф не является полностью сокращаемым.

¹⁾ Воспроизведенное по оригиналу свойство числа ω содержит в своем определении ошибку. Правильное формулирование этого свойства не представляется возможным, поскольку автор далее никак не использует данное свойство. Подчеркнем лишь физический смысл ω . При удалении p_i , производителя ресурса R_j , гарантировано, что всем оставшимся процессам-потребителям ресурса R_j будет выделено затребованное ими число единиц ресурса. Замена значения счетчика r_j числом ω и определяет данную ситуацию. — Прим. ред.

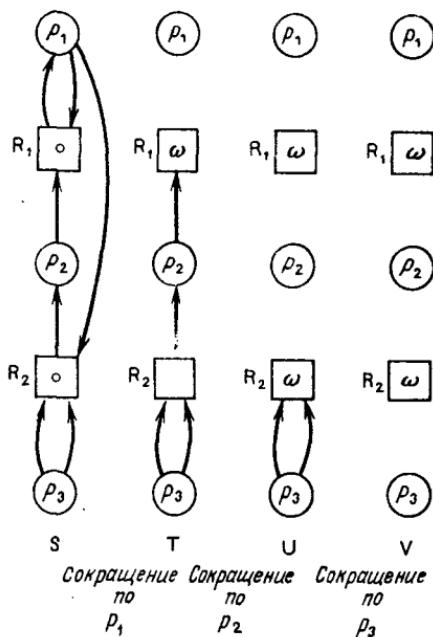


Рис. 8.11. Сокращения графа типа CR.

Однако мы можем все же использовать сокращение графа, чтобы определить, попал ли конкретный процесс в тупик.

Теорема 10

Процесс p_i не находится в тупике в системе с ресурсами типа CR тогда и только тогда, когда существует последовательность сокращений, приводящая к состоянию, в котором p_i не заблокирован.

Доказательство. (а) Предположим, что существует последовательность сокращений, которая оставляет p_i незаблокированным. Тогда p_i не находится в тупике, так как сокращения эквивалентны последовательности операций.

(б) Предположим, что p_i не находится в тупике в состоянии S . Тогда существует последовательность I (возможно, пустая) операций процессов $p_{i_1}, p_{i_2}, \dots, p_{i_x}$, приводящая к со-

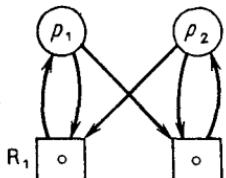


Рис. 8.12. Состояние, которое не является ни тупиковым, ни полностью сокращаемым.

стоянию T , где r_i не заблокирован. Рассмотрим последовательность J сокращений по процессам $p_{I_1}, p_{I_2}, \dots, p_{I_y}$, где для $k = 1, 2, \dots, y$, процесс p_{I_k} — это первый процесс в последовательности I , который ранее не появлялся в последовательности J и который не является изолированной вершиной в состоянии S ; пусть p_{I_y} есть последний процесс в последовательности I , удовлетворяющий этому свойству. Следует заметить, что любое сокращение, которое уменьшает количество доступных единиц ресурса, будет выполнять такое уменьшение только по запросам, существующим в состоянии S . Тогда состояние S может быть успешно сокращено процессами из последовательности J , так как перед сокращением любым процессом p_{I_k} доступных единиц каждого ресурса должно быть по крайней мере столько же, сколько перед первой операцией p_{I_k} в последовательности I . Следовательно, r_i не блокируется после сокращения, потому что он не блокируется в результате последовательности операций I .

Чтобы распознать тупик, нужно искать *различные* последовательности сокращений для каждого процесса, что является в общем случае очень неэффективной задачей. Тогда как в системах ресурсов типа SR тупик может быть вызван только операцией запроса, в случае CR к тупикам могут привести как запросы, так и приобретения. На рис. 8.12 приведен пример, где приобретение процессом p_1 единицы ресурса R_1 и единицы ресурса R_2 ставит p_2 в тупик. Способом, аналогичным доказательству теорем 2 и 4, может быть показано, что узел есть достаточное условие для тупика в выгодных состояниях и что цикл есть необходимое условие для тупика.

Обнаружение тупика значительно упрощается, если система ограничена *запросами на единицу ресурса*; это означает, что из любой вершины-процесса может быть направлено самое большое одно ребро запроса. Тогда теорема 6 справедлива также и здесь.

Теорема 11

Граф потребляемых ресурсов с запросами на единицу ресурса в выгодном состоянии представляет состояние тупика тогда и только тогда, когда граф содержит узел.

Доказательство. См. упражнение 3.

Для обнаружения тупика в этом важном специальном случае также может быть применен алгоритм нахождения узла из разд. 8.3.2.

Единственным практическим способом выхода из тупика является прекращение процессов, находящихся в тупике. Ликви-

дация процессов, не находящихся в тупике, не помогает, так как их ресурсы типа CR израсходованы; перехватывание ресурсов невозможно, потому что перехватывать нечего. Прекращение процесса более сложно, чем в случае ресурсов типа SR, так как кандидат на уничтожение может быть производителем ресурсов, требуемых другими процессами, т. е. другим процессом при некоторых обстоятельствах могут понадобиться сообщения, которые должны вырабатываться процессом, находящимся в тупике. Эта проблема может быть решена выдачей соответствующих сообщений об ошибке, когда делается попытка взаимодействовать с уничтоженным процессом, или в худшем случае прекращением также взаимодействующих процессов, если они известны. Например, если процесс, связанный с заданием пользователя или шагом задания, находится в тупике, то может быть выброшено все задание или шаг задания.

Когда известны *aприори* только процессы-производители и любой процесс является потенциальным потребителем, то ни одно состояние системы не является безопасным по той же самой причине, которая была изложена в начале этого раздела для систем без ограничений. Однако значение потребителей, а также производителей позволяет проверять систему на безопасность и применять простую стратегию предотвращения тупиков.

Производители и потребители известны

Допустим, что как производители, так и потребители каждого ресурса известны *aприори*, т. е. для каждого ресурса R существует непустое множество процессов-производителей π_R^P и непустое множество процессов-потребителей π_R^C . Процесс p может затребовать и приобрести единицы ресурса R , только если $p \in \pi_R^C$, и он может освободить единицы R , только если $p \in \pi_R^P$. Состояние системы опять представлено графом потребляемых ресурсов. В каждой такой системе существует одно состояние, называемое состоянием с ограничением на требования, которое играет особую роль. Состояние с ограничением на требования определяется следующим образом:

1. Процесс p имеет выданный запрос на единицу ресурса R_i , тогда и только тогда, когда $p \in \pi_{R_i}^P$. Никакие другие запросы не имеют места.

2. У всех ресурсов нет доступных единиц, т. е. $r_i = 0$, $i = 1, \dots, m$. Соответствующий граф называется *графом потребляемых ресурсов с ограничением на требования*. Пример приведен на рис. 8.13: $\pi_{R_1}^P = \{p_1, p_2\}$; $\pi_{R_1}^C = \{p_2\}$; $\pi_{R_2}^P = \{p_1\}$; $\pi_{R_2}^C = \{p_2\}$. Основной результат дается теоремой 12.

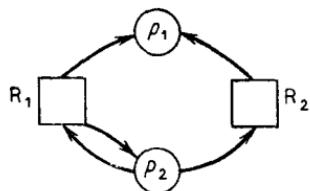


Рис. 8.13. Граф потребляемых ресурсов с ограничениями на требования.

Теорема 12

Все состояния в системе с потребляемыми ресурсами, в которой производители и потребители каждого ресурса известны, безопасны тогда и только тогда, когда граф потребляемых ресурсов с ограничением на требования является полностью сокращаемым.

Доказательство. (а) Допустим, что все состояния безопасны. Тогда состояние с ограничением на требования безопасно и полностью сократимо, так как сокращения соответствуют возможной последовательности операций процессов.

(б) Предположим, что состояние с ограничением на требования V является полностью сократимым последовательностью процессов $p_{v_1}, p_{v_2}, \dots, p_{v_k}$. Достаточно легко показать, что любая последовательность сокращений графа потребляемых ресурсов с ограничением на требования приводит к одному и тому же единственному несокращаемому графу. (Это аналогично лемме 1 из разд. 8.3.2, см. упр. 4.) Следовательно, V есть безопасное состояние. Теперь мы покажем, как любое состояние S может быть сокращено той же самой последовательностью, которая сокращает V . Первый процесс в сокращении V , а именно p_{v_1} , не может быть заблокирован в состоянии S . (p_{v_1} не может быть потребителем любых ресурсов, потому что он был бы тогда заблокирован в V .) Пусть S сокращено с помощью p_{v_1} до состояния S_1 . Второй процесс p_{v_2} не может быть заблокирован в S_1 , так как p_{v_2} должен быть потребителем только ресурсов, произведенных процессом p_{v_1} . (Если p_{v_2} есть изолированная вершина в S , мы игнорируем его.) В противном случае p_{v_2} не смог бы выполнить второе сокращение из V . Будем выполнять последовательность этих действий, заканчивая сокращения процессом p_{v_k} . Результирующее состояние, например S_k , должно состоять из изолированных процессов, потому что последовательность V включает все процессы-производители и потребители. Таким образом, S является полностью сократимым. Поскольку S есть произвольное состояние, ни одно состояние в системе не является тупиковым. Следовательно, все состояния безопасны.

Теорема 12 указывает на одну схему предотвращения тупиков: граф с ограничением на требования проверяется на полную сократимость. Если он полностью сокращаемый, систему можно запустить, не заботясь о тупике; в противном случае системный проектировщик попытается модифицировать взаимодействия процессов, чтобы достигнуть сокращаемости. Однако это может оказаться не легким. Например, граф с ограничением на требования не является полностью сокращаемым, если не существует по крайней мере одного процесса, который не потребляет ресурсов; если система спроектирована так, что каждый процесс должен как получать, так и посыпать сообщения, то это условие не выполняется. Тупик также может быть предотвращен устранением циклов в графе с ограничением на требования; отсутствие циклов является даже более сильным ограничением, чем полная сократимость. Следует заметить, что отсутствие безопасности не обязательно означает, что данная система когда-нибудь действительно попадет в тупик; логика программы каждого процесса может быть такой, что тупиковые „пути“ никогда не будут выбраны при выполнении.

Упражнения

1. Докажите, что в системе с ресурсами типа CR с известными производителями

- (а) Цикл есть необходимое условие для тупика.
- (б) Если состояние системы выгодное, то узел есть достаточное условие для тупика.

2. Рассмотрите систему с ресурсами типа CR с единственным ресурсом R и известными производителями ресурса R. Выведите простое необходимое и достаточное условие тупика в такой системе.

3. Докажите теорему 11. Рассмотрите доказательство теоремы 6 из разд. 8.3.2.

4. Докажите, что для любого состояния S в системе с ресурсами типа CR с известными производителями и потребителями $S \xrightarrow{*} V$, где V есть состояние с ограничением на требования.

5. Докажите, что все возможные последовательности сокращений данного графа потребляемых ресурсов с известными производителями и потребителями приводят к одному и тому же несокращаемому графу.

8.5. Графы обобщенных ресурсов

Модели из разд. 8.3 и 8.4 могут быть объединены для представления систем, содержащих как повторно используемые, так и потребляемые ресурсы. Мы полагаем, что производители известны для потребляемых ресурсов. Состояние операционной системы представляется графом обобщенных ресурсов $\langle N, E \rangle$, где

1. $N = \pi \cup \rho$. π есть непустое множество вершин-процессов $\{p_1, p_2 \dots, p_n\}$ и ρ есть непустое множество вершин-ресурсов $\{R_1, \dots, R_m\}$.

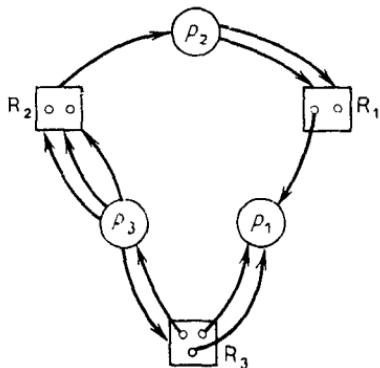


Рис. 8.14. Граф обобщенных ресурсов.

2. $\rho_s = \rho_s \cup \rho_c \cdot \rho_s$ есть множество повторно используемых ресурсов и ρ_c есть множество потребляемых ресурсов.

3. Каждое ребро $e \in E$ направлено между вершиной π и вершиной ρ . Если $e = (p_i, R_j)$, то e есть ребро запроса и интерпретируется как запрос процесса p_i на единицу ресурса R_j . Если $e = (R_j, p_i)$, то: (а) если $R_j \in \rho_s$, то e есть ребро назначения и показывает распределение единицы ресурса R_j процессу p_i , или (б) если $R_j \in \rho_c$, то e есть ребро производителя и показывает, что p_i есть производитель ресурса R_j . Ребро производителя постоянно направлено от каждого ресурса ко всем его процессам-производителям.

4. Для каждого $R_i \in \rho_s$ существует связанное с ним положительное целое t_i , обозначающее общее количество единиц R_i , т. е. $r_i = t_i - \sum_j |(R_i, p_j)|$.

5. Для всех $R_i \in \rho_s$ и процессов p_j должны выдерживаться следующие соотношения:

$$(a) \sum_j |(R_i, p_j)| \leq t_i.$$

$$(b) |(R_i, p_j)| + |(p_j, R_i)| \leq t_i.$$

6. Для каждого $R_i \in \rho_c$ существует неотрицательное целое r_i , указывающее количество доступных единиц ресурса R_i .

На рис. 8.2(а) и 8.14 показаны примеры графов обобщенных ресурсов; на рис. 8.14 $\rho_s = \{R_1, R_3\}$, $\rho_c = \{R_2\}$, $t_1 = 2$, $t_3 = 3$, $r_1 = 1$, $r_2 = 2$, $r_3 = 0$.

Операции типа запроса, приобретения и освобождения определяются так же, как в разд. 8.3.1. Сокращения определяются аналогичным образом. Следующие основные теоремы о тупиках легко доказываются с использованием результатов двух предыдущих разделов.

Теорема 13

Процесс p_i в системе с обобщенными ресурсами не находится в тупике тогда и только тогда, когда существует последовательность сокращений, приводящая к состоянию, в котором p_i не заблокирован.

Теорема 14

В графе обобщенных ресурсов

- (a) Цикл есть необходимое условие тупика.
- (b) Если состояние системы выгодное, то узел является достаточным условием для тупика.

Теоремы 6 и 11 также справедливы для графов обобщенных ресурсов.

Тупик может быть обнаружен сокращениями графа, но из-за того что существен порядок сокращений, этот метод совершенно неэффективен. Для запросов на ресурсы единичной емкости алгоритм определения узлов позволяет быстро определить наличие тупика. Известно несколько дополнительных результатов для случая обобщенных ресурсов.

Упражнения

1. Докажите теоремы 13 и 14.
2. Докажите аналог теорем 6 и 11 для случая обобщенных ресурсов.
3. Допустим, что максимальные потребности известны для ресурсов типа SR и что производители и потребители известны для ресурсов типа CR. Попытайтесь постулировать и вывести результаты, аналогичные теореме 7 из разд. 8.3.4 и теореме 12 из разд. 8.4.

8.6. Динамическое добавление и удаление процессов и ресурсов

Модели, разработанные в этой главе, подразумевают наличие постоянного множества процессов и ресурсов в системе. Это обычно справедливо для большинства системных процессов и многих повторно используемых ресурсов. Однако даже в современных системах процессы непрерывно создаются и разрушаются и некоторые ресурсы являются динамическими. Пример последних встречается в IBM OS/360, которая разрешает пользователям запрашивать и освобождать их собственные ресурсы типа SR посредством макрокоманд ENQ/DEQ. В этом разделе мы кратко рассмотрим системные тупики в среде, допускающей динамическое создание и уничтожение процессов и ресурсов.

Рассмотрим сначала следствия разрешения процессам входить в систему и покидать ее, если система содержит *постоянное* множество ресурсов типа SR. Удаление процесса является,

по существу, эквивалентным сокращению этим процессом и может помочь только в тупиковой ситуации, так как число доступных единиц ресурсов типа SR может возрасти. С другой стороны, в общем случае введение нового процесса увеличивает конкуренцию за ресурсы, и, таким образом, становится легче достигнуть тупикового состояния. В обоих случаях алгоритмы обнаружения тупика, представленные в разд. 8.3.2, непосредственно применимы к новому множеству процессов. В системе добавляются две новые операции:

- Создать* процесс. Она добавляет к графу или изолированную вершину-процесс, или, возможно, вершину-процесс с некоторым начальным распределением.

- Разрушить* процесс. Вершина-процесс и все ее ребра назначения и запроса удаляются из графа.

Рассуждения относительно предупреждения тупиков с ресурсами типа SR (разд. 8.3.4) применимы также к динамической ситуации. Добавление или удаление процесса в системе в соответствии со стратегией упорядочения ресурсов по-прежнему оставляет систему в безопасном состоянии. Если предотвращение тупика основывается на информации о максимальной потребности с ограничением на приобретение (см. теорему 7), то удаление процессов только упрощает проверку операций приобретения; добавление нового процесса к системе в безопасном состоянии по-прежнему приводит к безопасному состоянию, так как новый процесс, очевидно, может быть сокращен в *последнюю* очередь при сокращении графа потребностей.

Динамические процессы, работающие с ресурсами типа CR, приводят к несколько более сложным эффектам. Если известны только производители, тогда либо создание, либо прекращение процесса может привести в тупик. Удаление процесса-производителя может привести в тупик, в то время как удаление процесса, который не производит ресурсов, в некоторых случаях может предотвратить будущий тупик (рис. 8.15). В первом случае некоторые процессы могут бесконечно долго ожидать сообщения от производителя, который удален, если с этим ресурсом-сообщением был связан только один процесс-производитель; это легко может быть исправлено на практике путем посылки запрашивающим процессам сообщения о том, что производитель больше не существует. (Примитивы связи Бринча Хансена реализованы таким способом; см. разд. 3.5.) Аналогично, введение нового процесса-производителя может предотвратить возможный тупик (или даже ликвидировать тупик), тогда как добавление потребителя может ухудшить ситуацию (рис. 8.16). Если известны и производители и потребители, то могут наблюдаться те же эффекты. Однако результаты разд. 8.4 остаются

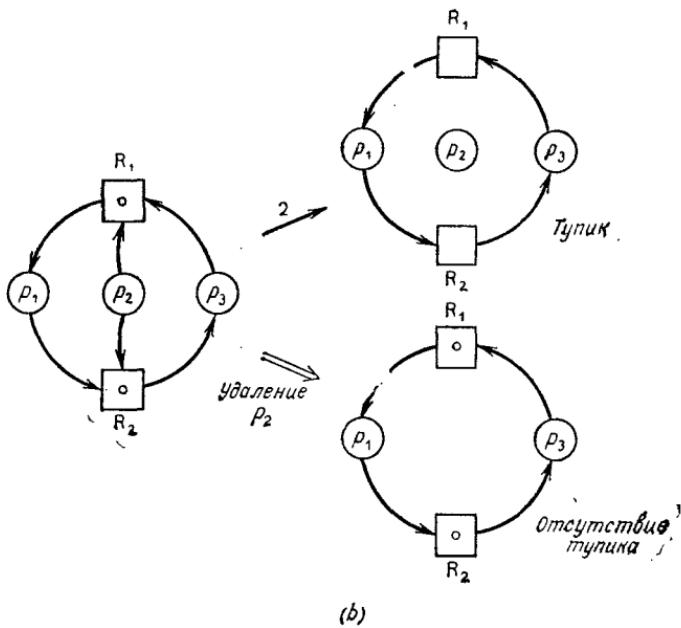
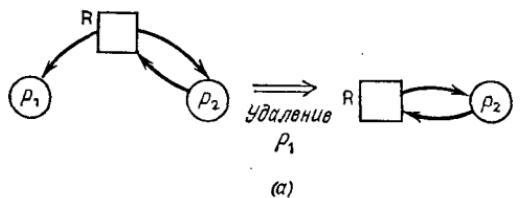


Рис. 8.15. Уничтожение процессов: (а) — уничтожение процесса, приводящее к тупику; (б) — предотвращение тупика в будущем с помощью уничтожения процесса.

справедливыми и могут непосредственным образом применяться для обнаружения тупика и предотвращения его.

На графовых моделях также можно производить динамическое добавление и удаление ресурсов. Удаление ресурса типа SR в худшем случае не приведет к тупику, а в лучшем — предотвратит или даже ликвидирует тупик; в последнем случае цикл или узел в графе ресурсов типа SR может исчезнуть в результате ликвидации ресурса. (Конечно, с практической точки зрения удаление ресурса типа SR, такого, как файл данных или буферная область, может быть катастрофическим для процесса, которому назначен ресурс, или процесса, запрашивающего ре-

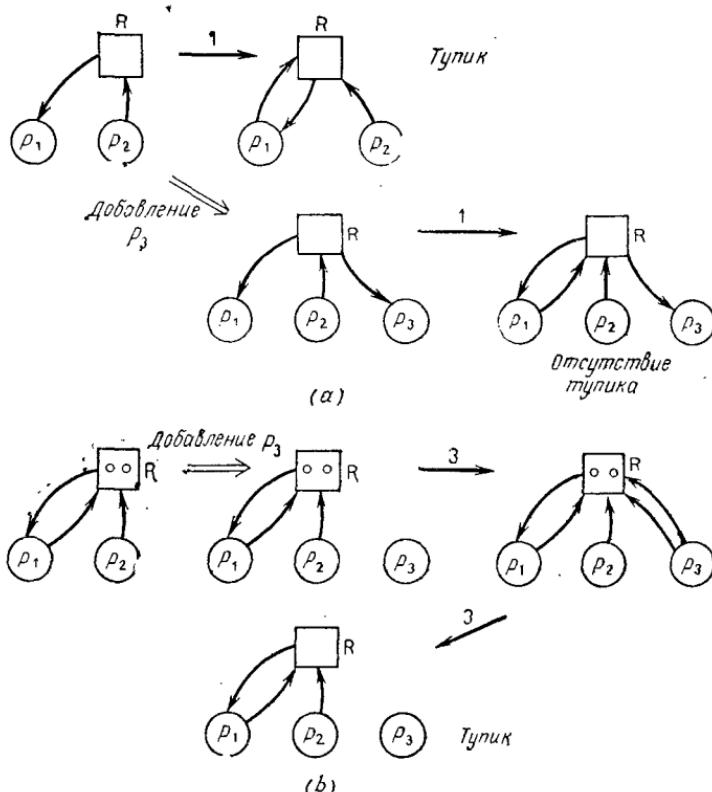


Рис. 8.16. Результаты добавления процессов: (а) — добавление процесса-производителя; (б) — добавление процесса-потребителя.

сурс, что делает необходимым прекращение также и этого процесса.) Введение нового ресурса типа SR вызывает только усложнение методов распознавания тупика. При предотвращении тупика с использованием информации о максимальных потребностях в ресурсах типа SR добавление нового ресурса имеет смысл только в том случае, если все процессы, запрашивающие этот ресурс, были введены *после* того, как новый ресурс определен. Если система была безопасна до того, как создан новый ресурс, она остается безопасной, при условии что в дальнейшем не нарушается ограничение на приобретение, приведенное в теореме 7. Аналогично, удаление ресурса типа CR выгодно лишь по отношению к тупику. Влияние добавления нового ресурса типа CR на тупик оставляем в качестве упражнения для читателя.

9. Файловые системы

Третий (и последний) важный компонент мультипрограммных систем¹⁾, файловая система, является самой большой частью многих МС, и для ее проектирования и реализации обычно требуются наибольшие усилия. В этой главе определяются основные функции файловых систем и описывается ряд методов и структур, применяемых для реализации этих функций.

Файл обычно определяется как набор записей данных, сгруппированных для того, чтобы управлять доступом к ним, их чтением и модификацией; запись данных — это просто линейный список объектов информационного характера. Примерами файлов являются программы на исходных или машинных языках, прикладные данные, такие, как счета или инвентарные списки, задания пользователей, справочники файлов, а также учетные данные о работе или производительности системы. Фактически все программы и данные, к которым имеет доступ вычислительная система, т. е. все программное обеспечение ЭВМ может в некоторые моменты обработки рассматриваться как файл. Файлы могут располагаться на разнообразных носителях, к которым относятся карты, диски, барабаны, магнитная лента, основная память и бумага; с каждым видом носителя связано обрабатывающее устройство, которое обеспечивает считывание данных с этого носителя или запись на него, а также обеспечивает операции управления и контроля. Часто один и тот же файл во время своего существования занимает различные носители. Например, файл на перфокартах, содержащий пользовательскую программу, может последовательно попасть на карты, диск и в основную память, по мере того как программа вводится и загружается; или файл счетов может одновременно находиться на магнитной ленте и бумаге.

Файловая система (*система управления данными*) — это то программное обеспечение, которое отвечает за создание, уничтожение, организацию, чтение, запись, модификацию и перемещение файловой информации, а также управление доступом к файлам и за управление ресурсами, которые используются файлами. Без файловой системы мультипрограммирование в общем

¹⁾ Два других компонента, рассмотренных в предыдущих главах, это подсистемы управления процессами и ресурсами соответственно.

виде не было бы возможным. Это именно тот компонент, который обеспечивает использование многих языковых процессоров, пользовательских библиотек программ и данных, обрабатываемых в режиме „в линию“, операции по организации спулинга в пакетных системах и практические интерактивные вычисления. Как мы уже упоминали раньше, весь ввод-вывод в МС обычно обслуживается посредством файловой системы.

9.1. Виртуальная и реальная файловая память

Наше обычное определение файла как набора связанных (родственных) записей данных не навязывает файлу какой-либо особой структуры (только подразделение на записи) и почти не зависит от носителя файла. Целесообразно сделать еще шаг в этом направлении и определить единообразную бесформатную *виртуальную файловую память* (MULTICS, 1967; Мэдник и Олсон, 1969). На самом абстрактном и неструктурированном уровне мы будем рассматривать файл как упорядоченную последовательность элементов с символическим именем, где элемент файла есть наименьшая адресуемая единица в виртуальном пространстве, такая, как байт или слово, *e*-й по порядку элемент файла с именем *F* адресуется парой $[F, e]$. Читатель заметит, что файл теперь идентичен *сегменту* виртуальной памяти, как определено в разд. 5.1.3.

Цель концепции виртуальной файловой памяти — обеспечить пользователей простым единообразным линейным пространством для размещения их файлов. Это пространство может быть далее структурировано (как структурирована ЭВМ и ее реальная файловая память) любым удобным способом, чтобы отразить истинную организацию данных и их обработку, или, если известны реальные аппаратурные устройства памяти, извлечь преимущества из их физических характеристик.

Самыми основными операциями в файловой системе являются: а) отображение запросов на доступ из виртуального на реальное файловое пространство и б) передача элементов между файловой памятью и основной памятью ЭВМ. Изучение этих и других операций требует определенного знания свойств и поведения *реальных* устройств для хранения файлов. В оставшейся части этого раздела кратко рассматриваются свойства самых распространенных устройств. Более детальные описания могут быть найдены в технической документации и в нескольких публикациях, например (Гиер, 1969, гл. 6). Мы ограничиваем обсуждение вспомогательным запоминающим устройством (ЗУ), работающим в режиме „в линию“ (*on-line*), которое является *повторно используемым*, т. е. средой памяти, с которой можно читать и на которую можно записывать много раз.

Память только для чтения или только для записи, такая, как карты, бумажная лента и перфолента, и связанные с ними обрабатывающие устройства не рассматриваются.

Основные устройства вспомогательной памяти, которые использованы как файловые, — это магнитная лента, диски, барабаны и ферритовая память. Ниже перечислены некоторые их показатели, важные для разработчиков систем и пользователей, причем не делается никаких различий между *устройством*, на котором данные хранятся, и *контроллером*, который управляет выборкой устройства, считыванием, записью и другими операциями:

1. *Емкость*. Это максимальное количество данных, которые могут храниться на устройстве.

2. *Размер записи*. Физическая запись есть непрерывно расположенная информация наименьшего объема, к которой можно адресовать на устройстве. Устройство может допускать записи *фиксированной* или *переменной* длины.

3. *Метод доступа*. Может быть возможным *прямой* доступ к любой записи на устройстве или устройство может быть только с *последовательным* доступом. В первом случае по аппаратурному адресу записи механизм считывания записи устанавливается непосредственно на запись. В случае только последовательного доступа, определенная запись достигается при явном прохождении в прямом или обратном направлении, минуя промежуточные записи; обычный способ работы состоит в том, чтобы получать доступ к записям последовательно в той же самой линейной последовательности, в которой они хранятся.

4. *Сменяемость*. Магнитная лента и некоторые типы дисков являются *сменными*, допуская существование автономной памяти. Это свойство существенно увеличивает объем виртуальной памяти, предоставляемой пользователю, так как оператору ЭВМ можно дать инструкции монтировать активные файлы и удалять неактивные.

5. *Скорость передачи данных*. Это скорость, обычно выражаемая в битах, байтах или словах/секунду, с которой данные могут быть переданы между основной памятью и устройством. Во время передачи ввод-вывод имеет приоритет над ЦОУ в отношении доступа к памяти и, если они оба обращаются к одному и тому же блоку основной памяти, „крадет“ циклы памяти у ЦОУ.

6. *Задержка (латентность)*. После того как команда чтения или записи получена контроллером устройства, обычно требуется дополнительное время (время *задержки*), прежде чем головки считывания или записи переместятся к началу требуемой записи и можно будет начать передачу данных. Это или время

разгона магнитной ленты до приобретения рабочей скорости из состояния полной неподвижности, или время поворота вращающихся дисков или барабанов. Во время этой задержки ЦОУ, если его тщательно программировать, может выполнять свои операции с нормальной скоростью.

7. *Время установки.* Дисковые устройства с движущимися головками чтения/записи часто должны выполнить до каждой

Таблица 9.1. Типичные характеристики устройств вспомогательной памяти

	Магнитная лента	Диск	Барабан	Ферритовое ЗУ
Емкость	$\sim 10^7$ В/бобину	$\sim 10^6$ — 10^8 В/пакет дисков	$\sim 10^7$ В	$\sim 10^7$ В
Размер записи	V	F или V	F	F
Метод доступа	S	D	D	D
Перемещаемость	R	R или NR	NR	NR
Скорость передачи данных	~ 30000 — 300000 В/с	~ 150000 — 300000 В/с	~ 300000 — 1200000 В/с	$\sim 10^7$ В/с
Задержка	~ 5 мс	~ 12 мс	~ 8 мс	—
Время установки	—	~ 75 мс	—	—
Относительная стоимость *)	1	2	3	4

Обозначения:

В—байты

F—фиксированная длина, V—переменная длина

S—последовательный, D—прямой

R—перемещаемый, NR—неперемещаемый

Примечание; Технология запоминающих устройств (ЗУ) изменяется так быстро и в пределах каждого класса доступен такой широкий спектр устройств, что приведенные выше цифры и свойства должны считаться очень приблизительными.

*) Чем выше показатели, тем больше стоимость.

операции считывания или записи *установку*, при которой головка физически перемещается к дорожке, содержащей желаемую запись. Так же как и во время задержки, ЦОУ может полностью использовать время поиска.

8. *Относительная стоимость.* Как можно ожидать, стоимость устройства возрастает с увеличением скорости и гибкости доступа к записям.

В табл. 9.1 перечислены типичные характеристики четырех основных классов устройств вспомогательной памяти в терминах перечисленных выше показателей. Основными препятствиями для применения магнитной ленты для файловой памяти в режиме „в линию“ является то, что она требует

последовательного доступа, и практически невозможно (на большинстве лентопротяжных устройств) делать выборочные изменения записей без перезаписи всей ленты. С другой стороны, ленты недороги, малы и мобильны, т. е. обладают свойствами, которые делают целесообразным использование их в качестве архивной автономной памяти, а также в качестве памяти для последовательных файлов. Благодаря большой емкости, возможности прямого доступа и низкой стоимости диски стали в настоящее время самыми широко используемыми устройствами вспомогательной памяти по сравнению с барабанами и ферритовой памятью большого объема. Барабаны обычно обеспечивают более высокие скорости передачи данных и меньшую задержку („латентность“), чем диски, и часто используются для хранения самых активных файлов, а также при страничной организации и в других динамических системах памяти. Ферритовое запоминающее устройство относительно недавно стало использоваться для файловой памяти, причем способ построения стандартной (но более медленной) основной ферритовой памяти переносится на вспомогательную память. Основные преимущества этой памяти — отсутствие времени задержки и установки и эффективная скорость передачи данных, сравнимая с временем доступа к основной памяти.

9.2. Компоненты файловой системы

Большинство главных компонентов файловой системы и решаемых ею задач для МС общего назначения может быть определено при изучении „пути“, который проходит типичный запрос на доступ к файлу, порождаемый на уровне виртуального файла. Мы рассмотрим простейший возможный пример — запрос на чтение следующей записи из виртуального файла на перфокартах.

Пользователь U полагает, что он считывает свои перфокарты, на которых находятся данные, с устройства чтения с перфокарт. Чтобы прочитать следующую карту (запись), U^1 может выдать любую из следующих команд:

- (1) $\text{ReadCard}(A)$,
- или $\text{Read1}(\text{MyCardFile}, A)$;

Любая из команд приводит к чтению следующей 80-символьной записи файла MyCardFile в ячейки основной памяти, например в ячейки $A[0], \dots, A[79]$; MyCardFile есть виртуальный файл, адресуемый как $[\text{MyCardFile}, 0][\text{MyCardFile}, 1] \dots$

¹⁾ Для простоты предположим, что процесс, связанный с U , также обозначается символом U .

и хранящийся на устройстве вспомогательной памяти, возможно, в результате операции спулинга.

Файл *MyCardFile* обычно имеет структуру, соответствующую последовательному устройству на перфокартах с 80-символьными записями. Такая структура данных и последовательная организация могут быть легко реализованы. Пусть *r* — указатель следующей записи, которую надо считать последовательно в *MyCardFile*. Тогда команда (1) может быть преобразована файловой системой в такую последовательность:

(2) *Read2(MyCardFile, A, r, 80);*
r := r + 80;

Первые две команды чтения, *Read1* и *Read2*, используют символьическое имя для обращения к файлу. Следующим шагом является нахождение *внутреннего имени* *MyCardFile* (пусть оно будет *mcf*), которое однозначно определяет файл в системе и обеспечивает поиск его *дескриптора*; *mcf* может просто быть адресом этого дескриптора во вспомогательной памяти. Если файл был объявлен ранее или к нему обращался *U*, то наиболее вероятно, что он „открыт“, и дескриптор появляется в *справочнике активных файлов* (AFD) в основной памяти; в противном случае файл *MyCardFile* должен быть найден по *справочнику* всех файлов в системе. Тогда выполняются следующие операции:

(3)₁ *open := false;*
if LookInAFD(MyCardFile, descr) then open := true
else SearchSystemsDirectory(MyCardFile, mcf);

Здесь *LookInAFD* — булева процедура, которая ищет файл в AFD; если поиск завершается успешно, она возвращает значение *true* и помещает дескриптор файла в *descr*. Процедура *SearchSystemsDirectory* помещает идентификатор файла в *mcf*.

После поиска в справочнике файл должен быть открыт, т. е. должен стать активным (если он еще не активен), затем проверяется допустимость запроса на чтение. Открытие файла, по существу, включает нахождение дескриптора и запесение его в AFD (если файл *MyCardFile* не использовался в режиме „в линию“, то оператор ЭВМ находит файл в автономной памяти и „монтирует“ его). Открытие файла *MyCardFile* может быть выражено следующим образом:

(3)₂ *if* \neg *open* *then*
begin
GetDescriptor(mcf, descr);
InsertInAFD(descr)
end;

Дескриптор файла *MyCardFile* включает в себя следующую информацию:

Имя:	<i>MyCardFile</i>
Положение:	\langle номер устройства, адрес \rangle
Организация:	Последовательная
Длина:	<i>L</i>
Тип:	Временный
Владелец:	<i>U</i>
Пользователи:	{ <i>U</i> }
Доступ:	{ <i>Read</i> }
⋮	⋮

Далее дескриптор используется, чтобы проверить, имеет ли *U* право запрашивать считывание:

(3)₃ if $\neg (U \equiv \text{descr}(\text{Users}) \wedge \text{Read} \equiv \text{descr}(\text{Access}))$
 $\wedge r + 79 \leqslant \text{descr}(\text{Length})$
then *Error*;

где *descr*(*X*) извлекает содержимое поля, названного *X*, из дескриптора. Программа *Error* может или отбросить запрос, выданный пользователем *U*, или послать *U* соответствующее сообщение. Наконец, если чтение действительно допустимо, дескриптор передается на следующий этап обработки:

(3)₄ *Read3(descr, A, r, 80);*

Таким образом, команда *Read2* в (2) вызывает четыре шага (3) работы файловой системы.

Запись, определенная в команде (3)₄, помещается затем по действительному адресу записи во вспомогательную память, где желаемая запись располагается как физическая. Во многих случаях применяют некоторый тип буферизации. Если предположить, что буферизация для файла *MyCardFile* была создана предварительно, например при открытии, то (3)₄ может вызвать следующие буферные операции:

(4)₁ *GetBuf(k); Move(Buf[k], A); ReleaseBuf(k);*

Здесь процедура *GetBuf(k)* возвращает индекс *k* следующего полного буфера *Buf[k]*, в то время как *ReleaseBuf(k)* освобождает *Buf[k]* (см. разд. 2.4). Завершение (4)₁ обычно заканчивает выполнение исходного запроса. С буферизацией файла может быть синхронизован процесс, который считывает следующий последовательный блок записей, когда доступно достаточно-

ное буферное пространство. Команда считывания может иметь вид

Read4(Device, RecordAddress, Buffer, No.ofElements);

Например, следующий набор записей для файла *MyCardFile* может быть обнаружен по физическому адресу *pra* на устройстве номер 17; система буферизации может быть задумана так, чтобы принимать 800 символов (10 логических записей) в 10 непрерывных буферов длиной 80 символов. Предположим, что файл *MyCardFile* хранится последовательно на устройстве. Операция *Read4* тогда будет иметь вид

(4)₂ *Read4(17, pra, Buf[nextR], 800);*

pra := pra + 800; nextR := nextR +_n 10;

[предполагается, что *nextR* — это указатель кольцевой буферной области памяти, содержащей *n* 80-байтовых буферов (см. разд. 2.4.2.)]. Когда эта операция будет завершена, распределение пространства вторичной памяти для этого файла может быть освобождено. Чтобы выполнить желаемую операцию, при обработке *Read4* из (4)₂ должна быть сгенерирована последовательность команд канала и приказов устройствам (программа ввода-вывода). Если файл хранится на диске с перемещаемым механизмом доступа, то программа ввода-вывода может также содержать команды, вызывающие вначале перемещение его на дорожку, содержащую желаемую запись. Затем полностью преобразованный запрос помещается в очередь для распределителя канала. Запрос достигает начала очереди, и операция ввода-вывода по (4)₂ выполняется в действительности. По завершении ввода-вывода ожидающим процессам посыпается соответствующее сообщение. Этим действием заканчивается обслуживание операции *Read*, указанной в (1).

Простой запрос на чтение приводит, как очевидно, к огромному числу операций. Это цена, которую мы платим за файловую систему общего назначения. Однако в приведенном примере мы попытались использовать большинство путей через эту систему; на практике существует много упрощений и сокращений. Например, когда файл *MyCardFile* открыт и выполнена буферизация, то поиск во внешнем справочнике устраняется, и с помощью эффективного просмотра таблицы в AFD обнаруживается дескриптор необходимого файла; запрос на чтение тогда, по существу, преобразуется в последовательность находления буфера (4). Кроме того, большая часть преобразования исходного запроса на чтение может быть выполнена во время компиляции или ассемблерования, а не во время выполнения.

Файловые системы могут иметь разнообразную структуру, но функции этих систем остаются подобными описанным выше.

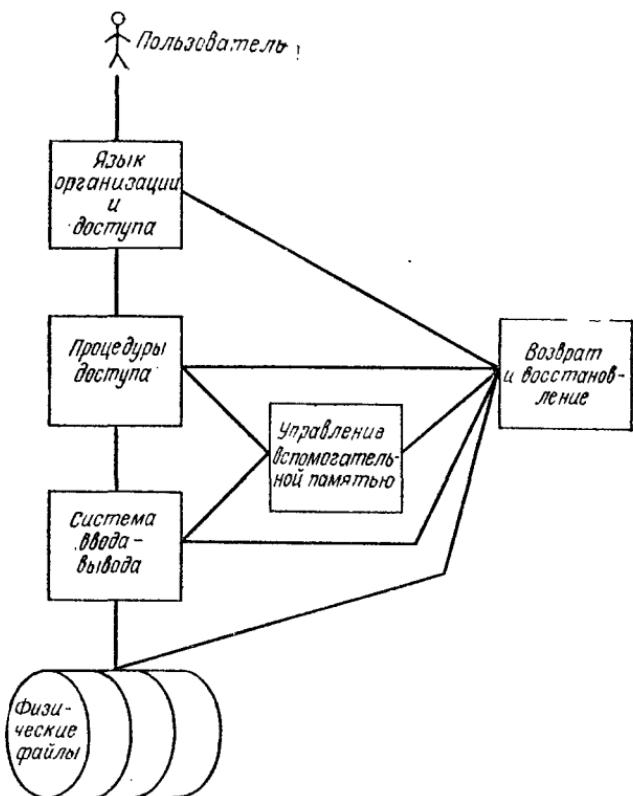


Рис. 9.1. Модули файловой системы.

Одна из возможных структур файловой системы включает следующие пять модулей, связанных между собой так, как это показано на рис. 9.1:

1. *Язык организации и доступа*. Это та часть файловой системы, которая связана с пользователем. В этом компоненте обеспечиваются средства определения различных логических организаций в виртуальном файловом пространстве и конструкции языка для доступа к файлам. Последний включает команды и декларации для создания, разрушения, считывания, записи, модификации и управления доступом к файлам. Во многих системах эти средства охватывают и физический уровень файла так, что пользователи могут обойтись без больших частей системы и до некоторой степени прямо определять физические организации и команды ввода-вывода.

2. *Процедуры доступа*. Они состоят из стандартных программ управления справочниками файлов и поиска в них, от-

крытия и закрытия файлов, отображения символьических имен файлов в их действительные адреса, управления санкционированным доступом к файлам, управления внутренними буферами и генерирования соответствующих программ ввода-вывода. Как видно из этого длинного списка, процедуры доступа составляют самую большую часть большинства систем.

3. *Система ввода-вывода.* Система ввода-вывода отвечает за поддержание очередей запросов на ввод-вывод, за планирование и запуск операций, обработку ошибок ввода-вывода, а также за обслуживание сигналов окончания ввода-вывода.

4. *Управление вспомогательной памятью.* Этот модуль отвечает за учет доступного пространства во вспомогательной памяти и распределение и освобождение блоков вспомогательной памяти по запросам. В сложной системе мы можем также включить сюда средства для перемещения файлов информации по иерархии запоминающих устройств; одним из важных факторов при определении, где хранится файл в данное время, может быть его действительная или ожидаемая активность.

5. *Возврат и восстановление.* Система должна иметь возможность восстанавливаться при аппаратурных и программных ошибках. С этой целью последний компонент обеспечивает восстановление после ошибок файловой системы (и ОС в целом).

В следующих разделах более подробно описаны некоторые из основных функций этих модулей и приемы их использования. Пользовательские „языки“ для доступа и описания файлов рассматриваться не будут.

9.3. Логическая и физическая организации

Несколько общих разновидностей организаций файлов рассматриваются с двух позиций: изучается логическая организация, определяемая пользователем, и соответствующая физическая организация, которая может быть эффективно реализована на устройствах вспомогательной памяти. (Мы будем также использовать термины „виртуальная“ и „реальная“ вместо „логическая“ и „физическая“ соответственно.) Виртуальный файл динамически или статически подразделяется на набор записей $R_0, R_1 \dots$, где запись есть непрерывный блок информации, передаваемый во время логических операций чтения или записи. Аналогично, реальный файл делится на физические записи; размер записи определяется характеристиками запоминающей среды. Физическая запись может состоять из нескольких логических записей, или, наоборот, логическая запись может распространяться на несколько физических.

Последовательные методы

Последовательная организация (*последовательный файл*) — это разновидность организации, при которой записи упорядочены и доступ к ним производится в последовательности, соответствующей упорядочению. Если последней была считана или записана i -я запись, т. е. R_i , то следующей доступной записью автоматически является R_{i+1} . Это самый старый тип структуры и используется он для файлов на магнитной ленте с первых дней возникновения ЭВМ.

Внутри файла записи могут быть *фиксированной* или *переменной длины*. Эти два варианта отражены на рис. 9.2. Файл с записями фиксированной длины имеет записи длиной n единиц; в случае переменной длины каждой записи R_i непосредственно предшествует указатель длины n_i . Чтобы создать эту структуру (*сегмент*) в файле виртуального пространства, достаточно иметь указатель на следующую последовательную запись. Тогда операции чтения и записи между файлом $[F, 0], [F, 1], \dots$ и блоком памяти с начальной ячейкой M могут быть выражены просто:

1. Фиксированная длина:

- (a) $\text{Read}(F, M, r, n); r := r + n;$
- (b) $\text{Write}(F, M, r, n); r := r + n;$

Передача происходит между $[F, r + i]$ и $M + i$, $i = 0, \dots, n - 1$.

2. Переменная длина:

- (a) $\text{Read}(F, T, r, 1);$
 $\text{Read}(F, M, r + 1, T);$
 $r := r + T + 1;$

Ячейка основной памяти T будет содержать длину записи после первой операции *Read*.

- (b) $\text{Write}(F, M, r, M + 1);$
 $r := r + M + 1;$

Предполагается, что длина записи хранится в M .

Можно также *отступить* на запись, однако в этом случае в конце каждой записи переменной длины должно также присутствовать поле длины.

Последовательная структура может быть физически реализована во вспомогательной памяти или *последовательно*, или с помощью *указателей*; мы предполагаем, что используются устройства прямого доступа. По самому простому методу логически непрерывные записи отображаются в физически непрерывные. Эта реализация также показана на рис. 9.2; запись и

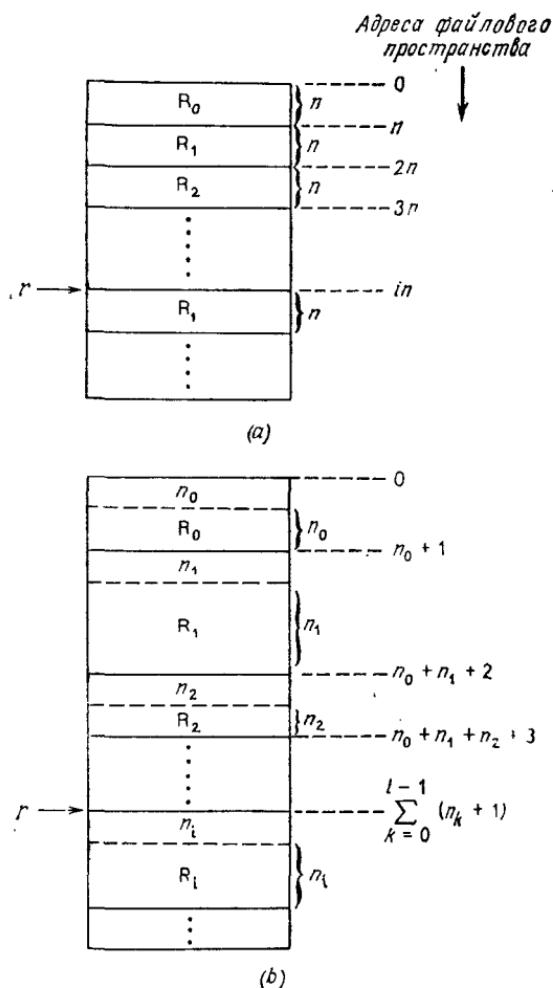


Рис. 9.2. Последовательные файлы: (а) — фиксированная длина; (б) — переменная длина.

Считывание реального файла аналогичны приведенным выше случаям 1 и 2. Записи файла, организованного с помощью указателей, не хранятся в общем случае непрерывно друг за другом. Они могут быть разбросаны по устройству внешней памяти. Каждая запись связана со следующей прямым указателем. Если желательно также перемещение в обратном направлении, то можно воспользоваться обратным указателем; тогда физическая запись R_i содержит указатель p_i к R_{i+1} и указатель q_i к R_{i-1} . Записи легко можно вставлять и удалять внутри структуры с указателями, тем самым расширяя и сокращая файл.

Последовательная реализация не допускает такой гибкости, так как непрерывность записей подразумевает, что (1) заранее должен быть объявлен максимальный размер файла и распределен достаточный блок виешней памяти и (2) вставки и удаления могут происходить только в конце файла. С другой стороны, если используется диск с перемещающейся головкой, то головка теоретически всегда располагается так, чтобы иметь доступ к следующей записи, тогда как в случае файла с указателями перед считыванием требуется установка. Это преимущество может оказаться иллюзорным на практике, если несколько активных файлов разделяют одно и то же устройство. Переходы от виртуального к реальному пространству не так просты, как вытекает из нашего рассмотрения, так как мы игнорировали тот факт, что физические и логические записи бывают обычно разных размеров.

Последовательные методы особенно применимы к файлам, ориентированным только на чтение (входным) или только на запись (выходным), где последовательно читается или пишется соответственно весь файл; они также широко используются в приложениях, связанных с обновлением, которые включают полное сканирование файла, например еженедельный просмотр счетов. Но существует много других ситуаций, когда записи должны быть непосредственно получены в некотором произвольном порядке, а не последовательно. Примерами являются опрос и обновление в реальном времени различных файлов, а также динамическое управление страничной памятью. Здесь подходят методы прямого доступа как для логического, так и для физического пространства; кроме того, если желательно, они также допускают последовательный доступ.

Прямой доступ

Продолжаем считать, что записи файла упорядочены, $R_0, R_1 \dots$, но теперь требуется получить прямой доступ к любой произвольной записи R_i . Предполагается, что каждая запись R_i имеет логическую идентификацию, или *ключ*, заданный числом i ; ключи могут быть результатом преобразования или процедуры сортировки более сложных алфавитно-цифровых ключей. Если файл логически организован, как на рис. 9.2(a), то адрес первого элемента записи R_i равен просто $i \times n$. Структура рис. 9.2(b) неудовлетворительна для прямого доступа к записям переменной длины. (Можно, конечно, просматривать последовательно с опережением каждую запись, пока не будет достигнута желаемая запись, но в общем случае это далеко не эффективно.) Чтобы управлять этим типом файла добавляется *таблица индексов*, как показано на рис. 9.3. Таблица упорядо-

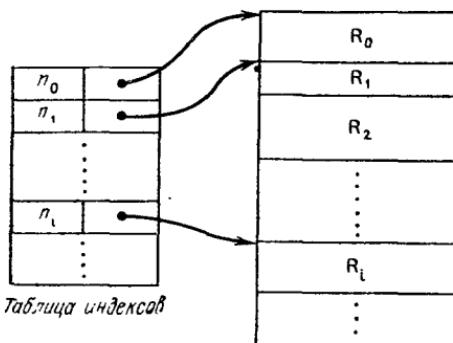


Рис. 9.3. Организация таблицы индексов.

чена таким образом, что i -й элемент содержит длину и адрес записи R_i . Тогда, чтобы прочитать или записать R_i , требуются две операции. Первая — это есть чтение соответствующей записи в таблице индексов и вторая — собственно доступ к R_i .

Если записи хранятся последовательно и непрерывно на физическом носителе, то адрес любой логической записи R_i в файле с записями фиксированной длины может быть получен в принципе вычислением $B + i \times n$, где B — это адрес положения R_0 во вспомогательной памяти. (Это предполагает, что размер физических и логических записей одинаков.) Файлы записей переменной длины и (или) файлы, которые используют организацию с указателями, могут обрабатываться в реальном пространстве с помощью таблицы индексов. Номер логической записи действует как индекс в линейной таблице для нахождения указателя на соответствующие физические записи. Нет необходимости хранить записи непрерывно или с явными указателями, что делает возможным динамическое распределение и освобождение пространства вспомогательной памяти. Этот метод также обеспечивает эффективную реализацию частично заполненных файлов. Каждая запись в таблице индексов может иметь бит, показывающий, содержит ли соответствующая запись информацию или что она не используется; если последнее верно, то нет необходимости распределять пространство записей. К файлам, которые по своей природе являются логически последовательными, также можно обращаться с помощью таблицы индексов; в этом случае осуществляется только последовательное продвижение по таблице.

Таблица индексов может быть организована как файл I записей фиксированной длины в виде дерева (Мэдник и Олсон, 1969). Например, если длина записи есть m и существует $k \leq m$ записей в файле F , индексируемом с помощью таблицы, то в I достаточно одной записи. Для $m < k \leq m^2$ записей в F каждая первая запись I_0 из I потенциально указывает на другую

таблицу длины m , которая в свою очередь содержит адрес записей в F . Адрес i -й записи в F получается из записи $I_r, [i \bmod m]$, где $r = i/m$. Для $k = m^2$ мы имеем $m + 1$ таблиц; I_0 — на первом уровне и I_1, \dots, I_m — на втором уровне дерева. Структура дерева распространяется на дальнейшие уровни для $k > m^2$.

Могут быть построены и другие организации посредством более сложных структур индексных таблиц. Одним из примеров является *ассоциативный* файл, в котором записи доступны непосредственно через поле ключа в записи; здесь ключ может быть „хеширован“ или преобразован для получения элемента в таблице индексов. Конечно, необходимо учесть возможность, когда два или более ключей хешируются на один и тот же элемент.

9.4. Процедуры доступа

9.4.1. Файловые справочники

Все системы имеют набор *справочников* (*словарей*, *каталогов*), которые идентифицируют и определяют местонахождение всех файлов, доступных для пользовательских и системных процессов. Элемент справочника содержит как минимум имя и физический адрес файла или его дескриптора; с другой стороны, в каждом элементе может храниться весь „дескриптор“ файла. Так как справочники сами являются данными, объектом поиска и модификации, то они часто рассматриваются как один файл, по специальному назначения; тогда операции файловой системы можно также использовать и для управления справочником. Мы изучим методы организации справочников и именования файлов.

Наиболее общим видом практической организации справочников является *деревовидная* структура, в которой каждая вершина дерева является справочником (файлом), а каждая ветвь — элемент справочника, который указывает или на другой справочник, или на файл данных. Если файлы данных добавляются к дереву, они будут занимать все листья. Корень дерева называется главным справочником. Эти понятия иллюстрируются рис. 9.4. (Пока не обращайте внимания на пунктирные связи.) Если мы потребуем, чтобы имена всех ветвей (данные и справочники) с *одной и той же* порождающей вершиной не повторялись, тогда для каждого файла существует однозначное имя — символическое имя „пути“ от корня к файлу. Это имя формируется конкатенацией идентификаторов ветвей, начиная с главного справочника и по мере прохода по дереву к желаемому файлу. (Следует заметить, что мы неявно подразумевали, что на данный файл указывает одна и только одна ветвь справочника; иначе символическое имя более не является уникаль-

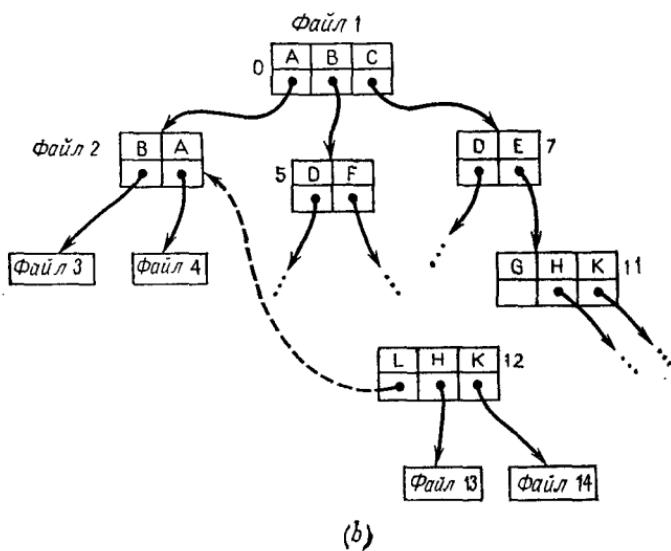
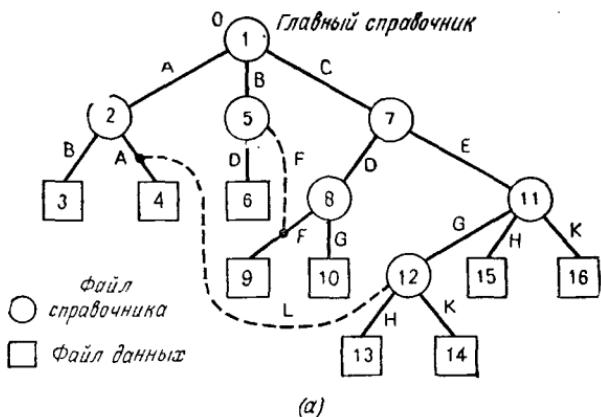


Рис. 9.4. Справочники с древовидной структурой: (а) — структура дерева; (б) — содержимое справочника.

ным — может существовать несколько путей от корня к файлу, — но каждое имя пути все еще определяет один файл.)

Примеры

1. На рис. 9.4(а) файл данных 14 имеет имя *C.E.G.K*. Справочник 8 имеет имя *C.D*.
2. Типичный пример дерева справочников приведен на рис. 9.5. Главный справочник имеет шесть элементов, указывающих на справочники: программ операционной системы (*OS*),

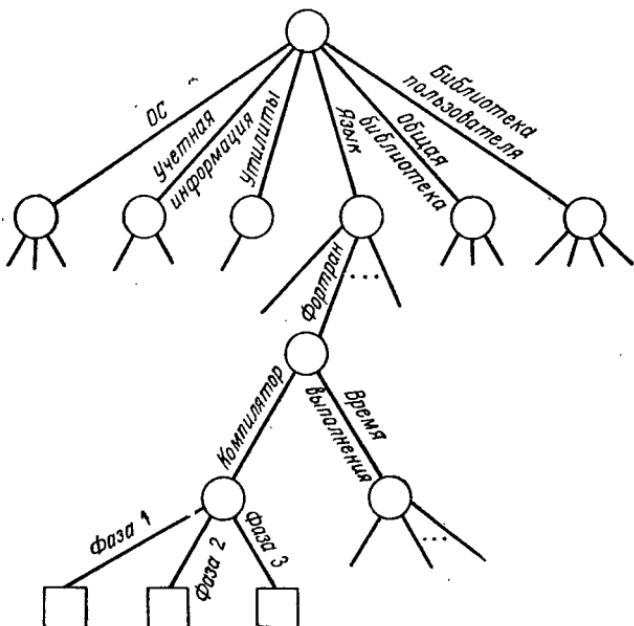


Рис. 9.5. Типичный справочник.

пользователей (*ACCOUNT*); основных утилит, таких, как печать перфокарт и дампы (*UTILITIES*), языковых процессоров (*LANGUAGE*), общих библиотек программ и данных, представленных внешними пользователями (*PUBLIB*), и личных библиотек пользователей (*USERLIB*). Уникальная идентификация для библиотеки во время выполнения, связанной с системой FORTRAN, записывается как *LANGUAGE.FORTRAN.RUN-TIME*.

Рассмотренное правило формирования имени устраивает любые конфликты в связи с идентификацией между пользователями, допуская в то же время почти произвольное назначение имен ветвям. Имя ветви может быть интерпретировано как имя, даваемое файлу пользователем в то время, когда он работает в пределах контекста данного файла. Имя пути в дереве может быть использовано как аргумент при поиске с целью нахождения любого файла данных или справочника файла. Так как справочники являются файлами, то могут быть определены также ограничения на доступ к ним. Тем самым осуществляется управление множеством пользователей, способных модифицировать и извлекать информацию из справочника.

Во многих системах применяется двухуровневая структура справочника; здесь главный справочник указывает на справоч-

ники для различных систем и файлов пользователей. Третий уровень содержит файлы данных. Примером этого типа является файловая система Кембриджского университета на ЭВМ ATLAS (Уилкс, 1968, гл. 6). Этот вид организации, конечно, проще, чем многоуровневый, но он уменьшает гибкость при назначении имени, а также при объединении и управлении использованием связанных файлов. (Почему?) МС более чем с двумя уровнями справочников для файлов имеют IBM OS/360 (IBM, 1965), MULTICS (1967) и TENEX (Боброу и др., 1972). Некоторые интересные аспекты методов, примененных в системе MULTICS, рассматриваются в следующем параграфе.

В системе MULTICS каждый пользователь (процесс) получает рабочий справочник; затем к файлам обращаются относительно текущего рабочего справочника. Например, если рабочий справочник пользователя U есть справочник 7 на рис. 9.4(а), то пользователь идентифицирует файл 16 по имени пути $E.K$. Соглашение о передвижении по дереву допускает идентификацию как наследственных, так и других групп файлов, которые не являются потомками рабочего справочника. Допустим, что значок „звездочка“ (*) обозначает создателя данного файла. Тогда к файлу 6 на рис. 9.4 обращаются по * .B.D по отношению к файлу 7; аналогично, если бы рабочим справочником был номер 12, то файл 9 имел бы имя *. *.D.F. Чтобы увеличить эффективность этих ссылок через справочники, между элементами справочников могут быть созданы *указатели*. Указатель — это элемент справочника, который указывает непосредственно на следующий элемент справочника, а не на файл. Это отражено с помощью пунктирных линий на рис. 9.4; существуют указатели от справочника 12 к элементу A справочника 2 и от справочника 5 к элементу F справочника 8. Следует заметить, что указателям приписаны имена таким же способом, как и ветвям. Тогда имя файла 9 относительно рабочего справочника 5 есть или * .C.D.F., или, используя указатель, просто F ; другой пример — к файлу 4 можно было бы обращаться относительно справочника 11 или по имени *. *.A.A, или по имени $G.L$. Нужно быть осторожным с указателями, потому что они разрушают основную структуру дерева и могут привести к несогласованности его частей; например, если файл 4 и соответствующая ветвь A справочника 2 были удалены, то указатель L в справочнике 12 укажет на несуществующий элемент.

Так как справочники могут содержать много записей и потенциально может быть определено большое число соответствующих им файлов, то большинство справочников хранится во вспомогательной памяти. Для перемещаемых библиотек и автономных систем файлов „локальные“ справочники часто располагаются на том же самом устройстве памяти, что и файлы,

на которые они указывают. Нормальным является, что все главные справочники целиком или частично расположены в основной памяти (временно или постоянно); в любом случае дескриптор главного справочника постоянно находится в основной памяти так, чтобы можно было начать поиск в этом файле.

Упражнение

Допустим, что все справочники файлов имеют следующий вид:

- Каждая запись справочника содержит символическое имя ветви, адрес вспомогательной памяти и длину порожденного файла (справочника или файла данных).
- Первые два слова файла содержат соответственно адрес вспомогательной памяти и длину порождающего справочника.

Предположим, что рабочий справочник D расположен в основной памяти в ячейках $W[i]$, $i = 1, \dots, w$. Напишите булеву процедуру

$\text{SearchDirectory}(\text{Path}, n, \text{Addr});$

которая ищет в дереве справочника файл, идентифицируемый именем пути $\text{Path}[i]$, $i = 1, \dots, n$, относительно D . Программа должна возвращать значение `false`, если требуемый файл не найден; в противном случае она возвращает значение `true` и адрес вспомогательной памяти файла в Addr . Каждый элемент Path может быть «звездочкой» или символическим именем ветви; например, путь `**.B.C.` запоминается как $\text{Path}[1] = \text{Path}[2] = '*'$, $\text{Path}[3] = 'B'$, $\text{Path}[4] = 'C'$. (Допустим, что указатели не используются.)

Используйте в вашей программе следующие процедуры:

(1) $\text{Read}(\text{FileAddr}, M, \text{Length});$

Она читает файл длины Length , размещенный во вспомогательной памяти по адресу FileAddr , в ячейки основной памяти $M[1], M[2], \dots, M[\text{Length}]$.

(2) $\text{Search}(M, \text{Length}, \text{Name}, \text{Addr}, \text{Len});$ (булева процедура)

Она находит поле имени файла, хранящегося в основной памяти $M[i]$, $i = 1, \dots, \text{Length}$, для символического имени Name . При неуспешном завершении возвращает значение `false`; в противном случае — значение `true`, адрес вспомогательной памяти Name в Addr и его длину в Len .

9.4.2. Дескрипторы файлов

Дескриптор файла, связанный с каждым файлом, содержит описательную информацию, требуемую файловой системой, чтобы выполнять функции обслуживания. В частности, это те данные в дескрипторе, которые позволяют переходить от виртуального к реальному файловому пространству (от логических к физическим записям), а также выполнять много других операций. Дескриптор создается тогда, когда файл только создается и обновляется тогда, когда файл перемещается, сокращается, расширяется или когда к нему обращаются.

Если данный файл может быть адресован в системе с помощью только одной записи в справочнике, то самым логичным будет расположить дескриптор внутри самого справочника \blacksquare

виде части каждой записи. На самом деле он часто хранится в отдельной области на устройстве памяти, содержащем соответствующие файлы. Это соглашение делает возможным использование относительно больших дескрипторов и позволяет хранить описание содержимого устройства вспомогательной памяти непосредственно внутри ее самой; несколько записей справочника могут тогда для удобства указывать на один и тот же файл. Обычно дескриптор является резидентным в основной памяти с момента, когда файл стал активным (стал открытым), до его закрытия; в течение этого времени он обычно увеличивается из-за обработки зависимой информации, такой, как буферы и программы доступа.

Фактически во всех файловых системах в дескрипторах содержатся три основных объекта данных: идентификатор файла, сведения о том, где файл хранится, и информация, управляющая доступом.

Идентификатор файла

Этот объект обычно состоит из символического имени N и внутреннего идентификатора I . N может быть любой строкой символов, такой, как имена ветвей в дереве справочников, рассматривавшихся в предыдущем разделе, или оно может быть уникальным символическим именем, таким, как имя пути от главного справочника к названному файлу; одно из преимуществ второй схемы состоит в том, что в данном случае информация об имени помогает как при реконструкции справочника после „системной аварии“, так и при проверке файловой системы на согласованность частей. I предназначен для того, чтобы обеспечить уникальный простой идентификатор для каждого файла, так чтобы можно было легко обнаружить „дескриптор“ и к файлу можно было удобно обращаться. Этот идентификатор может быть действительным адресом дескриптора или парой (i, d) , обозначающей i -й файл на устройстве d , или числом, полученным посредством расчета по некоторому правилу, например идентификатор может быть номером файла 325, или чем-нибудь более скрытым, таким, как время создания файла.

Физический адрес

Здесь определяется местоположение и протяженность файла. Для файлов, записи которых не хранятся последовательно, должны быть определены адрес и длина каждой физической записи — или непосредственно с помощью списка, или косвенным путем, например с помощью указателя на таблицу.

индексов или на первую запись в наборе записей, связанных указателями. Соответствие между логическими записями (непрерывно адресуемыми словами в виртуальном файловом пространстве) и соответствующими физическими адресами записей обеспечивается в этом компоненте или неявно через информацию о физических адресах, или явно, например с помощью таблицы.

Информация управления доступом

Средство, определяющее кто и как может иметь доступ к данному множеству файлов, является одной из центральных и важнейших служб, обеспечиваемых в многопользовательских системах с разделением ресурсов. Информация „кто“ и „как“ хранится в дескрипторе. Файловая система должна обеспечивать строгое соблюдение этих спецификаций управления доступом, и она хранит их в защищенном виде. Этому свойству файлов посвящен следующий раздел.

В дескрипторе содержится множество других полезных данных, например:

1. *История и результаты измерения.* Эта информация может включать дату создания, дату последнего использования или последнего считывания, количество раз, когда файл был открыт, и другие данные о его использовании.

2. *Диспозиция.* Файл может быть *временным* и разрушается при „закрытии“ в конце задания, для которого он был создан, или он может сохраняться неопределенно долго как постоянный файл.

3. *Кодирование информации.* Данные в файле могут быть закодированы как неинтерпретированные *двоичные* строки, причем предполагается, что они должны быть непосредственно загружены в машину, или как *символы* (например, в кодах EBCDIC или USASCII), которые должны быть декодированы и закодированы или распакованы и упакованы во время ввода и вывода соответственно.

4. *Физическая организация.* Например, последовательная, связанная, индексная; записи фиксированной/переменной длины и т. д. Эта информация может быть частью компонента „*физический адрес*“ в дескрипторе.

94.3. Управление доступом

Информация управления доступом определяет тип защиты, назначенной файлу *внутри* вычислительной системы. (Проблемы предохранения вычислительной установки, например кражи информации или радиоподслушивания, не рассматриваются.) Списки „кто“ и „как“ в дескрипторе должны иметь силу во

время нахождения файла в основной памяти, а также во время всех операций ввода-вывода над файлом. Это обеспечивается в файловой системе проведением всех запросов на доступ через программные процедуры управления, которые проверяют их допустимость. Однако после того как файл загружен в основную память, этот последний метод становится непрактичным, так как можно применять весь диапазон операций ЦОУ. Вместо этого данные для управления доступом, находящиеся в дескрипторе, применяются, чтобы установить соответствующие аппаратурные флаги памяти и управления защитой (в разд. 5.2.3 описаны некоторые из этих механизмов); следовательно, этот тип защиты ограничен характеристиками центрального процессора и управления памятью.

Файлы должны быть устойчивы по отношению к нескольким типам умышленных или случайных нарушений защиты. Сюда входит подмена одного пользователя другим, доступ к файлам пользователей, которые не имеют на это права, а также и некорректное использование файлов пользователями с разрешенным доступом, в том числе и владельцами файлов. Данные для управления доступом, определяемые в большинстве систем, могут быть представлены как списки полномочий в виде пар $\langle C, A \rangle$, интерпретируемые как: „Если пользователь отвечает условиям C , он может получить доступ к файлу по пути A “. Условие C , что существует, определяет пользователя, которому разрешен доступ, или группу пользователей (т. е. тех, кто может получить доступ к файлу), а A — атрибуты доступа — показывают, как файл может быть использован.

От атрибутов доступа обычно зависит разрешение выполнить одну или несколько из следующих операций: читать, писать, загрузить для выполнения, сократить (удалить часть файла) или присоединить (добавить данные к файлу). Кроме этого, право изменять содержимое списков полномочий должно быть атрибутом по крайней мере одного пользователя — владельца. Допустимое множество условий в C может быть представлено в виде явного списка пользователей, идентифицируемых, например, по имени или по порядковому номеру, или в виде проверок пароля или конкретного местоположения пользователя.

Несколько более общая схема позволяет владельцам файлов определять произвольные процедуры, которые преграждают все доступы к файлу. Владелец может затем задать любой из разработанных механизмов защиты, который ему требуется, например механизм определения времени суток, в которое определенный класс пользователей может получить доступ к файлам, или вступление в пространный диалог с пользователем файла о допустимости доступа.

Примеры

1. Файловая система MULTICS (MULTICS, 1967) хранит списки управления доступом в побочных записях-ветвях справочников. Атрибуты доступа — это *чтение, запись, выполнение, присоединение и прерывание*. „Прерывание“ применяется для реализации схемы, описанной в последнем параграфе. Если „включен“ атрибут *прерывание*, когда требуется доступ к файлу, то вызывается процедура, которая определяет действующие атрибуты доступа; иначе атрибуты в списке используются прямо. Между файлами справочника и файлами данных сделано разграничение в интерпретации атрибутов; для справочников файлов атрибут *выполнить* определяет разрешение на поиск справочника, тогда как атрибут *читать* определяет разрешение читать только те элементы справочника, которые доступны пользователю.

2. В Кембриджской системе мультидоступа (Уилкс, 1968) пользователи файлов, определенных в единственном справочнике, разделяются на четыре класса: владелец, множество партнеров (определенное владельцем), множество „держателей ключей“ (тех, кто знает пароль, выданный владельцем) и все остальные. Привилегии доступа определяются для каждого класса как глобальные списки „привилегии“ в справочнике владельца. Элемент в списке привилегий представляет выражение в виде

•
если $p_{i_1} \wedge p_{i_2} \wedge \dots \wedge p_{i_k}$ удовлетворяются,
то разрешены типы доступа $a_o a_p a_k a_r$,

где p_i есть условия, такие, как „Имя пользователя есть X “, „Пользователь работает за консолью“ или „Пользователь выдал правильный пароль“, а $a_o a_p a_k$ и a_r указывает на типы доступа, т. е. типы доступа, разрешенные владельцу, партнеру, держателю ключей и кому-нибудь еще соответственно. Каждый тип a_x есть битовая строка, показывающая некоторую комбинацию привилегий: *читать, загрузить для выполнения, писать, уничтожить и изменить привилегии доступа*. Аналогично каждый файл также имеет четыре строки b_o, b_p, b_k, b_r , которые определяют тип доступа, разрешенный каждому классу пользователей к конкретным файлам. Таким образом, запрос на доступ должен проходить как через глобальные ограничения в списках полномочий, так и через локальные ограничения, связанные с каждым файлом. Существует также второй вид глобального списка привилегий, который направлен на создание новых файлов, изменение справочников и изменение списков привилегий.

Существует другой тип защиты, не представленный в информации управления доступом, но который должен быть обес-

печен. Он необходим, когда несколько процессов могут потенциаль но использовать данный файл одновременно. Никаких проблем не возникает, если один и тот же файл в одно и то же время читают несколько пользователей. Однако, если пользователь переходит к процессу записи в файл, желательно, чтобы не были разрешены для этого файла никакие другие доступы на чтение или запись в течение этого рода деятельности; иначе результаты могут быть непредсказуемыми. Это еще один вариант знакомой нам проблемы критической секции, и он может быть решен с помощью методов, рассмотренных нами в гл. 3 (см. также Куртуа и др., 1971). В качестве примера можно привести Кембриджскую файловую систему (Кембридж, 1967), которая гарантирует, что: (1) когда файл открыт для обновления, никакой другой пользователь не может получить к нему доступ, и (2) не могут быть изменены данные управления доступом к файлу, в то время когда файл используется каким-либо способом; параллельное чтение разрешено.

Упражнение

Рассмотрите n циклических процессов ($n > 1$), каждый из которых имеет доступ к общему файлу F . Все процессы выполняют два системных обращения к файлу: *Read* и *Write* для поиска элементов файла F и записи в файл F соответственно. Пусть i -й процесс p_i имеет вид

begin L_i; ... Read; ... Write; ... go to L_i end;

Любое число операций *Read* может выполняться одновременно, но каждая операция *Write* есть критическая секция по отношению как к операции *Read*, так и другой операции *Write*. Более точно каждый элемент a_{uv} следующего ниже массива отвечает на вопрос: «Когда процесс p_i находится в программе u , может ли другой процесс p_j , $j \neq i$, выполнять программу v ?

		<i>v</i>	
		<i>Read</i>	<i>Write</i>
<i>u</i>		<i>Read</i>	Да
	<i>Read</i>	Да	Нет
	<i>Write</i>	Нет	Нет

Используя P и V операции Дейкстры и соответствующие семафоры (гл. 3), расставьте их относительно операций *Read* и *Write* в процессе p_i , чтобы правильно удовлетворить этим ограничениям.

9.4.4. Стандартные программы открытия и закрытия

После того как процесс объявил о своем намерении использовать или создать конкретный файл, для выполнения ряда работ, связанных с инициализацией, обычно вызывается набор процедур *открытия*. Открытие может произойти в момент, когда задание выбрано для загрузки и выполнения в ответ на явную команду *OPEN FILE*, или в более динамическом случае как

побочный эффект первого запроса на операцию с файлом¹⁾). Аналогично, стандартные программы закрытия вызываются при завершении обработки файла. Они обычно выполняются в конце задания или процесса или как результат явной команды *CLOSE FILE*.

Операции закрытия, по существу, освобождают ресурсы, используемые для организации ввода-вывода с этим файлом и переводят файл в неактивное состояние по отношению к этому конкретному пользователю.

Программы открытия отвечают за получение ресурсов, требуемых для организации ввода-вывода с файлом и за перевод файла в активное состояние. Это сводится к выполнению следующих задач:

A₁. Для старых файлов, которые каталогизированы в системе:

1. Найти файл и сделать его доступным для обработки: Просмотреть справочник, чтобы обнаружить файл. Если файл автономный на магнитной ленте или сменном диске, то распределить устройство и физически смонтировать файл.
2. Использовать информацию управления доступом, находящуюся в дескрипторе, чтобы проверить, имеет ли пользователь право на доступ к файлу запрошенным способом.

A₂. Для новых файлов, которые должны быть созданы: Распределить для файла вспомогательную память и (или) устройство (устройства). Может быть зарезервирован блок вспомогательной памяти, ленточное или дисковое устройство.

B. Для старого и нового файлов:

1. Получить основную память, необходимую для буферов ввода-вывода.
2. Загрузить стандартные программы, предназначенные для доступа к файлу и отображения виртуального пространства в реальное.
3. Возможно, сгенерировать „скелет” (заготовку) последовательностей команд ввода-вывода.
4. Вставить данные *A₁* или *A₂*, *B*, а также дескриптор в справочник активного файла (AFD); запись AFD называется также блоком управления файлом и блоком управления данными. Каждый пользователь может иметь личный AFD, или AFD могут быть объединены в единую структуру на уровне системы.

¹⁾ Команда *OPEN FILE* и команда *CLOSE FILE* или явно выдаются пользователем, или генерируются системой.

Если благодаря предварительному открытию другим пользователем файл уже активный, многие из приведенных выше шагов могут быть опущены.

Стандартные программы закрытия совершают работу, прямо противоположную открытию. В частности, закрытие включает следующие операции:

A₁. В случае *временных* файлов, созданных исключительно для текущих задач, и для файла, к которым *никогда* не будет обращений в будущем, разрушить файл, освободив его ресурсы вспомогательной памяти (и автономные) и удалив запись о нем из справочника.

A₂. В случае новых файлов, которые должны быть *сокращены*, т. е. для постоянных новых файлов, вставить соответствующую запись в структуру справочника.

B. Для постоянных старых и новых файлов:

1. Обновить дескриптор файла на основе новых данных, полученных при самом последнем его использовании; сюда могут относиться изменения характеристик файла, таких, как длина, и „история“ файла, к которой относятся, например, время последнего обращения или время последнего изменения.

2. Если необходимо, поставить маркеры конца файла после последней записи в файле.

3. Запомнить идентификатор и положение файла, с тем чтобы можно было в дальнейшем сделать его резервную копию. (Эти процедуры обсуждаются более подробно в разд. 9.7.)

C. Для *всех* файлов.

1. „Перемотать“ файл и, возможно, демонтировать с устройства.

2. Удалить запись о файле из AFD.

3. Освободить буферное пространство и зарезервированные устройства в соответствующих пулах.

9.5. Управление пространством вспомогательной памяти

Мы рассматриваем стратегии размещения и методы учета пространства во вспомогательной памяти прямого доступа; предполагается, что пространство на устройстве *разделено* между несколькими файлами, а не отведено целиком одному файлу. Не удивительно, что некоторые методы управления основной памятью, описанные в гл. 5, применимы с соответствующими модификациями и к случаю вспомогательной памяти.

Основной функцией распределителя является удовлетворение запросов на „блоки“ вспомогательной памяти фиксирован-

ной или переменной длины. Пространство обычно распределяется в единицах блоков *фиксированного размера* из-за фиксированной длины физической записи у многих носителей информации, большого размера доступной памяти и относительной простоты такого подхода. Эффективность ввода-вывода часто является основным критерием при выборе блока из нескольких подходящих свободных блоков; желательно минимизировать задержки от вращения и времена установки (где это имеет смысл), когда к полученным блокам будут обращаться позднее. Таким образом, желательно размещать всю память для одного и того же запроса и одного и того же файла „близко“ друг к другу. Однако блоки *фиксированного размера* не нуждаются в непрерывном физическом размещении, а могут быть связаны или индексированы некоторым способом, как описано в разд. 9.3; распределение и поиск блоков тогда аналогичны тем же действиям в случае *страничной* организации основной памяти, но без постоянной необходимости динамических замен (за исключением, возможно, систем, которые автоматически перемещают файлы по иерархии вспомогательной памяти). Блоки вспомогательной памяти освобождаются, когда файл переводится в автономный режим либо в результате запроса пользователя, либо из-за неактивности файла, либо когда файл сознательно разрушается или сокращается в размере.

Здесь также можно привести аргументы за и против статического распределения памяти по сравнению с динамическим, но они несколько отличаются от тех, что имели место в случае основной памяти. Размер файла часто не известен заранее, так что статическая стратегия распределения должна часто основываться на оценках размеров по верхней границе. Если нельзя допустить больших потерь пространства памяти, то для некоторых файлов по-видимому необходим динамический метод, например ввод и вывод с помощью спулинга. Как мы показали на примере 2 в разд. 8.1, такая стратегия может легко привести к тупиковой ситуации. Например, в подсистеме спулинга все процессы могут оказаться заблокированными, ожидая пространства памяти для вводного и выводного файлов. (Этот тип тупика может быть предотвращен, если пользователь для каждого задания запрашивает максимальное пространство, которое может понадобиться этому заданию. В более общем случае тупик допускается при условии, что он не случается слишком часто. Процедура восстановления обычно должна расформировать входную очередь заданий, дать возможность выполниться одному или нескольким активным заданиям, а затем необходимо вручную перезагрузить старую входную очередь. Более сложным, но удовлетворительным решением было бы начать печатать задание, прежде чем оно завершится.)

Существует несколько способов вести учет доступного пространства. Наиболее очевидный метод — связать вместе все свободные блоки указателями, хранящимися в каждом свободном блоке пространства. Этот способ прост, но явно неэффективен. Чтобы добавить или удалить информацию о свободном пространстве из этих списков, должны быть выполнены многочисленные операции ввода-вывода для нахождения соответствующего числа блоков и для изменения указателей. (Почему?) По этой причине более распространен метод индексных таблиц, подобных описанным в разд. 9.3. Для устройства ведется отдельная таблица или набор таблиц, содержащих информацию обо всем свободном пространстве. Эти таблицы могут быть организованы в форме связанных списков или массивов. В первом случае для каждого блока (фиксированного размера) вспомогательной памяти распределяется одно слово, а все свободные блоки связываются вместе. При такой организации относительно легко освобождать и размещать блоки, но затрачивается много памяти. Существенно меньшее пространство для таблиц потребуется в том случае, если каждый блок будет представлен единственным битом в таблице, причем „1“ означает свободный блок, а „0“ — недоступный. Нахождение свободных блоков занимает больше времени, чем в случае связанных таблиц. Но более быстрое освобождение свободных блоков и их эффективное представление делают битовый метод в общем случае предпочтительным. За исключением метода битовых таблиц, большинство записей о доступном пространстве из-за их большого размера необходимо обычно хранить во вспомогательной памяти; в основной памяти содержатся только „активные“ в данный момент записи.

Записи о доступном и используемом пространстве обеспечивают возможность некоторой полезной избыточности. Могут поддерживаться два отдельных набора таблиц, один — для свободного пространства и другой — для используемого пространства. Это позволяет выполнять некоторые элементарные проверки согласованности файловой системы, когда происходят аппаратурные и программные ошибки. Эти таблицы удобно рассматривать как файлы; к ним, таким образом, можно иметь доступ через команды файловой системы, и они могут появляться в справочнике файла.

Пример

В системе HASP II (Симпсон и др., 1969), широко используемой для вводного и выводного спулинга в IBM OS/360, применяется эффективная схема управления устройствами прямого

доступа (дисками и барабанами). Доступное пространство исчисляется „цилиндрами“. Цилиндр был первоначально определен для дисков с перемещаемыми головками; термин был распространен на барабан и дисковое устройство с фиксированными головками, где он может означать целое устройство или часть его в зависимости от соглашений по адресации. Главный план цилиндров содержит битовую таблицу для каждого устройства, причем каждый бит представляет единственный цилиндр; значение бита „1“ означает, что цилиндр свободен, а значение „0“ указывает, что цилиндр уже был распределен или файлу HASP, или некоторому другому файлу. Этот план содержится в основной памяти. Распределение вводных и выводных файлов, связанных с каждым заданием, содержится в плане цилиндров заданий с одним набором планов на задание; значение бита „1“ в таблице означает, что цилиндр распределен, тогда как „0“ означает, что он свободен. Эти планы хранятся во вспомогательной памяти до тех пор, пока не становятся активными. Когда файл задания динамически требует большего пространства во время вводного или выводного сценинга, задание получает память в единицах цилиндров для файла и в единицах дорожки для каждого запроса. Таким образом, когда за-прашивается дорожка, распределитель или предоставит следующую дорожку на распределенном (для этого файла задания) цилиндре, или, если на цилиндре нет больше свободных дорожек или это первый запрос, для файла задания будет предоставлен новый цилиндр, а запросу — первая дорожка этого цилиндра. Чтобы уменьшить время установки, система систематически пытается распределять цилиндр, ближайший к (или равный) последнему цилинду, к которому обращались на данном устройстве. Пространство освобождается сразу для целого файла; это выполняется просто с помощью операции ИЛИ над планом задания и главным планом цилиндров.

Существует еще один компонент подсистемы управления вспомогательной памятью, который должен быть тщательно спроектирован. Так как память ограничена по емкости, то должны быть обеспечены средства для перевода неактивных файлов в автономное состояние, например на магнитную ленту, а также для пересылок файлов по иерархии устройств ЗУ, когда таковая существует. Как правило, данные для перемещения включают информацию о текущем местонахождении файлов в системе, приоритеты пользователей файлов, относительную активность файлов и размеры файлов. В дополнение к стандартным программам перемещения существует необходимость копирования файлов для обеспечения резервирования (поддержки).

9.6. Иерархическая модель для файловых систем

Файловая система, представленная в разд. 9.2, приблизительно соответствует модели, описанной в работах (Мэдник, 1968) и (Мэдник и Олсон, 1969). Такая модель делает возможным, в принципе, систематически проектировать, создавать и понимать эти сложные системы с помощью разбиения их на управляемые замкнутые компоненты. В этом разделе данная модель рассматривается более детально. Мы придерживаемся той точки зрения, что обычный пользователь должен иметь дело только с виртуальными файлами, структура и особенности доступа которых отвечают его целям, и что он должен быть изолирован от машинно-зависимых объектов, таких, как конкретное устройство, содержащее его файл, алгоритмы распределения памяти, программные механизмы доступа, блокирование и буферизация; за все это отвечает файловая система.

Файловую систему можно разложить на ряд уровней, входящих в состав иерархической структуры, от самого низкого уровня реальной аппаратуры и планировщиков ввода-вывода до пользовательских интерфейсов на самом абстрактном уровне. Это согласуется с нашим описанием систем в разд. 1.4.1 и с методологией проектирования, описанной в разд. 4.5. Каждый уровень представляет более абстрактную машину (в этом случае „файловую“ машину) и реализуется набором процессов; связи ограничены так, чтобы любой процесс на уровне k мог посыпать сообщения только другим процессам на том же самом уровне, на уровне $k+1$ и на уровне $k-1$ (рис. 9.6). Иерархическая структура полезна потому, что она позволяет сконцентрировать усилия на проектировании отдельных процессов с четко определенными задачами и каналами связи с другими процессами.

Как показано на рис. 9.7, модель состоит из шести функциональных уровней. Ниже мы определим внутренние задачи каждого уровня, переходя от интерфейса между пользователем и виртуальных файлов вниз к аппаратуре:

L6. Методы логической организации. Уровень интерфейса пользователя предусматривает стандартные программы и процессы для наложения логической структуры на неформатированные виртуальные файлы (разд. 9.3). С каждым типом организации обычно бывает связана независимая стандартная программа или модуль.

L5. Поиск по справочнику. Первичной функцией процессов на этом уровне является преобразование символьических имен файла в идентификаторы, которые в свою очередь указывают на точное положение или файла (его дескриптора), или, возможно, таблицы, содержащей эту информацию. Для выполнения

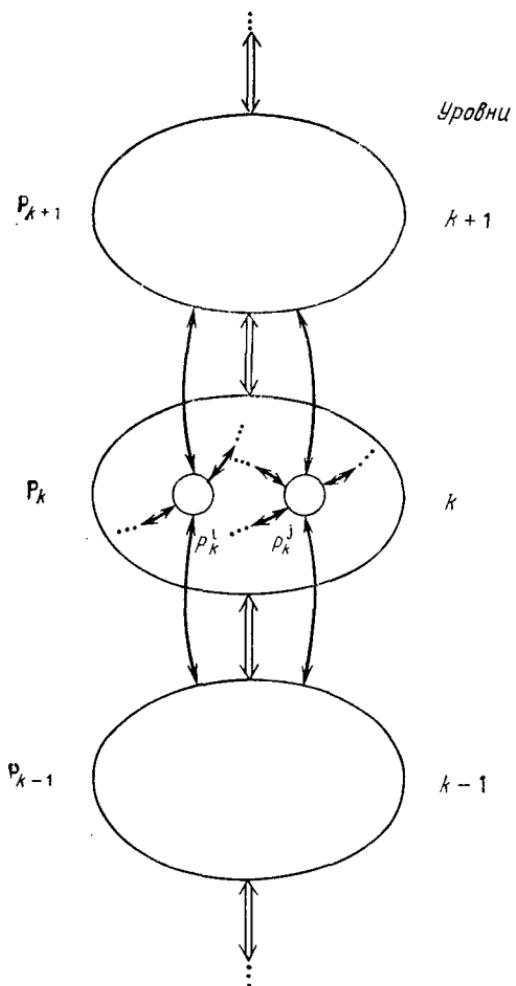


Рис. 9.6. Иерархия сообщающихся процессов.

этого преобразования производится поиск в справочнике записи об интересующем нас файле (разд. 9.4.1).

L4. Базовая файловая система. Эта часть активизирует и деактивизирует файлы с помощью вызова стандартных программ открытия и закрытия, она также, если необходимо, отвечает за проверку прав доступа пользователя при каждом запросе к файлу. Первой задачей является нахождение дескриптора для нужного файла (разд. 9.4.2, 9.4.3, 9.4.4).

L3. Методы физической организации. Исходный запрос пользователя на доступ по определенным адресам логического файла преобразуется в запросы по физическим адресам вспомога-

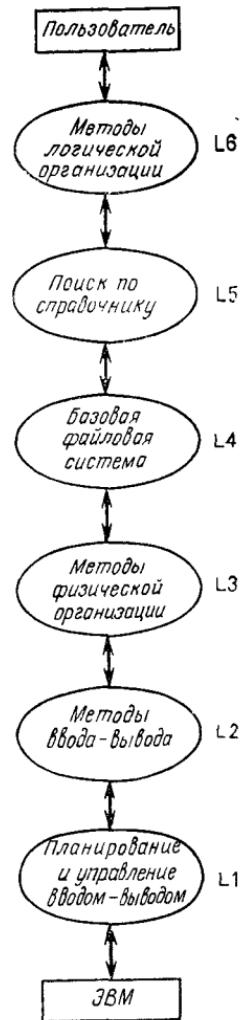


Рис. 9.7. Модель файловой системы.

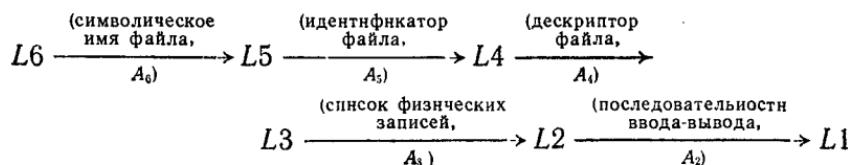
тельной памяти, отражающие действительное положение и организацию желаемых записей (разд. 9.3). На этом уровне рассматривается также распределение вспомогательной памяти (разд. 9.5) и буферов в основной памяти.

L2. Методы ввода-вывода. Запрашиваемая операция (операции) и физическая запись (записи) преобразуются в соответствующие последовательности команд ввода-вывода, команд канала и приказов контроллеру. Выполняются локальные оптимизации, например, с целью минимизировать задержку от вращения в пределах этих операций (процессы устройств, описанные

в разд. 7.3, являются прототипами процессов, проходящих на этом уровне).

L1. Планирование и управление вводом-выводом. На этом уровне происходит действительная работа с очередями, планирование, запуск и контроль в отношении всех запросов на ввод-вывод. Этот уровень непосредственно „связан“ с аппаратурой ввода-вывода в вычислительной системе. Здесь также выполняется основное обслуживание прерываний ввода-вывода.

Одной из причин для такого конкретного выбора уровней и функций является простота связи, которая существует между уровнями. Информация, проходящая вниз от пользователя через последовательные уровни, — это в основном запросы на выполнение работы и следующие данные:



где A_i представляет другие параметры, такие, как тип операции, длина и адрес в основной памяти.

Сообщения, проходящие вверх по иерархии, обычно являются сообщениями о завершении. Части этой иерархии внутри некоторой конкретной системы часто могут пропускаться пользователями специальных классов и (или) для определенных файлов; например, некоторые супервизорные программы могут иметь непосредственный доступ к базовой файловой системе или к модулям планирования и управления вводом-выводом.

Пример

Рассмотрите снова пример чтения файла с перфокарт, приведенный ранее в разд. 9.2. Соответствие между различными операциями, выполняемыми в ответ на запрос пользователя *ReadCard* и уровни модели таковы:

$$(2) \leftrightarrow L6, (3)_1 \leftrightarrow L5, ((3)_2, (3)_4) \leftrightarrow L4, (4)_1 \leftrightarrow L3, \\ (4)_2 \leftrightarrow L2.$$

Модель предложена относительно недавно и, очевидно, требует дальнейшей разработки и уточнения. Тем не менее в настоящее время она полезна из педагогических соображений для систематического изучения компонентов и задач файловых систем и может оказаться более полезной при реализации систем. Одним из аспектов проектирования файловых систем, который

не включен в модель, является резервирование и восстановление при ошибках. В следующем разделе обсуждаются некоторые детали этой важной проблемы.

Упражнение

Выберите мультипрограммную ОС, которая вам знакома, и опишите ее файловую систему в терминах шести уровней, представленных в этом разделе. Где она согласуется и где она отклоняется от модели?

9.7. Восстановление при отказах системы

Количество аппаратурных и программных ошибок может быть сокращено созданием надежных и часто избыточных машинных компонентов, а также посредством систематического логического проектирования, программирования и отладки программ. К сожалению, отказы или „поломки“ все же гарантированы в большинстве сложных систем. Это могут быть катастрофические отказы, которые, по существу, разрушают жизненно важные части МС, и для восстановления системы требуются большие усилия, или отказы могут быть менее серьезными и локализованными в небольшом количестве таблиц и (или) программ. Модули для анализа и восстановления после таких ошибок должны быть составной частью любого системного проекта, а не (как это обычно делается) некоторой „заплатой“, которая ставится после того как остальные части операционной системы уже реализованы. Существует необходимость в основном в „элегантных“ методах восстановления, которые прозрачны для пользователя, действуют быстро и требуют мало работы в реальном времени от персонала, обслуживающего систему во время отказа; примером крайне неэлегантной процедуры восстановления является простая перезагрузка самой последней версии этой ОС, обычно хранящейся автономно на ленте или диске, после которой пользователи должны переопределить *все* свои файлы. (Для вычислительного центра это хороший способ потерять всех своих заказчиков.) Проблему восстановления стоит рассмотреть в этой главе, так как с точки зрения восстановления в системе файловая система содержит наиболее важную информацию — справочники выполняющихся заданий, информацию о положении всех программ и файлов данных, подробности распределения автономной и неавтономной памяти, учетные записи, а также другие данные, имеющие критическое значение.

Вначале кратко рассмотрим причины и следствия системных отказов. Аппаратурные ошибки могут быть следствием отказов в любых компонентах ЭВМ, а также в соединяющих их линиях связи. Ошибка может вызвать полный выход из строя такого

компоненты, как дисковый механизм, или она может проявиться более неуловимо как невоспроизводимый переход, например, или нарушение четности в ЗУ, вызванное колебанием питания. Отказы аппаратуры, как правило, выявляются самой аппаратурой и во многих случаях могут быть локализованы в пределах небольшой области; примером является механическая поломка периферийного устройства, такого, как устройство считывания с перфокарт. Часто, однако, ситуация может быть гораздо более серьезной и вызывать многие из проблем, которые будут рассматриваться ниже применительно к программным ошибкам; к этой категории относятся электронное разрушение информации в кубе ЗУ или повреждение частей диска из-за неправильного функционирования головок считывания/записи или подвижных рычагов.

В системных и пользовательских программах редко (если вообще когда-либо) отсутствуют ошибки. Типичные ошибки программирования — это передача управления по неправильному, возможно, несуществующему адресу, ошибки в адресации операндов, приводящие к нахождению и модификации неправильных частей данных, и „бесконечное“ зацикливание. Тщательное проектирование систем и ЭВМ с соответствующими аппаратурными механизмами защиты должно привести к тому, что ошибки пользователя никак не могут воздействовать на ОС и на других пользователей¹), попытки обратиться к данным или передать управление памяти, выделенной системе или другим заданиям пользователей, обычно выявляются аппаратурой, тогда как неправильные вызовы системных средств, например через механизм прерываний, обнаруживаются системными программами. Таким образом, отказы программного обеспечения могут быть следствием главным образом ошибок в системных программах (и в проекте системы), а не ошибок пользователей. Последствиями этих ошибок могут быть:

1. Полное разрушение функционально важных таблиц, таких, как очереди процессов или активные справочники.
2. Появление таких адресов и указателей, например, в справочниках файлов, которые не связаны с допустимыми элементами списка.
3. Неправильные списки ресурсов, таких, как таблицы свободного пространства во вспомогательной памяти.
4. Даже запись или чтение неверных данных из неверных файлов.

¹⁾ Тем не менее для многих студентов, изучающих вычислительную технику, все еще является вопросом чести обнаружить слабое место в защите и прорваться в систему.

Обнаружение причины и следствий отказа системы в общем случае является очень трудной задачей, так как к тому времени, когда отказ становится очевидным, выполнение программы обычно уходит далеко от неправильного участка. Существует несколько общих правил, которые можно предложить по крайней мере для изоляции указанных последствий. Общий метод заключается в том, чтобы проверить все системные структуры данных на согласованность. Примерами того, как может быть проверена согласованность, являются:

1. *Прослеживание адресов в таблицах.* Обычно набор адресов, которые могут определяться указателем, фиксирован и относительно мал; кроме того, каждый „узел“ структуры списка имеет некоторую идентифицирующую информацию в заголовке, которая может быть проверена. Например, записи в справочнике файла должны указывать на другие справочники или файлы; а структура указателя обычно представляется деревом с файлами данных в качестве листьев; последние могут быть проверены прослеживанием указателей. Другой очевидный пример найден в двусвязных списках, где последовательные прямые и обратные указатели должны определять одну и ту же запись. Правильные адреса вспомогательной памяти в таблицах также обычно могут находиться только в пределах небольшого диапазона возможных адресов.

2. *Использование избыточной информации.* Для целей нормальной обработки часто бывает удобно поддерживать несколько копий одной и той же информации, но в различных формах. Например, структура данных о ресурсах может содержать детали распределения, такие, как имя каждого процесса, владеющего некоторым ресурсом, в то время как дескрипторы процесса могут избыточно содержать список всех ресурсов, связанных с процессом; списки доступного и используемого пространства, упомянутые в разд. 9.5, также дублируют одну и ту же информацию. Эта избыточность может также быть специально предусмотрена для проверки согласованности в системе.

3. *Применение методов контрольной суммы.* Бухгалтеры уже давно используют контрольные суммы и смешанные итоги, чтобы обеспечить независимую проверку правильности таблицы платежей. Идея заключается в том, чтобы объединить все данные в таблице или файле в единственное число, например или путем суммирования, или с помощью операции „исключающее ИЛИ“ или некоторой другой; подсчет выполняется, когда таблица создается впервые и всякий раз, когда она изменяется. Когда к таблице или файлу получают доступ, контрольная сумма подсчитывается снова, чтобы убедиться в том, что присутствуют все данные.

Существуют много других примеров; многие из них попадают в пределы обычных контрольных функций программ, например проверка того, что счетчик числа буферов неотрицательный.

Локализованные аппаратные поломки, которые не привели к порче программного обеспечения и которые не существенны для непрерывной работы ЭВМ, обычно вызывают удаление записей об этом аппаратном ресурсе из структур данных и, возможно, замену его эквивалентным ресурсом; к этому классу относится поломка периферийного устройства, не имеющего особой важности. Чтобы добиться элегантного восстановления при более серьезных проблемах с аппаратурой или при отказах программного обеспечения, требуется наличие актуальных копий операционной системы, файловой системы и даже содержимого основной памяти. Серии восстановлений файлов являются основой всех процедур восстановления. В этом случае отказы обрабатываются посредством выборочной загрузки резервных копий файлов в попытке воссоздать всю систему в таком состоянии, в каком она была непосредственно перед отказом¹⁾). Восстановление при катастрофических „поломках“ может быть выполнено с помощью универсальной, но длинной процедуры перезагрузки, тогда как менее серьезные отказы рассматриваются более частным способом в надежде добиться более быстрого и более полного восстановления.

Процедуры резервирования, которые выполняют *дамп* частей системы, должны быть как удобными, так и эффективными. Делать дамп всего набора информации в режиме „в линию“ в выбранные интервалы времени в общем случае является недовлетворительным решением, так как это занимает слишком много времени и не может выполняться достаточно часто, чтобы обеспечивать актуальные копии. При стандартном подходе практикуют два типа дампов для резервирования: *выборочные инкрементные дампы*, получаемые через относительно небольшие интервалы времени, и более полные дампы в *контрольной точке*, производимые через более длинные интервалы времени.

Икрементные дампы копируют все файлы, которые были или *модифицированы*, или *созданы* со времени получения последнего такого дампа; файлы, к которым имелся доступ только для чтения, в дамп не включаются. Это гарантирует, что всегда доступна самая последняя версия любого файла. Дампирование

¹⁾ Следует заметить, что полное восстановление системы *непосредственно* перед отказом впоследствии приведет к *такой же* катастрофе; таким образом, ошибка должна быть найдена, а затем исправлена или обойдена, или же в случае более «хитрых» или неустойчивых ошибок необходимо менее полное восстановление актуального состояния, с тем чтобы ОС могла работать в то время как проводится достаточно продолжительное изучение системы.

может запускаться через регулярные фиксированные интервалы, скажем каждые несколько минут или часов, когда системе нечего делать, почти всегда в конце каждого задания или фазы в пределах задания, или же в случае комбинации этих событий; вторая схема несколько опасна сама по себе, так как не может быть выдано никаких гарантий относительно периодичности дампа. В любом случае должна поддерживаться некоторая запись об измененных или созданных файлах. Цель дампов контрольной точки — дать возможность быстро загрузить относительно недавнюю версию системы так, чтобы задания могли быть запущены и дальнейшее восстановление могло бы производиться в более медленном темпе. Обычно этот дамп копирует *самые последние использованные* части системы — как системные, так и пользовательские файлы. Полный цикл дампов в контрольной точке может выполняться приблизительно через каждые несколько дней, но может быть запрограммирован так, чтобы копировать небольшие порции через меньшие интервалы времени. Записи о резервных копиях файлов, содержащие такие сведения, как время, тип, и положение дампа, могут быть удобно включены в файловые справочники; другими словами, дампы рассматриваются как файлы. Конечно, мы здесь сталкиваемся со знакомой проблемой: Что случится, если разрушится справочник? Ясно, что имеется необходимость в автономной записи справочников файла, содержащих резервные копии, но, так как эти справочники часто изменяются, они появляются в инкрементных дампах, и содержимое справочников сохраняется.

Если имеются инкрементные файлы и файлы резервных копий, то восстановление при катастрофических отказах происходит следующим образом:

1. Перезагрузить систему из самого последнего инкрементного дампа. Перезагруженная система теперь используется для управления оставшимися действиями восстановления, и одновременно запускаются другие задания пользователей.

2. Начиная с самого последнего инкрементного дампа, идти назад во времени по инкрементным дампам, восстанавливая системные и пользовательские файлы и справочники.

В настоящее время невозможно предложить универсальную процедуру для восстановления системы после катастрофических отказов. Однако если можно решить, что ошибки затрагивают небольшой набор файлов и таблиц, то последние часто можно восстановить так, чтобы ошибки повлияли на небольшую группу пользователей или вообще не повлияли на них, и обеспечить почти немедленное продолжение системных операций. Например, если отказ влияет только на единственное задание

или текущую смесь заданий, то можно просто удалить задание (задания) и перезапустить его (их); если файлы были модифицированы во время первого неудачного выполнения, тогда или пользователи должны быть оповещены о, возможно, „плохом“ файле, или, что более желательно, автоматически должен быть получен самый последний файл из одного из дампов резервной копии. Аналогично, если при проверке после отказа оказывается ошибочным только небольшое число системных файлов или таблиц, то возможно исправить их и продолжить работу. В качестве одного из примеров могут быть рассмотрены файлы и таблицы в оперативной памяти, относящиеся к учетной информации о пользователях; если они частично или полностью разрушены в результате системной ошибки, они могут быть легко воссозданы из файлов резервной копии, за исключением информации о текущем задании. Другой пример —несогласованность в списках доступного и используемого пространства вторичной памяти; списки могут быть перестроены с помощью прослеживания структур справочников.

Примеры

1. В процедурах получения резервной копии Кембриджской системы мультидоступа (Уилкс, 1968) применяются два набора лент — „первичный“ циклический набор S_1 для частого инкрементного дампирования и „вторичный“ циклический набор S_2 для более полных, но менее частых дампов; каждый из наборов S_1 и S_2 используется циклически. Для инкрементных дампов выбирается следующая лента T в S_1 , и все новые и модифицированные файлы копируются в T , а также любые файлы на ленте T (из последнего цикла), которых еще нет в S_2 . Во вторичном цикле выбирается следующая лента U и S_2 , и копируются все файлы на U , которых еще нет в S_2 , а также предыдущие файлы на U , которые еще имеют постоянный статус.

2. Файловая система в MULTICS (1967) включает инкрементное, а также системное и пользовательское дампирование в контрольных точках, причем копирование делается дважды для большей надежности. Контрольные точки систем состоят из текущей учетной информации, справочника, а также файлов компонентов ОС, тогда как пользовательские контрольные точки дампируют все сегменты, к которым были обращения с момента последней контрольной точки. В дополнение к процессу перезагрузки в случае катастрофических ошибок, MULTICS имеет процедуру „спасения“ в режиме ON-LINE, которая пытается исправить менее серьезные отказы, обеспечивая согласованность справочников и таблиц пространства.

3. Система спулинга HASP II (Симпсон и др., 1969) в IBM OS/360 (см. также пример в разд. 9.5) копирует резервную информацию при завершении каждого шага каждого задания, а также с заранее определенным шагом по времени, приблизительно каждую минуту. В дампируемые данные включаются таблица задания, распределение цилиндров диска, а также таблица контрольных точек печати, которая допускает „горячий“ (быстрый) старт заданий, которые находятся в состоянии выдачи на печать.

Упражнение

Изучите системные таблицы, структуры данных и справочники в некоторых МС и разработайте каталог тестов согласованности для этих структур. Какую дополнительную информацию для проверки вы бы рекомендовали?

Приложение Проектирование мультипрограммной системы

A1. Введение

В приложении описывается наглядный проект, предлагающий разработку и реализацию мультипрограммной операционной системы (МОС) для гипотетической вычислительной машины, модель которой легко можно построить (Шоу и Вайдерман, 1971). Цель — собрать воедино и применить в почти реальном сочетании некоторые из концепций и средств, рассмотренных в этой книге. В частности, проектировщик или разработчик должен иметь дело непосредственно с проблемами ввода-вывода, обработки прерываний, синхронизацией процессов, планированием, управлением основной и вспомогательной памятью, структурами данных процессов и ресурсов, системной организацией.

Мы полагаем, что проект будет программираться для большой центральной машины (система „хозяин“), которая, с одной стороны, не позволяет пользователям вмешиваться в операционную систему или машинные ресурсы, но, с другой стороны, обеспечивает полный набор сервисных средств, включая файловые средства, средства отладки и хороший язык высокого уровня. Общая стратегия — создать модель гипотетической машины на „машине-хозяине“ и написать МОС для этой модели. Операционная система МОС и имитатор будут содержать приблизительно от 1000 до 1200 карт программы, причем большая их часть относится к МОС. Проект может быть выполнен приблизительно в течение двух месяцев студентами, несущими при этом нормальную академическую нагрузку.

Характеристики и компоненты вычислительной машины для работы МОС определены в следующем разделе, в разделе А3 описывается формат заданий пользователей. Путь следования задания пользователя через систему, а также функции и главные компоненты МОС описаны в разделе А4. Затем в следующем разделе (А5) приводятся подробные требования, предъявляемые к проекту. В заключительном разделе А6 описаны некоторые ограничения.

A2. Машинные спецификации

Вычислительная машина для МОС описывается с двух позиций: как „виртуальная“ машина, воспринимаемая типичным

пользователем, и как „реальная” машина, используемая самой МОС и лицами, занятыми ее проектированием и реализацией.

1. Виртуальная машина

Виртуальная машина, как она воспринимается обычным пользователем, показана на рис. А.1. Максимальный размер памяти 100 слов, адресуемых от 00 до 99; каждое слово разделено на четыре одиобайтовых элемента, где байт может содержать любой символ, приемлемый для „машины-хозяина”. Центральный процессор (ЦОУ) имеет три основных регистра: четырехбайтовый общий регистр *R*, однобайтовый „булевский” триггер *C*, который может находиться либо в состоянии „*T*” (true), либо в „*F*” (false), и двухбайтовый счетчик команд *IC*.

Слово памяти может быть интерпретировано как команда или слово данных. Код операции в команде занимает два старших байта слова, а адреса operandов помещаются в двух младших байтах. В табл. А.1 приведены форматы и интерпретация каждой команды. Заметим, что команда (*CD*) читает только первые 40 колонок карты и что команда вывода (*PD*) печатает новую строку из 40 символов. Первая команда программы должна всегда размещаться в ячейке 00.

Для такой простой машины может быть быстро написан пакет программ с ограниченными вычислениями, с ограниченным вводом-выводом и программ, сбалансированных в этих отношениях. Почти наверняка можно сказать, что будут допущены программные ошибки обычного вида. (Обе эти характеристики желательны, так как МОС должна быть в состоянии обрабатывать разнообразные задания и ошибки пользователя.)

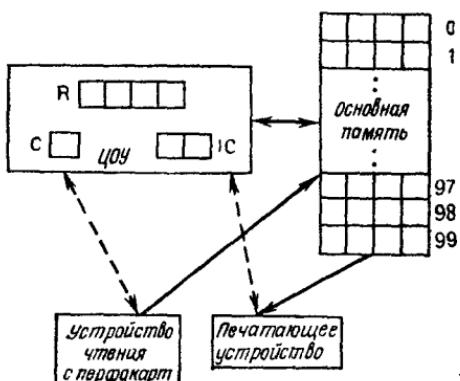


Рис. А.1. Виртуальная машина пользователя.

Таблица A.1. Набор команд виртуальной машины

Команда		Интерпретация
Операция	Операнд	
<i>LR</i>	$x_1 x_2$	$R := [a];$
<i>SR</i>	$x_1 x_2$	$a := R;$
<i>CR</i>	$x_1 x_2$	$\text{if } R = [a] \text{ then } C := 'T' \text{ else } C := 'F';$
<i>BT</i>	$x_1 x_2$	$\text{if } C = 'T' \text{ then } IC := a;$
<i>GD</i>	$x_1 x_2$	<i>Read</i> ($[\beta + i]$, $i = 0, \dots, 9$);
<i>PD</i>	$x_1 x_2$	<i>Print</i> ($[\beta + i]$, $i = 0, \dots, 9$);
<i>H</i>		<i>halt</i>

Примечания: 1. $x_1, x_2 \in \{0, 1, \dots, 9\}$
 2. $a = 10x_1 + x_2$
 3. $[a]$ означает «содержимое ячейки a »
 4. $\beta = 10x_1$

2. Реальная машина

(а) Компоненты

На рис. А.2 приведена схема реальной машины. ЦОУ может работать либо в режиме *ведущего*, либо в режиме *ведомого*. В режиме ведущего команды из супервизорной памяти обрабатываются непосредственно процессором языка высокого уровня (HLP); в режиме ведомого HLP интерпретирует „микропрограмму” в памяти типа „только для чтения”, которая имитирует

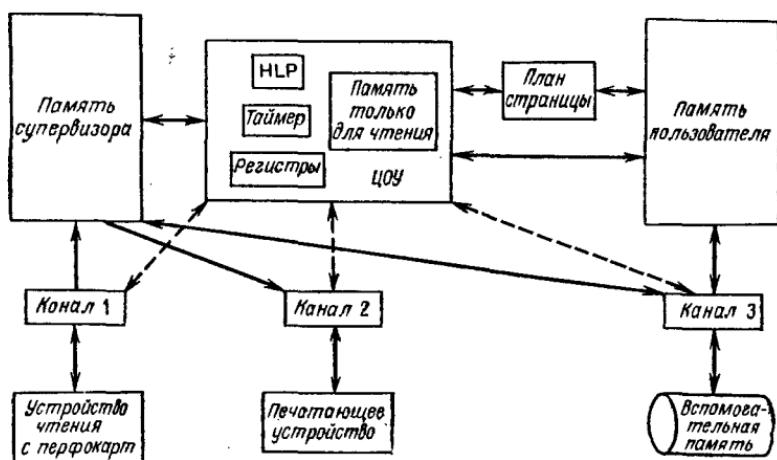


Рис. А.2. Реальная машина.

(моделирует) ЦОУ виртуальной машины и имеет доступ к программам виртуальной машины в памяти пользователя через механизм страничной организации. НЛР — это любой подходящий и доступный процессор для языка высокого уровня. (Эта схема позволяет программировать эмулятор виртуальной машины и МОС на языке высокого уровня, имеющемуся на „машине-хозяине”, и в то же время обеспечивается определенное соответствие реальным вычислительным машинам.) Интерес представляют следующие регистры ЦОУ:

C: однобайтовый „булевский” триггер,

R: четырехбайтовый общий регистр,

IC: двухбайтовый счетчик ячеек виртуальной машины,

PI, SI, IOI, TI: четыре регистра прерывания,

PTR: четырехбайтовый регистр таблицы страниц,

CHST[i], $i = 1, 2, 3$: три регистра состояния канала,

MODE: режим ЦОУ ведущего или ведомого.

Память пользователя состоит из 300 четырехбайтовых слов, адресуемых от 000 до 299. Для целей страничной организации она разделена на 30 блоков по десять слов. Супервизорная память определяется как такой объем памяти, который требуется для МОС.

В качестве вспомогательной памяти служит высокоскоростной барабан, имеющий 100 дорожек, причем на каждой дорожке размещается 10 четырехбайтовых слов. Пересылка 10 слов на дорожку или с дорожки осуществляется за одну единицу времени. (Время задержки от вращения игнорируется.)

Устройство чтения с перфокарт и печатающее устройство осуществляют ввод и вывод одной записи за три единицы времени. Эти устройства имеют те же характеристики, что и устройства виртуальной машины, т. е. при выполнении чтения и записи соответственно пересыпается 40 байтов (10 слов) информации или из первых 40 колонок карты, или в первые 40 позиций печати.

Каналы 1 и 2 соединяют периферийные устройства с супервизорной памятью, а канал 3 — вспомогательную память с супервизорной и пользовательской памятью.

(б) Работа в режиме ведомого

При работе в режиме ведомого память пользователя адресуется посредством аппаратуры страничной организации. Регистр *PTR* содержит длину и адрес базы таблицы страниц для пользовательского процесса, развивающегося в текущий момент. Четыре байта $a_0a_1a_2a_3$ в *PTR* интерпретируются следующим образом: a_1 — это длина таблицы страниц минус 1, а $10a_2 + a_3$ —

это номер блока памяти пользователя, в котором расположена таблица страниц, где a_1a_2 и a_3 — целые числа.

Двухзначный адрес команды или операнда, $x_1 x_2$, в виртуальном пространстве отображается аппаратурой настройки в реальный адрес памяти пользователя:

$$10[10(10a_2 + a_3) + x_1] + x_2,$$

где $[\alpha]$ означает „содержимое по адресу α ”, и предполагается, что $x_1 \leq a_1$. Требуется, чтобы все страницы процесса были бы загружены в память пользователя перед выполнением.

Предполагается, что каждая команда виртуальной машины эмулируется за одну единицу времени. Все прерывания, имеющие место во время работы в режиме ведомого, отрабатываются в конце командных циклов и вызывают переключения в режим ведущего. Операции GD , PD и H приводят к прерываниям супервизорного типа, т. е. к „вызовам супервизора”. Прерывание программного типа возникает, если эмулятор воспринимает недопустимый код операции или если $x_1 > a_1$ при составлении перемещения (недопустимый адрес виртуального пространства).

(с) Работа в режиме ведущего

Программы в режиме ведущего, располагающиеся в супервизорной памяти, имеют доступ к памяти пользователя и регистрам ЦОУ. В режиме ведущего ЦОУ *непрерываемо*; однако, если происходит событие, вызывающее прерывание от таймера или ввода-вывода, устанавливается соответствующий регистр прерываний. Регистры прерываний могут быть опрошены и сброшены при помощи команды $Test(x)$, которая возвращает некоторое значение и действует следующим образом:

```
if  $x = 1$  then begin  $Test := IOI$ ;  $IOI := 0$  end else
if  $x = 2$  then begin  $Test := PI$ ;  $PI := 0$  end else
if  $x = 3$  then begin  $Test := SI$ ;  $SI := 0$  end else
if  $x = 4$  then begin  $Test := TI$ ;  $TI := 0$  end else
if  $(IOI + PI + SI + TI) > 0$  then  $Test := 1$  else  $Test := 0$ ;
```

Весь ввод-вывод пользователя осуществляется в режиме ведущего. Операция ввода-вывода запускается командой

$StartIO(Ch, S, D, n);$

где Ch — номер канала, S — массив исходных блоков (длиной 10 слов каждый), D — массив блоков для получения информации, а n — число блоков, которые должны быть переданы. Если $StartIO$ выдается занятому каналу, то ЦОУ приставляет в состоянии ожидания до тех пор, пока не освободится канал, после чего $StartIO$ воспринимается. (Выдача $StartIO$ занятому ка-

налу обычно нежелательна.) Состояние любого канала может быть определено проверкой регистров состояния канала $CHST$; $CHST[i] = 1$, если i -й канал занят и $CHST[i] = 0$, когда канал i свободен ($i = 1, 2, 3$).

Чтобы переключиться в режим ведомого, выдается команда

$Slave(ptr, c, r, ic);$

$Slave$ устанавливает PTR в ptr , C в c , R в r , IC в ic , а затем включает режим ведомого в начале цикла выполнения эмулятора.

Обычно команды в режиме ведущего выполняются за *нуль* единиц времени. Однако иногда необходимо заставить ЦОУ ожидать некоторый определенный интервал времени, прежде чем продолжить работу дальше. Это происходит неявно, когда команда $StartIO$ выдается занятому каналу. Явное ожидание достигается командой

$Superwait(t);$

которая заставляет ЦОУ простоявать t единиц времени в состоянии ожидания.

(d) Каналы

Когда команда $StartIO$ принимается адресуемым каналом i , регистр $CHST[i]$ устанавливается в 1 (занят), и пересылка ввода-вывода осуществляется полностью параллельно с продолжающейся работой ЦОУ. При завершении ввода-вывода регистр $CHST[i]$ устанавливается в 0, и сигнал прерывания от ввода-вывода сбрасывается.

(e) Таймер

Аппаратура таймера уменьшает содержимое ячейки супервизорной памяти TM на 1 после каждого 10 единиц времени работы ЦОУ. Прерывание от таймера имеет место, когда значение TM достигнет нуля; значение TM продолжает уменьшаться с той же скоростью, так что TM может также иметь отрицательные значения. Значение TM может быть установлено и опрошено в режиме ведущего.

(f) Прерывания

Возможны четыре типа прерываний;

- (1) программные: защита (для таблицы страниц), недействительный код операции,
- (2) супервизорные: GD , PD , H ,
- (3) ввода-вывода: прерывания по завершении операции,
- (4) от таймера: уменьшение до нуля.

События, вызывающие прерывания типа 1 и 2, могут случаться только в режиме ведомого; события типа 3 и 4 могут иметь место и в режимах ведущего и ведомого, а некоторые из таких событий могут происходить одновременно. Причина, вызывающая прерывание, записывается в регистры прерывания независимо от того, запрещены (режим ведущего) или разрешены (режим ведомого) прерывания.

Регистры прерывания принимают при прерывании следующие значения:

- (1) $PI = 1$: защита; $PI = 2$: недействительный код операции;
- (2) $ST = 1 : GD$; $SI = 2 : PD$; $SI = 3 : H$;

(3) $IOI = 1$: канал 1; $IOI = 2$: канал 2; $IOI = 4$: канал 3; если несколько прерываний по завершении ввода-вывода возникают одновременно, то значения суммируются; например, $IOI = 6$ указывает, что возникли прерывания по завершении работы обоих каналов 2 и 3.

- (4) $TI = 1$: таймер.

Следующая программа описывает действия аппаратуры при прерывании в режиме ведомого:

```
comment Сохранить состояние ведомого процесса в ячейках
      супервизорной памяти c, r, ic;
```

```
c := C; r := R; ic := IC;
```

```
comment Переключиться на режим ведущего;
```

```
MODE := 'master';
```

```
comment Определить причину прерывания и передать управ-
      ление;
```

```
if IOI ≠ 0 then go to IOint else
```

```
if PI ≠ 0 then go to PROGint else
```

```
if SI ≠ 0 then go to SUPint else
```

```
go to TIMint;
```

```
comment IOint, PROGint, SUPint, TIMint — это ячейки супер-
      визорной памяти;
```

Следует заметить, что порядок проверки регистров прерываний подразумевает аппаратурную приоритетную схему; это может быть легко изменено программным обеспечением, работающим в режимах ведущего.

A3. Форматы карт задания, программы и данных

Задание пользователя представляется как колода управляющих карт, программных карт и карт данных в следующем порядке:

〈карта JOB〉, 〈программа〉, 〈карта DATA〉, 〈данные〉, 〈карта ENDJOB〉.

1. *〈Карта JOB〉* содержит четыре элемента;

(1) \$AMJ колонки 1—4, это означает мультипрограммное задание (A Multiprogramming JOB),

(2) *〈ид задания〉* колонки 5—8, уникальный 4-символьный идентификатор задания,

(3) *〈оценка времени〉* колонки 9—12, 4-значная максимальная оценка времени,

(4) *〈оценка строк вывода〉* колонки 13—18, 4-значная максимальная оценка объема вводимой информации.

2. Каждая карта колоды *〈программа〉* содержит информацию в колонках 1—40, *i*-я карта содержит начальное состояние ячеек виртуальной памяти пользователя:

$10(i - 1), 10(i - 1) + 1, \dots, 10(i - 1) + 9, i = 1, 2, \dots, n,$

где *n* — число карт в колоде *〈программа〉*. Каждое слово может содержать команду виртуальной машины или четыре байта данных. Число карт *n* в программной колоде определяет размер пространства пользователя, т. е. *n* карт определяют $10 \times n$ слов, $n \leq 10$.

3. *〈Карта DATA〉* имеет формат:

\$DTA колонки 1—4

4. Колода *〈данные〉* содержит информацию в колонках 1—40 и представляет собой данные пользователя, полученные по команде *GD* виртуальной машины.

5. *〈Карта ENDJOB〉* имеет формат:

\$END колонки 1—4

〈ид задания〉 колонки 5—8; содержит ту же информацию, что и *〈карта JOB〉*

〈Карта DATA〉 пропускается, если в задании нет карт *〈данные〉*.

A4. Операционная система

Основной целью МОС является эффективная обработка потока пользовательских заданий, представленных в виде пакета. Эта цель достигается мультипрограммированием системных и пользовательских процессов.

Задание *J* будет проходить последовательно через следующие фазы:

1. Вводной спуллинг. Задание вводится с устройства чтения с перфокарт и пересыпается на барабан.

2. Основная обработка. Программная часть *J* загружается с барабана в память пользователя. Теперь *J* готово к выполнению и превращается в процесс *j*. До тех пор пока *j* не закончится либо нормально, либо в результате ошибки, его состояние будет многократно переключаться между следующими состояниями:

- (а) *готовность* — ожидание назначения ЦОУ;
- (б) *счет* — развитие на ЦОУ;
- (с) *блокировка* — ожидание удовлетворения запроса на ввод-вывод.

Требования ввода-вывода преобразуются средствами МОС в операции ввода-вывода для барабана.

3. Выводной спуллинг. Вывод результатов заданий *J*, включая сведения о расходе ресурсов, системные сообщения и его начальную программу; печатается с барабана.

В общем случае в фазе основной обработки будут одновременно находиться многие задания.

МОС должна быть документирована и запрограммирована как набор интерактивных процессов. В типичном проекте могут быть предусмотрены следующие главные процессы:

<i>Readin_Cards:</i>	Читать карты в супервизорную память.
<i>JOB_to_Drum:</i>	Создать дескриптор задания и переслать задание на барабан.
<i>Loader:</i>	Загрузить задание в пользовательскую память.
<i>Get_Put_Data:</i>	Обрабатывать команды ввода-вывода виртуальной машины.
<i>Lines_from_Drum:</i>	Считать строки, предназначенные для печати с барабана в супервизорную память.
<i>Print_Lines:</i>	Вывести строки на печать.

Операционная система обычно активизируется прерываниями, происходящими при работе в режиме ведомого. Обычно подпрограммы обработки прерываний после того, как они обслужат прерывание, вызывают планировщик процессов (распределитель ЦОУ).

Основная задача МОС — управление аппаратурными и программными ресурсами. В их состав входят память пользователя, память на барабане, канал 3, программные буферы, дескрипторы заданий и ЦОУ.

МОС также отвечает за сбор статистических данных об использовании аппаратуры и о характеристиках задания. Программное обеспечение производит сбор следующих статистических данных.

1. *Использование ресурсов.* Доля общего времени, в течение которого каждый канал занят, доля общего времени, когда занят ЦОУ (в режиме ведомого), среднее значение использования памяти пользователя и среднее значение использования барабана.

2. *Характеристики задания.* Среднее время выполнения (на виртуальной машине), среднее время пребывания в системе, средний объем требуемой памяти пользователя, средняя длина ввода и средняя длина вывода.

Такие статистические данные будут печататься в конце про-гона.

A5. Требования к проекту

Должны быть спроектированы и реализованы три набора программных модулей:

1. Основные имитаторы для аппаратуры, включая систему прерываний, таймер, каналы, считывающие и печатающие уст-ройства, вспомогательную память, память пользователя и стра-ничную систему режима ведомого. (НЛР и супервизорная па-мять предполагаются доступными непосредственно из „системы-хозяина“.)

2. „Микропрограмма“, которая эмулирует виртуальную ма-шину.

3. МОС.

Эти три части должны быть четко и ясно разделены. Не должно возникать трудностей при изменении временных па-раметров и параметров размера аппаратуры, особенно размера памяти на барабане и памяти пользователя, времени ввода-вы-вода, времени выполнения команд и „частоты“ таймера.

Студенты должны работать небольшими бригадами из двух или трех человек, причем каждая бригада выполняет полный про-ект. Через несколько недель после выдачи задания пре-ставляется на рассмотрение полный проект МОС как набор взаимодействующих процессов. Проект включает описание глав-ных процессов в системе и их взаимодействия, методов, кото-рые будут использованы для распределения ресурсов и управ-ления ими, а также идентификацию и содержание основных структур данных.

Для функциональной проверки должен быть подготовлен пакет, состоящий приблизительно из 60 заданий („прогон“).

A6. Некоторые ограничения

МОС и машина отклоняются от реальности из-за упрощения одних свойств реальных систем и пренебрежения другими. Существенными свойствами, которые опущены, являются: более общая виртуальная машина, которая могла бы допустить многошаговые задания и использование языковых трансляторов, система для организации и управления большим разнообразием файлов данных, средство связи с оператором, а также работа ЦОУ в режиме ведущего в течение ненулевого интервала времени. Спецификации проекта могли бы быть расширены в некоторых вышеуказанных направлениях, но дополнительные расходы при этом оказываются неприемлемыми. Вместо этого лучше выполнить подобное проектирование, уделяя внимание другим компонентам и свойствам операционных систем, таким, как файловые системы или разделение памяти.

Список литературы

Тт

?

Александер (Alexander M. T.)

1970 Time sharing supervisor programs. The University of Michigan Computing Center, Ann Arbor, Michigan (May).

Ахо, Деннинг, Ульман (Aho A. V., P. J. Denning, J. D. Ullman)

1971 Principles of optimal page replacement. *J. ACM*, 18, No. 1 (Jan) 80—93.

Баэр, Сейгер (Baer J. L., G. R. Sager)

1972 Measurement and improvement of program behavior under paging systems. In W. Freiberger (ed.), *Statistical Computer Performance Evaluation*, Academic Press, New York, 242—264.

Бетурн, Буленжер, Ферри, Кайзер, Котт, Краковяк, Мосьер (Bétourné C., J. Boulenger, C. Kaiser, J. Kott, S. Krakowiak, J. Mossier)

1969 Process management and resource sharing in the multiaccess system «ESOPE». Proc. ACM Second Symposium on operating system Principles, Princeton University, Princeton, N. J. (Okt.), 67—79.

Биледи (Belady L. A.)

1966 A study of replacement algorithms for a virtual storage computer. *IBM Syst. J.*, 5, No. 2, 78—101.

Боброу, Буркфил, Мерфи, Томлинсон (Bobrow D. G., J. D. Burchfiel, D. L. Murphy, and R. S. Tomlinson)

1972 TENEX, a paged time sharing system for the PDP-10. *Comm. ACM*, 15, No. 3 (March), 135—143.

Боэтнер (Boettner D. W.)

1969 Command (job control) languages for general purpose computing systems. University of Michigan Computing center, Ann Arbor, Michigan (June).

Бринч Хансен (Brinch Hansen P.)

1970 The nucleus of a multiprogramming system. *Comm. ACM*, 13, No. 4 (April), 238—241, 250.

Бринч Хансен (Brinch Hansen P.) (ed.)

1971 RC 4000 Multiprogramming System, 2nd ed. A/S Regnecentralen, Copenhagen, Denmark.

Бэтсон, Джу, Вуд (Batson A., S. Ju, D. C. Wood)

1970 Measurements of segment size. *Comm. ACM*, 13, No. 3 (March) 155—159.

Вареха, Ратледж, Голд (Vareha A. L., R. M. Rutledge, M. M. Gold)

1969 Strategies for structuring two-level memories in a paging environment. Proc. ACM Second Symposium on Operating Systems Principles, Princeton University, Princeton, N. J. (Oct.), 54—59.

Вегнер (Wegner P.)

1968 Programming Languages, Information Structures, and Machine Organization. McGraw-Hill, New York.

Вейдерман (Weiderman N.)

1971 Synchronization and simulation in operating system construction. Ph. D. thesis, Tech. Rep. 71—102, Computer Science, Cornell University, Ithaca, N. Y. (Sept.).

Вирт (Wirth N.)

1966 A note on «Program Structures for Parallel Processing». *Comm. ACM*, 9, No. 5 (May), 320—321.

Вирт (Wirth N.)

1969 On multiprogramming, machine coding and computer organization. *Comm. ACM*, 12, No. 9 (Sept.), 489—498.

Гиер (Gear C. W.)

1969 Computer Organization and Programming. McGraw-Hill Book Company, New York.

Грис (Gries D.)

1971 Compiler Construction for Digital Computers. John Wiley & Sons, New York. (Есть русский перевод: Д. Грис. Конструирование компиляторов для цифровых вычислительных машин. Пер. с англ.—М.: Мир, 1975.)

Грэхем (Graham R. M.)

1968 Protection in an information processing utility. *Comm. ACM*, 11, No. 5 (May), 365—369.

Далей, Денис (Daley R. C., J. D. Dennis)

1968 Virtual memory, processes, and sharing in Multics. *Comm. ACM*, 11, No. 5 (May), 306—312.

Далей, Нейман (Daley R. C., P. G. Neumann)

1965 A general purpose file system for secondary storage. Proc. AFIPS 1965 Spring Joint Comput. Conf., 27, Part I. Spartan Books, New York, 213—230.

Дейкстра (Dijkstra E. W.)

1965a Cooperating sequential processes. Mathematics Dept., Technological University, Eindhoven, The Netherlands.

Дейкстра (Dijkstra E. W.)

1965b Solution of a problem in concurrent programming control. *Comm. ACM*, 8, No. 9 (Sept.), 569.

Дейкстра (Dijkstra E. W.)

1968a The structure of the «THE»-multiprogramming system. *Comm. ACM*, 11, No. 5 (May), 314—346.

Дейкстра (Dijkstra E. W.)

1968b Co-operating sequential processes. In F. Genuys (ed.), Programming Languages, Academic Press, New York, 43—112. (Есть русский перевод: Э. Дейкстра в сб. «Языки программирования» под ред. Ф. Женюи. Пер. с англ.—М.: Мир, 1972.)

Дейкстра (Dijkstra E. W.)

1969 Notes on structured programming. EWD 249, Technological University, Eindhoven, The Netherlands. (Есть русский перевод: Э. Дейкстра. Структурное программирование. Пер. с англ. — М.: Мир, 1975.)

Деннинг (Denning P. J.)

1968 The working set model for program behavior. *Comm. ACM*, 11, No. 5 (May), 323—333.

Деннинг (Denning P. J.)

1970 Virtual memory. *Computing Surveys*, 2, No. 3 (Sept.), 153—189.

Деннинг (Denning P. J.)

1972 A note on paging drum efficiency. *Computing Surveys*, 4, No. 1 (March), 1—3.

Деннис, Ван Хорн (Dennis J. B., E. C. Van Horn)

1966 Programming semantics for multiprogrammed computations. *Comm. ACM*, 9, No. 3 (March), 143—155.

Деннис (Dennis J. B.)

1965 Segmentation and the design of multiprogrammed operating systems. *J. ACM*, 12, No. 4 (Oct.), 589—602.

Зальцер (Salzer J. H.)

1966 Traffic control in a multiplexed computer system. MAC-TR-30 (thesis), MIT, Cambridge, Mass. (July).

Кембридж (Cambridge)

1967 Cambridge Multi-Access System Manual. University Mathematical Laboratory, Cambridge University, Cambridge, England.

Килбури, Эдвардс, Лэнгиган, Самнер (Kilburn T., D. B. G. Edwards, M. J. Langigan, F. H. Sumner)

1962 One-level storage system. *IRE Trans. EC-11*, 2 (April), 223—235.

Комфорт (Comfort W. T.)

1965 A computing system design for user service. Proc. AFIPS 1965 Fall Joint Comput. Conf., 27. Spartan Books, New York; 619—628.

Кнут (Knuth D. E.)

1968 The art of Computer Programming, Vol. I. Addison-Wesley, Reading, Mass. (Есть русский перевод: Д. Кнут. Искусство программирования для ЭВМ, т. 1, Основные алгоритмы. Пер. с англ. — М.: Мир, 1976.)

Конвей (Conway M.)

1963 A multiprocessor system design. Proc. AFIPS 1963 Fall Joint Comput. Conf., 24. Spartan Books, New York, 139—146.

Конти, Гибсон, Питковски (Conti C. J., D. H. Gibson, S. H. Pitkowsky)

1968 Structural aspects of the System/360 model 85, I General organization. *IBM Syst. J.*, 7, No. 1, 2—14.

Корбато, Дагgett, Далей (Corbató F. J., M. M. Dagget, R. C. Daley)

1962 An experimental time-sharing system. Proc. AFIPS 1962 Spring Joint Comput. Conf., 21, 335—344.

- Коффман, Вариан (Coffman E. G., Jr., L. C. Varian)
1968 Further experimental data on behavior of programs in a paging environment. *Comm. ACM*, 11, No. 7 (July), 471—474.
- Коффман, Элфрик, Шошани (Coffman E. G., M. J. Elphick, A. Shoshani)
1971 System deadlocks. *Computing Surveys*, 3, No. 2 (June), 67—78.
- Коффман, Клейнрок (Coffman E. G., L. Kleinrock)
1968 Computer scheduling measures and their countermeasures. Proc. AFIPS
1968 Spring Joint Comput. Conf., 11—21.
- Куртуа, Хейманс, Парнас (Courtois P. J., R. Heymans, D. L. Parnas)
1971 Concurrent control with «Readers» and «Writers». *Comm. ACM*, 14, No. 10
(Oct.), 667—668.
- Липтей (Liptay J. S.)
1968 Structural aspects of the System/360 model 85, II The cashe. *IBM Syst. J.*, 7, No. 1, 15—21.
- Лэмпсон (Lampson B. W.)
1968 A scheduling philosophy for multiprocessing systems. *Comm. ACM*, 11,
No. 5 (May), 347—360.
- Майер, Сиврайт (Meyer R. A., L. H. Seawright)
1970 A virtual machine time-sharing system. *IBM Syst. J.*, 9, No. 3, 199—218.
- Миллер (Miller W. F.)
1968 Lecture notes, C. S. 246, Computer Science Dept., Stanford University,
Stanford, Calif. (unpublished).
- Мински (Minsky M. L.)
1967 Computation: Finite and Infinite Machines. Prentice-Hall Inc., Englewood
Cliffs, N. J.
- Морис (Morris R.)
1968 Scatter storage techniques. *Comm. ACM*, 11, No. 1 (Jan.), 38—44
- Мэдник (Madnick S. E.)
1968 Design strategies for file systems: a model. Scientific Center Report,
2nd Revision, April 1970, IBM Corp., Cambridge scientific Center, Cam-
bridge, Mass.
- Мэдник, Олсон (Madnick S. E., J. W. Alsop)
1969 A modular approach to file system design. Proc. AFIPS 1969 Spring
Joint Comput. Conf., 34, AFIPS Press, Montvale, N. J., 1—13.
- Мэттсон, Джекси, Слутц, Трейдер (Mattson R. L., J. Gecsi, D. R. Slutz,
I. L. Traiger)
1970 Evaluation techniques for storage hierarchies. *IBM Syst. J.*, 9, 2, 78—117.
- Найр (Naur P. (ed.))
1963 Revised report on the algorithmic language ALGOL 60. *Comm. ACM*, 6,
No. 1 (Jan.), 1—17.
- Оппенгеймер, Вайзер (Oppenheimer G., N. Weizer)
1968 Resource management for a medium scale time-sharing system. *Comm.*
ACM, 11, No. 5 (May), 313—322.

Органик (Organick E. I.)

1972 *The Multics System: An Examination of its Structure*. The MIT Press, Cambridge, Mass.

Рассел (Russel R. D.)

1972 A model for deadlock-free resource allocation. SLAC Report No. 148, Stanford Linear Accelerator Center, Stanford, Calif. (June). Ph. D. thesis, Computer Science, Stanford University.

Розен (Rosen S. (ed.))

1967 *Programming Systems and Languages*. McGraw-Hill, New York.

Розен (Rosen S.)

1969 Electronic computers: a historical survey. *Computing Surveys*, 1, No. 1 (March), 7—36.

Розин (Rosin R. F.)

1969 Supervisory and monitor systems. *Computing Surveys*, 1, No. 1 (March), 37—54.

Рэнделл (Randell B.)

1969 A note on Storage fragmentation and program segmentation. *Comm. ACM*, 12, No. 7 (July), 365—369, 372.

Рэнделл, Кюнер (Randell B., C. J. Kuehner)

1968 Dynamic storage allocation systems. *Comm. ACM*, 11, No. 5 (May), 297—306.

Симпсон, Грабтри, Рей, Филипс, Хит (Simpson T. H., R. P. Grabtree, R. O. Ray, G. H. Phillips, R. B. Hitt)

1969 Houston automatic spooling priority system — II (Version 2). IBM Type III Program No. 360D-05.1.014, IBM Corp., Program Information Department, Hawthorne, N. Y.

Уилкс (Wilkes M. V.)

1965 Slave memories and dynamic storage allocation. *IEEE Trans. EC* — 14 (April), 270—271.

Уилкс (Wilkes M. V.)

1968 *Time-sharing Computer Systems*. American Elsevier, New York. (Есть русский перевод: М. Уилкс. Системы с разделением времени. Пер. с англ. — М.: Мир, 1972.)

Уолман (Wolman E.)

1965 A fixed optimum cell-size for records of various lengths. *J. ACM*, 12, No. 1 (Jan.), 53—70.

Уотсон (Watson R. W.)

1970 *Timesharing System Design Concepts*. McGraw-Hill, New York.

Файн, Джексон, Мак-Исаак (Fine E. C., C. W. Jackson, P. V. McIsaac)

1966 Dynamic program behavior under paging. Proc. ACM 21st Nat. Conf., Thompson Book Co., Washington, D. C., 223—228.

Фрейбергс (Freibergs I. F.)

1968 The dynamic behavior of programs. Proc. AFIPS 1968 Fall Joint Comput. Conf., 33, Part 2, 1163—1167.

- Хаберман (Habermann A. N.)
1969 Prevention of system deadlocks. *Comm. ACM*, 12, No. 7 (July), 373—377, 385.
- Хавендер (Havender J. W.)
1968 Avoiding deadlock in multitasking systems. *IBM Syst. J.*, 7, No. 2, 74—84.
- Хебалкар (Hebalkar P. G.)
1970 Deadlock-free sharing of resources in asynchronous systems. MAC-TR-75 (Ph. D. thesis), Massachusetts Institute of Technology, Cambridge, Mass. (Sept.).
- Холт (Holt R. C.)
1971a Comments on prevention of system deadlocks. *Comm. ACM*, 14, No. 1 (Jan.), 36—38.
- Холт (Holt R. C.)
1971b On deadlock in computer systems. Ph. D. thesis, Tech. Rep. 71—91, Computer Science, Cornell University, Ithaca, N. Y. (Jan.).
- Холт (Holt R. C.)
1972 Some deadlock properties of computer systems. *ACM Computing Surveys*, 4, No. 3 (Sept.), 179—196.
- Хорнинг, Рэнделл (Horning J. J., B. Randell)
1973 Process Structuring. *Computing Surveys*, 5, No. 1 (March), 5—30.
- Цуркер, Рэнделл (Zurcher F. W. and B. Randell)
1968 Iterative multi-level modeling — a methodology for computer system design. Proc. IFIP Congress 1968, North — Holland Publishing Co., Amsterdam, The Netherlands, 867—871.
- Шоу, Вейдерман (Shaw A. C., N. Weiderman)
1971 A multiprogramming system for education and research. Proc. IFIP Congress 71, North-Holland Publishing Co., Amsterdam, The Netherlands, 1505—1509.
- Шоу, Вейдерман, Эидрюс, Фелсин, Рейбер, Вонг (Shaw A. C., N. Weiderman, G. Andrews, M. Felcyn, J. Rieber, G. Wong)
1973 A multiprogramming nucleus with dynamic resource facilities. Tech. Report 72-12-04, Computer Science Group, University of Washington, Seattle, Washington (Revised, April).
- Шошани, Коффман (Shoshani A., E. G. Coffman)
1969 Prevention, detection, and recovery from system deadlocks. Computer Science Lab., Tech. Report No. 80, Dept. of Electrical Engineering, Princeton University.
- Эйзенберг, Мак-Гайер (Eisenberg M. A., M. R. McGuire)
1972 Further comments on Dijkstra's concurrent programming control problem. *Comm. ACM*, 15, No. 11 (Nov.), 999.
- Элспас, Левит, Уолдингер, Ваксман (Elspas B., K. N. Levitt, R. J. Waldinger, A. Waksman)
1972 An assessment of techniques for proving program correctness. *Computing Surveys* 4, No. 2 (June), 97—147.

Burroughgs

1964 B5500 Information Processing Systems, Reference Manual. Burroughs Corporation, Detroit, Michigan.

Burroughs

1967 B6500 Information Processing Systems, Characteristics Manual. Burroughs Corporation, Detroit, Michigan.

CDC

1969 Control Data 6400/6500/6600 Computer Systems Reference Manual. Control Data Corporation, St. Paul, Minnesota.

CDC

1971 Control Data 6000 Computer Systems, SCOPE Reference Manual. Pub. No. 60189400, Control Data Corporation, Sunnyvale, Calif.

CLASP

1971 Users Manual. Office of Computer Services, Cornell University, Ithaca, N. Y.

CP-67/CMS

1969 Program 360D-05.2.005, IBM Corp., Program Inf. Dept., Hawthorne, N. Y. (June).

IBM

1963 IBM 7090 data processing system multiprogramming package. IBM Special Systems Feature Bulletin L22-6641-3, IBM Corp., White Plains, N. Y.

IBM

1965 IBM System/360 Operating System. Concepts and Facilities. Form C28-6535, IBM Corp., Poughkeepsie, N. Y.

IBM

1967 IBM System/360 Operating System: Supervisor and Data Management Services. Form C28-6646, IBM Corp., Programming Systems Publications, Poughkeepsie, N. Y.

MTS

1967 The MTS Manual, Vols. I and II, University of Michigan Computing Center Publications, Ann Arbor, Michigan (Dec.)

Multics

1967 J. L. Bash, E. G. Benjafield, and M. L. Gandy, The Multics operating system—an overview of Multics as it being developed, Project MAC, MIT, Cambridge, Mass. (May). См. также Органик (1972).

Sigplan

1972 SIGPLAN Notices 7, 11 (Nov.) Special issue on control structures in programming languages. B. M. Leavenworth, (ed.).

TSS

1967 IBM System/360 time sharing system, resident supervisor program logic manual. Form A27-2719-0, IBM Corp., Poughkeepsie, N. Y.

XDS

1969 XDS 940 Computer Reference Manual. Xerox Data Systems, El Segundo, Calif.

Словарь терминов

<i>accounting</i> — учетная информация	<i>currenttime = current time</i> — текущее время
<i>Activate</i> — активизировать	<i>Data</i> — данные
<i>AddBuf</i> = <i>Add Buffer</i> добавить буфер	<i>deadlocked</i> — находящийся в тупике, заведенный в тупик
<i>Addtreetree</i> = <i>Add to tree</i> — добавить к дереву	<i>delay</i> — задерживать, задержка
<i>Adr</i> = <i>Address</i> — адрес	<i>Dequeue</i> — вывести из очереди
<i>allocate</i> — распределить	<i>Destroy</i> — разрушить, уничтожить
<i>Allocator</i> — распределитель	<i>device flag</i> — флаг устройства
<i>and</i> — и (союз)	<i>Dname</i> = <i>device name</i> — имя устройства
<i>ans</i> = <i>answer</i> — отвечать, ответ	<i>edge</i> — ребро, дуга
<i>arg</i> = <i>argument</i> — аргумент	<i>ENTRYPPOINT</i> = <i>entry point</i> — точка входа
<i>ask</i> — спрашивать	<i>Execute</i> — выполнить
<i>BASE ADDR</i> = <i>Base address</i> — базовый адрес	<i>EVENT</i> — событие
<i>blocked</i> = <i>blocked active</i> — блокированный активный	<i>firststatein</i> = <i>first state input</i> — начальное состояние ввода
<i>blockeds</i> = <i>blocked suspended</i> — блокированный приостановленный	<i>Flag</i> — флаг
<i>Blockstart</i> = <i>Block start</i> — начальное блокированиe	<i>FORTRANRUNTIME</i> = <i>FORTRAN run time</i> — время работы транслятора с языка ФОРТРАН
<i>Buf</i> = <i>Buffer</i> — буфер	<i>free</i> — свободный
<i>Bufin</i> = <i>input buffer</i> — буфер ввода	<i>freeprocessors</i> = <i>free processors</i> — свободные процессоры
<i>busy</i> — занятый	<i>Gbuf</i> = <i>get buffer</i> — получить буфер
<i>CARDRDR</i> = <i>Card reader</i> — устройство считывания с перфокарт	<i>Get</i> — получить
<i>ChangePriority</i> = <i>change priority</i> — изменить приоритет	<i>GetBuf</i> = <i>get buffer</i> — получить буфер
<i>cause</i> — вызывать, причина	<i>GetDescriptor</i> = <i>get descriptor</i> — получить дескриптор
<i>channelfree</i> = <i>free channel</i> — свободный канал	<i>GetEmptyBuffer</i> = <i>get empty buffer</i> — получить пустой буфер
<i>channelnumber</i> = <i>channel number</i> — номер канала	<i>GetInternalName</i> = <i>get internal name</i> — получить внутреннее имя
<i>channelprogram</i> = <i>channel program</i> — программа канала	<i>GetNextInputFullBuffer</i> = <i>get next input-full buffer</i> — получить очередной заполненный при вводе буфер
<i>channelprogramarea</i> = <i>channel program area</i> — область (памяти) для программы канала	<i>GetNewInternalProcessName</i> = <i>get new internal process name</i> — получить новое внутреннее имя процессы
<i>column</i> — колонка, столбец	<i>identificationdetails</i> = <i>identification details</i> — детали идентификации
<i>completionmessage</i> = <i>completion message</i> — сообщение о завершении	<i>inarea</i> = <i>input area</i> — область ввода
<i>Compute</i> — вычислить	<i>Insertinrecoverycost</i> = <i>insert in reco-</i>
<i>Computeonly</i> = <i>compute only</i> — только вычислить	
<i>consumer</i> — потребитель	
<i>CopyDescriptor</i> = <i>copy descriptor</i> — копировать дескриптор	
<i>create</i> — создать	

<i>very cost</i> — включить в стоимость восстановления	<i>progeny</i> — потомок
<i>Interruptclass</i> = <i>interrupt class</i> — класс прерываний	<i>Queue</i> — очередь
<i>interruptdetails</i> = <i>interrupt details</i> — информация о прерывании	<i>Rbufin</i> = <i>request input buffer</i> — запросить буфер ввода
<i>Iocompletion</i> = <i>input-output completion</i> — завершение операций ввода — вывода	<i>Rbufout</i> = <i>request output buffer</i> — запросить буфер вывода
<i>Ioinstruction</i> = <i>input-output instruction</i> — команда ввода-вывода	<i>ready</i> — готовый
<i>IOP</i> = <i>input-output program</i> — программа ввода-вывода	<i>readya</i> = <i>ready active</i> — готовый активный
<i>Iorequest</i> = <i>input-output request</i> — запрос ввода-вывода	<i>readys</i> = <i>ready suspended</i> — готовый приостановленный
<i>Join</i> — соединить	<i>Receive</i> — получить
<i>Kill</i> — убить	<i>ReleaseEmptyBuffer</i> = <i>release empty buffer</i> — освободить пустой буфер
<i>Left</i> — левый	<i>ReleaseOutputFullBuffer</i> = <i>release output-full buffer</i> — освободить заполненный для вывода буфер
<i>Loc</i> = <i>location</i> — положение, ячейка	<i>Removefrom</i> = <i>remove from</i> — удалить из
<i>Load</i> — загрузить	<i>Removefromr-queue</i> = <i>remove from r-queue</i> — удалить из очереди <i>r</i>
<i>LookInAFD</i> = <i>look in AFD</i> — просмотр в АФД	<i>RemoveProcessDescriptor</i> = <i>remove process descriptor</i> — удалить дескриптор процесса
<i>loop</i> — цикл	<i>RemoveResourceDescriptor</i> = <i>remove resource descriptor</i> — удалить дескриптор ресурса
<i>MAINSTORE</i> = <i>main store</i> — основная память	<i>requestdetails</i> = <i>request details</i> — детали запроса
<i>Mask</i> — маска, маскировать	<i>resourceclass</i> = <i>resource class</i> — класс ресурса
<i>master</i> — хозяин, ведущий	<i>RealRead</i> = <i>real read</i> — действительное чтение
<i>MODE</i> — способ	<i>RealWrite</i> = <i>real write</i> — действительная запись
<i>mutex</i> = <i>mutual excluding</i> — взаимное исключение	<i>ReleaseBuf</i> = <i>release buffer</i> — освободить буфер
<i>MYPROG</i> = <i>my program</i> — моя программа	<i>Restore state</i> — восстановить состояние
<i>NAME</i> — имя	<i>resume</i> — возобновить
<i>Next</i> — следующий, очередной	<i>Right</i> — правый
<i>nextget</i> = <i>get next</i> — получить очередной	<i>Rightlogicalshift</i> = <i>right logical shift</i> — логический сдвиг вправо
<i>nextio</i> = <i>input-output next</i> — очередной ввод-вывод.	<i>root</i> — корень
<i>Nextreadya</i> = <i>next ready active</i> — очередной готовый активный	<i>row</i> — ряд, строка
<i>No.ofElements</i> = <i>there are not elements</i> — нет элементов	<i>Run</i> — продолжаться, развиваться
<i>oldlast</i> = <i>old last</i> — старый последний	<i>running</i> — развивающийся
<i>open</i> — открыть	<i>Scheduler</i> — планировщик
<i>OPTIMIZED</i> — оптимизированный	<i>search</i> — искать
<i>order</i> — порядок	<i>SearchDirectory</i> = <i>search in directory</i> — искать по справочнику
<i>outarea</i> = <i>output area</i> — область вывода	<i>SearchSystemsDirectory</i> = <i>search in systems directory</i> — искать по системному справочнику
<i>Parent</i> — отец	<i>Send</i> — посылать,
<i>Path</i> — путь	<i>Sendanswer</i> = <i>send answer</i> — послать ответ
<i>plus</i> — плюс	
<i>posted</i> — отмеченный	
<i>PRINTER</i> — печатающее устройство	
<i>product</i> — продукт	
<i>producer</i> — производитель	

Sendmessage = *send message* — послать сообщение
Slave — подчиненный, ведомый
Startio = *start input-output* — начать ввод-вывод
StoreState = *store state* — запомнить состояние
Suspend — приостанавливать
Store — запоминать
Switch — переключать
TakeBuf = *take buffer* — взять буфер
Terminate — завершить

Text — текст
Transfer to = *transfer to* — перейти к
Treesearch = *tree search* — поиск по дереву
Uname = *unit name* — имя устройства
VERSION — версия
Waitanswer = *wait answer* — ждать ответа
Waitmessage = *wait message* — ждать сообщения

Оглавление

<i>Предисловие редактора перевода</i>	5
<i>Предисловие</i>	7
1. Организация вычислительных систем	11
<i>1.1. Некоторые определения</i>	11
<i>1.2. Нотация для алгоритмов</i>	14
<i>Упражнение</i>	16
<i>1.3. Исторический аспект</i>	16
<i>1.3.1. Ранние системы</i>	16
<i>1.3.2. Второе поколение аппаратуры и программного обеспечения</i>	20
<i>1.3.3. Системы третьего и последующих поколений</i>	21
<i>1.4. Некоторые аспекты операционных систем</i>	23
<i>1.4.1. Виртуальные машины, трансляция и распределение ресурсов</i>	24
<i>1.4.2. Четыре ключевые проблемы</i>	27
<i>1.5. Организация систем</i>	28
2. Системы пакетной обработки	31
<i>2.1. Введение</i>	31
<i>2.2. Связывание и загрузка</i>	32
<i>2.2.1. Статическое перемещение</i>	34
<i>2.2.2. Процесс связывания</i>	35
<i>Упражнение</i>	40
<i>2.3. Методы ввода-вывода</i>	40
<i>2.3.1. Прямой ввод-вывод</i>	42
<i>2.3.2. Косвенный ввод-вывод</i>	43
<i>2.4. Программное обеспечение буферизации ввода-вывода</i>	45
<i>2.4.1. ЦОУ опрашивает канал</i>	46
<i>2.4.2. Составные буфера и сопрограммная структура программ</i>	49
<i>Упражнения</i>	55
<i>2.4.3. Канал прерывает ЦОУ</i>	55
<i>Упражнение</i>	58
<i>2.4.4. Объединение буферов в пул для ввода и вывода</i>	58
<i>Упражнения</i>	65
<i>2.5. Супервизор ввода-вывода</i>	67
3. Взаимодействующие процессы	69
<i>3.1. Параллельное программирование</i>	69
<i>3.1.1. Применения</i>	69
<i>Упражнение</i>	72
<i>3.1.2. Некоторые программные конструкции для параллелизма</i>	72
<i>Упражнения</i>	77
<i>3.2. Концепция процесса</i>	77
<i>3.3. Проблема критической секции</i>	79
<i>3.3.1. Проблема</i>	79
<i>3.3.2. Программное решение (Дейкстра, 1965а, 1968б)</i>	81
<i>Упражнения</i>	85

3.4. Семафорные примитивы	86
3.4.1. <i>P</i> - и <i>V</i> -операции	86
3.4.2. Взаимное исключение с помощью семафорных операций	87
3.4.3. Семафоры как счетчики ресурсов и синхронизаторы в проблемах производителя и потребителя	88
Упражнения	101
3.5. Реализация семафорных операций	103
3.5.1. Реализация с «заятым» ожиданием	104
Упражнения	105
3.5.2. Устранение занятого ожидания	105
Упражнение	107
3.6. Другие синхронизирующие примитивы	107
Упражнения	111
4. Введение в системы мультипрограммирования	112
4.1. Доводы в пользу мультипрограммирования	112
4.2. Компоненты систем	114
4.2.1. Характеристики аппаратуры	114
4.2.2. Базовое программное обеспечение	117
4.3. Ядро операционной системы	122
4.4. Пользовательский интерфейс	129
4.4.1. Командный и управляющий языки	129
4.4.2. Управление заданием	131
4.5. Элементы методологии проектирования	133
5. Управление основной памятью	137
5.1. Статическая и динамическая настройка адресов	137
5.1.1. Аппаратурная настройка адреса	137
5.1.2. Аргументы в пользу статического и динамического перемещений	139
Упражнение	143
5.1.3. Типы виртуальной памяти	143
5.2. Принципы сегментации и страничной организации	145
5.2.1. Односегментное пространство имен	145
Упражнение	147
5.2.2. Многосегментное пространство имен	152
5.3. Защита реальной и виртуальной памяти	155
Упражнения	163
5.4. Стратегии распределения памяти	163
5.4.1. Распределение памяти в иерархиях системах	163
Упражнение	171
5.4.2. Распределение в страничных системах	172
Упражнения	179
5.5. Оценка страничной организации	180
5.6. Иерархии памяти	184
6. Разделение процедур и данных в основной памяти	189
6.1. Необходимость разделения ресурсов	189
6.2. Условия разделения программ	191
6.3. Разделение в системах со статическим распределением	194
Упражнение	196
6.4. Динамическое разделение	196
6.4.1. Форма процедурного сегмента	198
6.4.2. Связывание данных	201
6.4.3. Обращения к процедурам	202

7. Управление процессами и ресурсами	205
7.1. Структуры данных для процессов и ресурсов	206
7.1.1. Дескрипторы процесса	207
7.1.2. Дескрипторы ресурсов	211
7.2. Основные операции над процессами и ресурсами	218
7.2.1. Управление процессом	218
Упражнения	223
7.2.2. Примитивы ресурсов	223
Упражнение	229
7.2.3. Полномочия процесса	229
7.3. Прерывания и процессы ввода-вывода	230
7.4. Организация планировщиков процессов	234
7.4.1. Ведущие и разделяемые планировщики	237
7.4.2. Приоритетное планирование	239
Упражнения	243
7.5. Методы планирования	243
Упражнение	248
8. Проблема тупиков	249
8.1. Примеры тупиков в вычислительных системах	250
Упражнение	256
8.2. Модель системы	256
Упражнения	259
8.3. Тупик в случае повторного использования ресурсов	260
8.3.1. Графы повторного использования ресурсов	260
8.3.2. Распознавание тупика	263
Упражнения	273
8.3.3. Выход из тупика	273
Упражнения	276
8.3.4. Методы предотвращения тупиков	277
Упражнения	282
8.4. Системы с потребляемыми ресурсами	282
Упражнения	290
8.5. Графы обобщенных ресурсов	290
Упражнения	292
8.6. Динамическое добавление и удаление процессов и ресурсов	292
9. Файловые системы	296
9.1. Виртуальная и реальная файловая память	297
9.2. Компоненты файловой системы	300
9.3. Логическая и физическая организации	305
9.4. Процедуры доступа	310
9.4.1. Файловые справочники	310
Упражнение	314
9.4.2. Дескрипторы файлов	314
9.4.3. Управление доступом	316
Упражнение	319
9.4.4. Стандартные программы открытия и закрытия	319
9.5. Управление пространством вспомогательной памяти	321
9.6. Иерархическая модель для файловых систем	325
Упражнение	329
9.7. Восстановление при отказах системы	329
Упражнение	335

Приложение. Проектирование мультипрограммной системы	336
A1. Введение	336
A2. Машины с спецификации	336
A3. Форматы карт задания, программы и данных	342
A4. Операционная система	343
A5. Требования к проекту	345
A6. Некоторые ограничения	346
Список литературы	347
Словарь терминов	354

Алан Шоу

**ЛОГИЧЕСКОЕ ПРОЕКТИРОВАНИЕ
ОПЕРАЦИОННЫХ СИСТЕМ**

Научн. ред. Л. Н. Бабынина

Младш. научн. ред. Э. Г. Иванова

Художник Д. А. Аникеев

Художественный редактор В. И. Шаповалов

Технический редактор Е. В. Ящук

Корректор Н. В. Андреева

ИБ № 2509

Сдано в набор 11.12.80. Подписано к печати 27.05.81.
Формат 60×90 $\frac{1}{4}$. Бумага типографская № 2. Гарнитура
литературная. Печать высокая. Объем 11,25 бум. л. Усл. пе.
л. 22,50. Усл. кр.-отт. 22,50. Уч.-изд. л. 20,88. Изд. № 1/0957.
Тираж 49 000 экз. Зак. 941. Цена 1 р. 70 к.

**ИЗДАТЕЛЬСТВО «МИР»
129820, Москва, И-110, ГСП. 1-й Рижский пер., 2**

Ленинградская типография № 2 головное предприятие ордена
Трудового Красного Знамени Ленинградского объединения
«Техническая книга» им. Евгении Соколовой Союзполиграф-
прома при Государственном комитете СССР по делам изда-
тельств, полиграфии и книжной торговли. 198052, г. Ленин-
град, Л-52, Измайловский проспект, 29