АКАДЕМИЯ НАУК СССР
СИБИРСКОЕ ОТДЕЛЕНИЕ

Серия «Наука и технический прогресс»

В. А. Вальковский
В. Э. Малышкин

Элементы современного программирования и суперЭВМ

Ответственный редактор
доктор физико-математических наук
В. Е. КОТОВ

НОВОСИБИРСК
«НАУКА»
СИБИРСКОЕ ОТДЕЛЕНИЕ
1990

ББК 22.18 + 32.973—01
В16
УДК 519.692

Рецензенты

доктор физико-математических наук *Н. Н. Миренков*
кандидат физико-математических наук *Б. Г. Чеблаков*

Утверждено к печати
Вычислительным центром СО АН СССР

Вальковский В. А., Малышкин В. Э.

В16 Элементы современного программирования и суперЭВМ.— Новосибирск: Наука. Сиб. отд-ние, 1990.— 143 с.— (Серия «Наука и технический прогресс»).

ISBN 5—02—029652—Х.

В книге рассказывается об устройстве параллельных ЭВМ, языках параллельного программирования, способах и алгоритмах организации вычислений, подходах к решению задач на ЭВМ. Предполагается, что читатели хоть немного знакомы с каким-либо языком программирования, например Бейсиком, сами пробовали решать задачи и теперь хотели бы расширить свой кругозор, получить пищу для новых размышлений. В книгу вошли редко популяризируемые материалы: конструкции параллельных языков, алгоритмы поиска данных, асинхронное управление, сложные структуры данных.

Издание адресовано широкому кругу читателей: школьникам старших классов, студентам, инженерам, всем тем, кто интересуется нюансами программирования для ЭВМ.

В $\frac{1402000000-194}{054(02)-90}$ 46—90 НП ББК 22.18 + 32.973—01

ISBN 5—02—029652—Х

К читателю

Электронные вычислительные машины (ЭВМ) применяются буквально во всех сферах нашей жизни: науке, производстве, культуре, медицине, общественной жизни и т. д. Невозможно назвать такой род деятельности человека, где нельзя было бы использовать ЭВМ. Компьютеры произвели переворот в нашем восприятии мира: безлюдные заводы-автоматы уже никого не удивляют, не будоражат фантазию, это нормально. Недавно появившиеся персональные компьютеры становятся такими же постоянными спутниками человека, как наручные часы.

Обычно люди имеют дело с компьютерами как пользователи, с их помощью они покупают авиабилеты, производят расчеты, получают справки, играют с ними в шахматы и т. д. В этом случае нет никакой необходимости знать о том, как работает ЭВМ, как она устроена, что такое программа, как ее создать и т. д. Но часто, особенно для решения новых задач, возникает необходимость разрабатывать новые программы — программировать решение задачи. Программирование оказалось крайне увлекательным занятием. Может показаться странным, но сегодня многие люди с наслаждением и упоением отдаются программированию, ничуть не меньшими, чем, скажем, дворовые мальчишки-футболисты — игре. Программирование — не просто и не только наука, но и искусство, и спорт, и творчество, и забава — это как в жизни, и каждая программа есть реальный мир, только существует он в ЭВМ, программа — венец в решении задачи.

Задача решается в несколько этапов: 1) вначале в реальном мире возникает проблема (надо, например, уметь хорошо управлять экономикой); 2) затем она формулируется в виде задачи (и уже видно, где и какие математические методы могут быть применены); 3) задача формализуется, строится математическая модель реального мира (экономическая система в пер-

вом приближении описывается системами линейных алгебраических уравнений); 4) взаимодействия в модели описываются алгоритмами (способы решений систем линейных уравнений); 5) как реализация алгоритмов появляются программы; 6) и, наконец, если в программы ввести реальные данные (заводы, цены, материалы, килограммы, поезда и т. д.) и начать программы исполнять, в ЭВМ возникает блистательный математический мир, в чем-то похожий на настоящий (заводы начинают покупать материалы, производить продукцию, продавать товар, будут формироваться цены, выплачиваться премии и т. д. и т. п.).

Этот мир может «жить» многократно, возрождаясь с началом работы программы и погибая по ее завершению, меняться в зависимости от введенных данных; с разными данными мы фактически получаем и разные миры, объединенные общими правилами «игры» — общими алгоритмами взаимодействия. Так что каждый программист — это Творец, бог созданных им миров. И каждый программист, видя хорошо работающую программу, вправе заявить: «Это хорошо». В пылу творения не следует, однако, забывать о том, что математический мир настолько близок к реальному, насколько близки к нему математическая модель и ее реализация, и нередко они мало похожи.

Создание своего мира, отражающего реалии, оказывается не простым занятием, проблема эта комплексная. Даже Господь, говорят, творил свой мир в течение шести дней, каждый день занимаясь разными делами, на седьмой отдыхал. И программист в шесть этапов решает задачу, на седьмом восхищается результатами трудов своих. Вскоре замечает несовершенства созданного мира, но в отличие от мифического Господа, настоящий программист рук не опускает, а принимается за модернизацию, даже если его об этом никто не просит. Конец модернизации нередко приходит лишь с «концом света» (увлекшегося программиста отлучают от ЭВМ, дают другую задачу и т. д.).

Специфика программирования такова, что человек в течение всей жизни в этой профессии вынужден учиться. Создание своего пусть маленького, но работающего мира делает из человека Творца, дает ощущение всесилия, безграничности своих возможностей,

которое у наиболее квалифицированных программистов (суперпрограммистов) ничем не отличается от нахальства. Жизнь, конечно же, никогда не упускает возможности поправить, воспитать в нужном направлении человека, мир вычислений столь сложен и так быстро изменяется, новые проблемы требуют решения новых задач, и программист после краткого пребывания на Олимпе вдруг обнаруживает зияющие пробелы в своей профессиональной подготовке. И все начинается сначала.

В книге рассматриваются лишь немногие, важные с точки зрения авторов, вопросы организации вычислений и разработки программ. Конечно, здесь не найти готовых рецептов, но в ней обсуждаются и иллюстрируются различные сложные проблемы, понятия, задачи и алгоритмы, знание которых совершенно необходимо не только для создания хороших программ, но и вообще для формирования алгоритмического стиля мышления.

Эта книга не совсем популярна в обычном смысле слова. Это, скорее, «книга для чтения» тем, кто уже сделал первые шаги в программировании, успел полюбить эту нетривиальную научную дисциплину и хотел бы получить пищу для дальнейших разышлений.

Записи иллюстративных программ даны на языках Фортран и Паскаль, которые легко поймут и те, кто знаком лишь с Бейсиком.

В книге немало трудного для первого чтения материала, «по диагонали» ее не прочитать. Главы довольно независимы между собой, поэтому чтение можно начинать с любой приглянувшейся без особого ущерба для понимания. Дочитавшись до состояния полного непонимания, надо вернуться назад (сделать backtracking) и повторно начать знакомиться с материалом. Надо постараться, чтобы эти витки развернулись в спираль растущего понимания предмета.

Основное внимание в книге уделено параллельным ЭВМ, проблемам параллельной обработки данных. Иллюстрируя преимущества и недостатки параллелизма, авторы кардинальным образом «распараллелили» работу: разделы «Зачем нужны параллельные ЭВМ», «Как устроены современные ЭВМ», «Как задать параллельные вычисления» написал В. А. Вальковский, а все остальное — В. Э. Малышкин.

● Зачем нужны параллельные ЭВМ

Эволюция человечества включает отдельные периоды, когда та или иная область производственной человеческой деятельности претерпевает необычайно быстрый подъем и становится определяющей в обществе. Так было в свое время с выплавкой металлов — объем выплавки стали и сейчас остается одним из наиболее важных показателей экономической развитости государства. Так произошло и с добычей нефти. Сейчас происходит формирование еще одного стратегического показателя такого рода, но, в отличие от предыдущих, он имеет менее материальный характер.

Этот показатель — вычислительная мощность государства, т. е. число операций, которое может выполнить вся вычислительная техника за единицу времени, если речь идет о совокупной мощности, или экземпляр вычислительного устройства, если речь идет об относительной мощности. Ясно, что прогресс в любой без исключения другой области человеческой деятельности так или иначе связан с ЭВМ, с возможностью быстрой переработки информации, хранения больших объемов информации, анализа и управления на ее основе. Сложность расчетов, связанных с эффективным планированием и использованием ресурсов, быстро возрастающие объемы данных, подлежащих обработке,— все это предъявляет повышенные требования к *производительности* ЭВМ, под которой понимается число операций, выполняемых ЭВМ за секунду времени. Производительность должна также возрастать, и возрастать не медленнее, чем объемы потенциальных вычислений. И дело здесь не только в том, что без такого роста программистам нужно будет дольше ждать решения своих задач. Есть большой круг задач (так называемые *задачи реального времени*), решение которых бессмысленно, если оно будет получено позднее, чем через определенный промежуток времени после ввода начальных данных. Этот промежуток называется *так-*

том реального времени. Например, если машина вычисляет прогноз погоды на завтра, основываясь на погодных данных сегодняшнего дня, то выдача результатов более чем через сутки будет бессмысленна, так как «завтра» уже наступит, и мы можем непосредственно увидеть и измерить этот прогноз. По подсчетам экономистов увеличение точности прогноза всего на 1 % в масштабах страны дает экономический эффект, исчисляемый многими миллионами рублей в год.

Общая схема задачи реального времени включает некоторый объект, функционирующий в некоторой среде. Для поддержания функционирования или осуществления более оптимального функционирования он нуждается в «помощи» ЭВМ. Для этого он соединяется каналом с ЭВМ и регулярно через определенный промежуток времени — *такт реального времени* — или нерегулярно — время от времени — посылает порцию информации, которую машина должна обработать и вернуть через промежуток времени, не превышающий некоторую другую величину — «*период реакции*». В тактированных системах период реакции, как правило, укладывается в длительность такта реального времени.

В качестве примеров систем реального времени рассмотрим системы следующих четырех типов:

1) плановые расчеты предприятия (такт 1 год);
2) прогнозирование погоды (такт 1 сутки);
3) резервирование авиабилетов (такт 1 с);
4) радиолокационное слежение (0.001 с).

Первая, вторая и третья системы имеют фиксированный такт, но все увеличивающийся со временем объем вычислений: нужды производства требуют разработки все более точных планов, возрастает число авиарейсов. В системах четвертого типа такт постоянно уменьшается даже при вычислении того же объема. Так или иначе возникает необходимость производить все большее число операций за единицу времени, т. е. увеличивать вычислительную мощность ЭВМ.

Рассмотрим, за счет чего может возрасти вычислительная мощность? Принципиально каждая ЭВМ состоит из элементов, из которых наиболее типичные — *канал*, *переключатель* и *запоминающий элемент*. На этом уровне рассмотрения процесс вычисления — это процесс «перебегания» каких-то сигналов от одной запоминающей инстанции к другой и переключение ка-

ких-то контактов. Таким образом, если сигналы будут «бегать», а контакты менять состояние быстрее, то и быстродействие будет выше. Быстродействие окажется также выше, если расстояния между элементами будут меньше. Таким образом, быстродействие пропорционально $\frac{S}{r} + V$, где S — скорость распространения сигналов; V — скорость переключения; r — среднее расстояние. Отсюда универсальная рекомендация для увеличения быстродействия — делать элементы ЭВМ (каналы, переключатели, память), обладающие более высокой скоростью и меньшими размерами.

Это определяет один из путей достижения высокой производительности ЭВМ — *микроминиатюризация* устройств. Эволюция вычислительной техники на этом пути особенно заметна. Кажется, совсем недавно самыми современными машинами считались машины фирмы ИБМ, отечественными аналогами которых являются ЭВМ серии ЕС, каждая из них, даже «младшая» в серии, требовала для установки большую специально оборудованную комнату. Но вот появились так называемые персональные ЭВМ, имеющие не меньшую производительность, но не требующие специального электроснабжения и вентиляции, и помещающиеся на одном столе. Впервые размер собственно ЭВМ стал меньше, чем размер дисплея, за которым с ней работают.

По линии ЭВМ сверхвысокой производительности была создана машина Gray-1, имеющая быстродействие на два порядка больше, чем самые большие из серии ЕС, но занимающая объем не более одного кубического метра. Многие современные суперЭВМ располагаются на одной или двух платах размером 20×20 см. Дальнейшее развитие ведет к тому, что вся ЭВМ, кроме, конечно, дисплея и ряда других обслуживающих устройств, будет выполняться как одна интегральная схема, которая будет на выходе непрерывной полностью автоматизированной технологической цепочкой. Такие схемы назвали сверхбольшими интегральными схемами, но не из-за их большого размера, а потому что они содержат очень большое количество тех самых запоминающих и переключательных устройств, о которых шла речь выше.

Можно было бы беспредельно повышать быстродействие вычислительных устройств, все более уменьшая и уменьшая их размеры, увеличивая скорость прохождения сигналов и ускоряя работу переключателей. Однако скорость распространения сигнала ограничена скоростью света, а в современных ЭВМ уже приближается к ней, и резервов в числителе формулы $\frac{S}{r} + V$ почти не осталось. Примерно такие же проблемы возникают и в знаменателе первого слагаемого. На практике они выражаются, например, в том, что если «укладывать» элементы очень близко друг к другу, то весь комплекс будет чрезмерно нагреваться и потребуется система теплоотвода, «пронизывающая» его и мешающая дальнейшему «сближению» элементов. Большие надежды возлагаются на использование эффекта сверхпроводимости, но до практического осуществления этих идей еще далеко. Как было показано во многих работах, скорость переключательных элементов также ограничена величиной, обусловленной некоторыми физическими пределами.

Таким образом, решение задачи повышения производительности ЭВМ путем микроминиатюризации наталкивается на принципиальные трудности. Минимально возможный размер элементов ЭВМ вряд ли может быть сделан меньше элементов человеческого мозга — нейронов. Природа обычно выбирает наилучшее решение: в данном случае и быстродействие значительное, и расстояние невелико. Однако изучение человеческого мозга с точки зрения информатики находится, к сожалению, в зачаточном состоянии и непосредственных выходов в практику пока нет.

Таким образом, увеличение быстродействия ЭВМ за счет качества, т. е. скорости элементов ЭВМ, на текущем этапе в значительной мере исчерпало себя. Какой же выход? Остается обратить внимание на количество элементов. Если взять не один, а несколько вычислительных комплексов и попытаться так поставить задачу, чтобы одновременно решались различные ее части, то, видимо, совокупное время решения задачи при этом уменьшится. Это второй путь повышения производительности ЭВМ. Параллельное программирование идет именно таким путем. Оно видоизменяет информационно-логическую структуру задачи, приспо-

сабливает ее к более быстрой обработке за счет увеличения количества вычислителей. Говорят, что задача распараллеливается на несколько процессоров.

Возникает другой вопрос, можно ли для типичных задач организовать их прохождение таким образом, чтобы одновременно обрабатывались различные компоненты. Оказывается, и мы неоднократно убедимся в том, что для большинства задач это возможно. Некоторые задачи уже по своей природе являются параллельными, а последовательными их сделало традиционное последовательное программирование, ориентированное на однопроцессорные ЭВМ. Чтобы обработать такие задачи параллельно, их нужно перепрограммировать. Другие задачи кажутся с виду сугубо последовательными в том смысле, что известные методы их решения не раскладываются на несколько процессоров. Тогда для них приходится искать более тонкие методы, основанные на малоизвестных закономерностях. Иногда они оказываются весьма неожиданными.

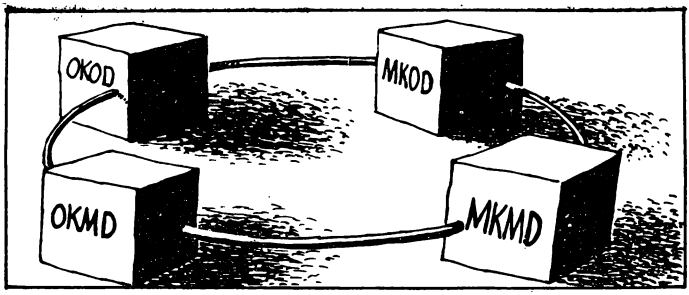
Итак, предположим, что мы, используя как качественные, так и количественные возможности усовершенствования ЭВМ в целях повышения ее производительности, создали сверхсложную и небольшую по размерам многопроцессорную ЭВМ. Возникает законный вопрос, как реально использовать эту производительность. Нам не важна скорость выполнения операций «на холостом ходу», нас интересует скорость решения конкретной задачи, особенно если это задача реального времени. Чтобы решить задачу на такой суперЭВМ, ее нужно запрограммировать. И не как-нибудь, а эффективно. Здесь-то и возникает еще одна проблема эффективного использования вычислительной техники, решение которой открывает третий способ повышения реальной производительности ЭВМ.

Дело в том, что по мере развития и использования вычислительной техники возникли две противоречивые тенденции. С одной стороны, ЭВМ становятся все сложнее, в частности приобретают не одно, а несколько исполнительных устройств. Таким образом, процесс программирования для них становится все более трудоемким и требующим глубоких знаний архитектуры ЭВМ. С другой стороны, все большее число пользователей вовлекается в общение с ЭВМ, используя их не только для решения своих задач, но и для более «про-

заических» целей: хранения информации, редактирования текстов и даже в быту. Эти пользователи не являются профессиональными программистами, следовательно, средний уровень компетенции пользователя в отношении ЭВМ постоянно снижается.

Получается своеобразная вилка: машины становятся все сложнее, а пользователь — все «глупее». Ликвидировать этот разрыв должна та часть программного обеспечения ЭВМ, которая служит для связи пользователя и ЭВМ и называется *интерфейсом*. С развитием ЭВМ интерфейс должен становиться все более интеллектуальным и решать многие проблемы, которые раньше либо не возникали, либо решались пользователем. К проблемам, которые не возникали, можно отнести проблему распределения задачи на несколько процессоров, а проблема, которая традиционно решалась пользователем, а теперь должна решаться ЭВМ — это проблема программирования задачи по ее спецификации, выраженной на специализированном языке пользователя.

Описанные нами три направления в развитии вычислительной техники: *микроминиатюризация* (качественный фактор), *распараллеливание* обработки информации (количественный фактор) и *интеллектуализация* интерфейса (удобство пользования) являются наиболее важными с точки зрения повышения производительности ЭВМ и ее эффективного применения.



● 1. Как устроены современные ЭВМ

Возможность и необходимость организации параллельной обработки информации обусловили разработку ЭВМ, где одновременно над решением одной задачи могут работать несколько исполнительных устройств. Хотя организация вычисления, идущая «от структуры задачи», более предпочтительна, к настоящему времени еще далеки от совершенства программные и аппаратные средства, позволяющие оптимально «настраиваться» на задачу. Поэтому при решении задач определяющими являются структурная организация вычислительных средств, с помощью которых задачи решаются, степень их специализации. Вот почему вопросы, связанные с рассмотрением архитектуры, занимают в книге и первое, и основополагающее место.

Классификация ЭВМ

Начнем изучение с общей схемы традиционной ЭВМ (см. рис. 1.1), которая в самом грубом приближении состоит из:

а) процессора (Пр);
б) оперативной памяти (ОП);
в) внешней памяти (ВП);
г) устройство ввода-вывода (I/O);
д) устройства управления (УУ).

Процессор — основной элемент ЭВМ, определяющий ее производительность. Он предназначен для выполнения команд из списка команд, который фиксирован для каждой ЭВМ. Команды выполняются над машин-

ными словами, которые могут находиться либо в памяти, либо в специальных регистрах процессоров. Типичными командами, которые присутствуют практически в каждой ЭВМ, являются:

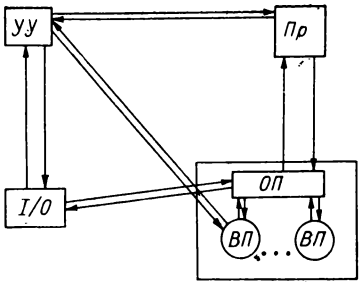


Рис. 1.1

а) сложение (умножение) содержимого двух регистров или регистра и ячейки памяти;

б) логическое сложение или умножение;

в) сдвиг;

г) деление;

д) занесение или считывание в регистр (память).

Обычно процессор современной ЭВМ включает специализированные устройства, каждое из которых выполняет одно из действий. Так, *сумматор* производит сложение, *умножитель* — умножение и т. д.

Память служит для хранения программ и данных и состоит из оперативной и внешней. *Оперативную память* удобнее всего представить как упорядоченную последовательность *ячеек*, в каждой из которых может храниться одно машинное слово. Каждая ячейка имеет *адрес*, по которому можно записать в нее данное или считать содержимое. Если содержимым ячейки является команда, работающая, скажем, с двумя операндами, то структура соответствующего слова такова:

Код операции	Адрес 1-го операнда	Адрес 2-го операнда	Адрес для засылки результата

Здесь адреса могут быть как номерами ячеек памяти, так и именами регистров процессора. Если же в ячейке находится данное, то оно занимает обычно всю ячейку. Наиболее часто используемые типы данных следующие:

а) символ; для представления различных символов используется специальный двоичный код;

б) целое число в двоичной или двоично-десятичной записи;

в) действительное число с фиксированной или плавающей запятой; формат действительного числа, записанного в ячейке, имеет вид

m = мантисса	l — порядок

эта запись обозначает число $m \cdot 2^l$.

Важное значение имеет характеристика, называемая циклом обращения к памяти, или просто циклом памяти. *Цикл памяти* — это время, требуемое для записи в память некоторого значения и считывания его из памяти. Цикл памяти не равен сумме времен, требуемых на запись и на считывание, так как сразу после записи нельзя произвести считывание и сразу после считывания нельзя производить запись: память должна некоторое время «устояться». Цикл памяти определяет скорость работы памяти. Он измеряется в микросекундах (1 мкс $= 10^{-6}$ с) и наносекундах (1 нс $= 10^{-9}$ с).

Оперативная память является быстрой памятью. В любую ее часть можно очень быстро записать информацию и из любой части быстро считать информацию. Эта скорость достигается за счет сложных и дорогих элементов, поэтому объем оперативной памяти обычно сравнительно невелик — около миллиона машинных слов. При решении же сложных задач требуется память значительно бо́льшая. Для того чтобы решить систему линейных уравнений из тысячи уравнений, нужно хранить столько информации, что она не поместится в оперативную память. А в ней еще должны быть программы, системные таблицы и т. д. Поэтому для хранения больших объемов информации вводят так называемую *внешнюю память*. Эта память более медленная, считывать и записывать в нее можно только целыми страницами. Но она располагается на дисках или лентах, количество которых можно увеличивать практически неограниченно. Это служит предпосылкой для возможности решения задач с большим объемом данных.

Важное значение имеет так называемая *виртуальная* (дословно «кажущаяся») память. Часто ее называют еще математической, идея ее состоит в следующем. К оперативной памяти мы можем обратиться из

программы очень просто. Достаточно указать номер ячейки. Если же оперативная память имеет объем, скажем, 1 млн слов, а программисту необходимо записать 2 млн слов данных, в оперативную память они все не помещаются и нужно привлекать внешнюю память. Обращение же к внешней памяти из программы гораздо сложнее. Эффективную работу с ней может организовать только программист высокой квалификации. Виртуальная память — дополнительное сервисное средство, позволяющее программисту работать с памятью так, как будто вся память оперативная. Если объем данных превосходит объем реальной физической оперативной памяти, то работает специальная программа, «скрытая» от программиста, которая переписывает данные во внешнюю память и извлекает их оттуда, когда будет нужно. Таким образом, программист работает с адресами математической памяти, которые затем преобразуются в адреса физической памяти, и только после этого в нее вводится запись или считывание.

Устройство ввода-вывода служит для считывания в память внешних устройств пользователя его программ и данных, а также для вывода пользователю результатов расчета.

Устройство управления организует работу устройств.

Между собой все перечисленные устройства соединены каналами. *Канал* — сложное устройство. Кроме функций передачи информации он выполняет и другие функции: буферизацию, преобразования форматов и типов данных, синхронизацию в случае необходимости и т. д.

В зависимости от размеров, структуры, принципов действия ЭВМ могут быть классифицированы по различным параметрам. Основной, хотя и весьма условный признак классификации — величина ЭВМ. При этом под величиной понимаются не габариты устройств, а производительность процессора и объем памяти. А эти величины могут быть крайне слабо связаны с внешними габаритами. Вспомним, что ЭВМ КРЕЙ-1 занимает объем порядка 1 $м^3$, но по быстродействию она на два порядка превосходит коммерческие ЭВМ типа ИБМ и ЕС, у которых соответствующие исполнительные устройства располагаются на площади в десятки квадратных метров. По величине ЭВМ принято

классифицировать на малые, средние, большие и сверхбольшие. В последнее время в качестве обособленного выделяют класс персональных ЭВМ. К сверхбольшим, или суперЭВМ, относятся машины, которые обычно существует в единичных экземплярах, не запускаются в массовое производство и которые разрабатывались специально для достижения рекордных характеристик на каких-то классах задач. Именно в суперЭВМ наиболее систематическим образом используются современные архитектурные решения.

По характеру области применения ЭВМ можно подразделить на универсальные и специализированные. *Универсальные ЭВМ* обладают достаточно широким диапазоном команд и объемом памяти, чтобы на них более или менее эффективно можно было решить задачи любой предметной области. *Специализированные ЭВМ* ориентированы на некоторый узкий класс задач или даже одну задачу. Их архитектурные особенности таковы, что решение задач вне этого класса весьма не эффективно или невозможно вообще.

Наконец, самый важный признак классификации ЭВМ — степень параллелизма при обработке информации. В этом отношении ЭВМ можно условно подразделить на последовательные, с параллельным вводом-выводом и параллельные. Термин *последовательные* будем применять к ЭВМ, имеющим единственный процессор, который выполняет и операции счета, и операции ввода-вывода. Причем в каждый момент времени он может выполнять только одну из этих операций. По существу, ЭВМ последовательного действия уже давно нет. Как минимум, в любых современных машинах ведется одновременная обработка разрядов слова. Но с учетом вышеприведенной оговорки к чисто последовательным можно отнести, например, машину М-20, давно спятую с производства.

Важная ступень совмещения процессов обработки — введение специализированного процессора ввода-вывода, который занимается только операциями ввода-вывода данных и программ и который может функционировать одновременно с работой центрального процессора. Таким образом, ЭВМ обрабатывает сразу несколько задач, и высвободившееся время центрального процессора используется более эффективно.

Если ЭВМ имеет несколько центральных процессоров, которые обладают возможностью одновременного счета, то она называется *параллельной* или, в другой терминологии, *многопроцессорным вычислительным комплексом* (МВК).

По принципам действия ЭВМ подразделяют на *цифровые* и *аналоговые*. Первые из них используют представление информации в дискретном виде. Прообразом таких устройств являются счеты. Вторые используют для представления информации и расчетов «непрерывные» принципы. В этом смысле простейшим аналоговым устройством является логарифмическая линейка. В последнее время большое развитие стали получать аналого-цифровые, так называемые «*гибридные*» ЭВМ. Они включают устройства обоих типов. Для сопряжения разнотипных устройств служат специальные преобразователи информации из одной формы в другую — аналого-цифровые и цифроаналоговые.

Особенностью цифровых ЭВМ является то, что они способны перерабатывать информацию с очень высокой точностью. Чем больше длина машинного слова, тем выше точность. Аналоговые вычислители выдают только весьма приближенный результат, зато очень быстро. Те же самые вычисления на логарифмической линейке можно выполнить за несколько смещений, в то время как на счетах нужно сделать десятки перебросов костяшек. Поскольку в современных задачах большое значение имеет точность результата, цифровые ЭВМ получили гораздо большее распространение, чем аналоговые. Применение аналоговой техники сконцентрировалось в основном в областях, где нужна, быть может, грубая, но быстрая реакция, например в управлении технологическими процессами.

Параллельные многопроцессорные вычислительные комплексы

МВК, как и однопроцессорные ЭВМ, могут быть классифицированы по многим признакам. При рассмотрении характера параллелизма большую популярность получила классификация Флинна. В соответствии с ней все вычислительные средства делятся на четыре класса: ОКОД, МКОД, ОКМД, МКМД.

Первый класс — ОКОД (одиночный поток команд — одиночный поток данных) — представляет собой обычные последовательные ЭВМ, которые в каждый момент времени обладают возможностью обработки единственного выбора аргументов одной командой. Принципиальная схема такой ЭВМ совпадает со схемой на рис. 1.1.

Второй класс — МКОД (множественный поток команд — одиночный поток данных) — включает конвейерные или магистральные системы.

Системы первого типа называются *конвейерными*, потому что ускорение вычисления в них достигается за счет разделения всей вычислительной работы на последовательность некоторых более специализированных этапов и организации передачи обрабатываемых данных от одного этапа к последующему, аналогично тому, как в производстве на конвейере при изготовлении изделия оно передается от одного рабочего к другому. Один сделает некоторую небольшую операцию, другой — другую, третий — третью операцию, и на выходе конвейера появляется готовое изделие. А так как рабочий, специализирующийся на одной операции, после определенного периода освоения делает ее очень быстро, общая производительность при этом значительно повышается. То же самое можно сказать и о конвейерной обработке информации.

Рассмотрим простой пример сложения двух чисел, скажем, 12,7 + 7,55. Предположим, обычная однопроцессорная машина выполняет сложение за одну единицу времени, или, как еще часто говорят, за один такт. Однако работа по суммированию является для машины довольно сложной. В ней можно выделить по крайней мере три шага. На первом шаге делается выравнивание порядков — т. е. сдвиг второго числа таким образом, чтобы запятые оказались на одном уровне. На втором шаге суммируются дробные части, при этом, быть может, образуется единица переноса. На третьем шаге складываются целые части чисел с прибавлением этой единицы, если она появилась. Точно так же складывает числа и человек. Если для каждой из этих операций выделить отдельный процессор, то суммирование может быть изображено, как показано на рис. 1.2. Так как весь процесс суммирования занимает один такт времени, естественно предположить, что каждый из этих этапов занимает 1/3 такта. И при

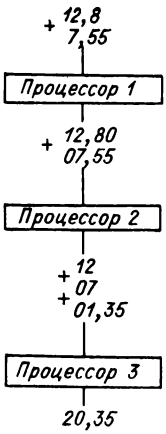


Рис. 1.2

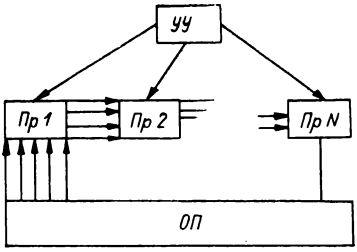


Рис. 1.3

сложении одной пары чисел мы не получаем никакого выигрыша во времени, так как $1/3 + 1/3 + 1/3 = 1$.

Теперь предположим, что нам необходимо просуммировать элементы двух массивов размерности, например, восемь, т. е. получить суммы $A(k)+B(k)$, $k = 1, \ldots, 8$. Пусть суммирование ведется в естественной последовательности: $A(1)+B(1)$, $A(2)+B(2)$, ..., $A(8)+B(8)$. На последовательном процессоре каждое сложение требует одного такта, и все суммы будут получены через 8 тактов. При суммировании же на конвейере (см. рис. 1.2) через 1/3 такта на выходе процессора для выравнивания появляются выровненные числа $A(1)$, $B(1)$, они идут на вход процессора *2*, а процессор *1* освобождается, и в него можно ввести вторую пару чисел $A(2)$, $B(2)$. Еще через 1/3 такта обрабатывающаяся пара $A(1)$, $B(1)$ окажется на выходе процессора *2* и пойдет на вход процессора *3*; выровненная пара чисел $A(2)$, $B(2)$ пойдет на вход процессора *2*, а процессор *1* освободится для обработки третьей пары. Через еще 1/3 такта, т. е. через один такт, если считать с начала обработки, на выходе процессора *3* появится окончательный результат (см. рис. 1.2), а результаты последующих сложений будут выдаваться через каждую треть такта. Их остается семь, поэтому общее время для суммирования всех восьми чисел $1 + 7\frac{1}{3} = 3\frac{1}{3}$ такта.

По сравнению с последовательной обработкой имеем выигрыш во времени более чем в два раза. В общем случае, если длина конвейера p, каждый этап требует $1/p$ такта и обрабатывается массив длины M,

то время обработки будет равно $1+(M+1)/p$. Таким образом, на больших массивах конвейерный процессор считает почти в p раз быстрей, чем обычный. Принципиальная схема конвейерного устройства общего вида изображена на рис. 1.3.

Третий класс МВК — ОКМД — (одиночный поток команд — множественный поток данных) — включает МВК, принципиальная схема которых изображена на рис. 1.4, *а*. К МВК этого типа относятся *матричные* процессоры и частично *ассоциативные устройства* (устройства ассоциативной памяти). МВК типа ОКМД состоят из N идентичных процессорных элементов, работающих под управлением одного УУ. Важная особенность УУ заключается в том, что на каждом такте оно предписывает исполнение одной и той же команды на всех N процессорах. Таким образом, одна команда выполняется над различными данными (см. рис. 1.4, *б*). Оперативная память является, как правило, *разделенной* между процессорами. Каждый процессор имеет возможность быстро обратиться к своему участку памяти, в то время как обратиться к памяти соседа значительно сложнее. Зато между собой процессоры могут быть соединены некоторыми каналами, позволяющими передавать данные соседям и получать их от соседей непосредственно, т. е. без промежуточного обращения к памяти.

Наконец, последний класс систем (рис. 1.5) — МКМД (множественный по-

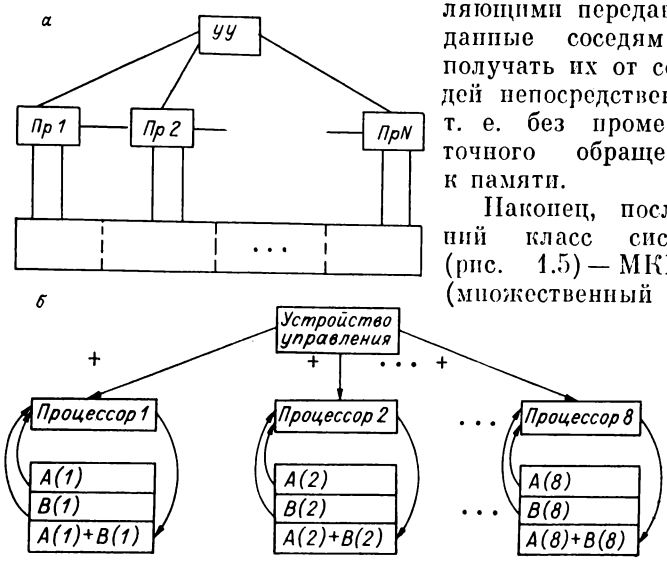


Рис. 1.4

ток команд — множественный поток данных) — включает МВК из N процессоров, каждый из которых имеет собственное устройство управления. Память может быть как разделенной так и *общей*. Обычно каждый процессор имеет свою небольшую так называемую *локальную память* и доступ к некоторой общей для всех процессоров оперативной памяти. Процессоры могут быть как идентичными, так и разнородными. В первом случае система называется *однородной вычислительной системой* (ОВС), а во втором — *неоднородной*. Устройства управления рассылают на каждом такте, вообще говоря, различные команды для всех процессоров. Поэтому имеется множественность как данных, так и команд. Различные УУ могут быть связаны между собой и влиять на работу друг друга. Может быть выделено также некоторое специальное управляющее устройство, в том числе и одно из УУ, которое организует работу других управляющих устройств. В этом случае система называется *централизованной*, а иначе — *децентрализованной*.

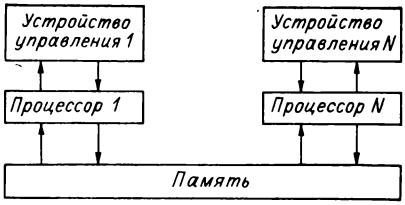


Рис. 1.5

Введем некоторые общие понятия для систем любого из перечисленных классов.

Система называется *программно-коммутируемой*, или *программно-настраиваемой*, если она обладает способностью изменять топологию связей между процессорами, а также между процессорами и другими устройствами в соответствии с командами, которые можно ввести в программу. Программная коммутируемость может быть физической, настройка действительно повлечет пересоединение некоторых каналов, и виртуальной, т. е. коммутация будет обозначать подготовку некоторого оптимального пути для передачи данных и сообщений по фиксированной сети связей.

При описании систем типа МКМД мы упоминали про локальную память процессора. Обычно такая память делается очень быстрой и называется *сверхоперативной*, или *кешем*. Часто она имеет стековую струк-

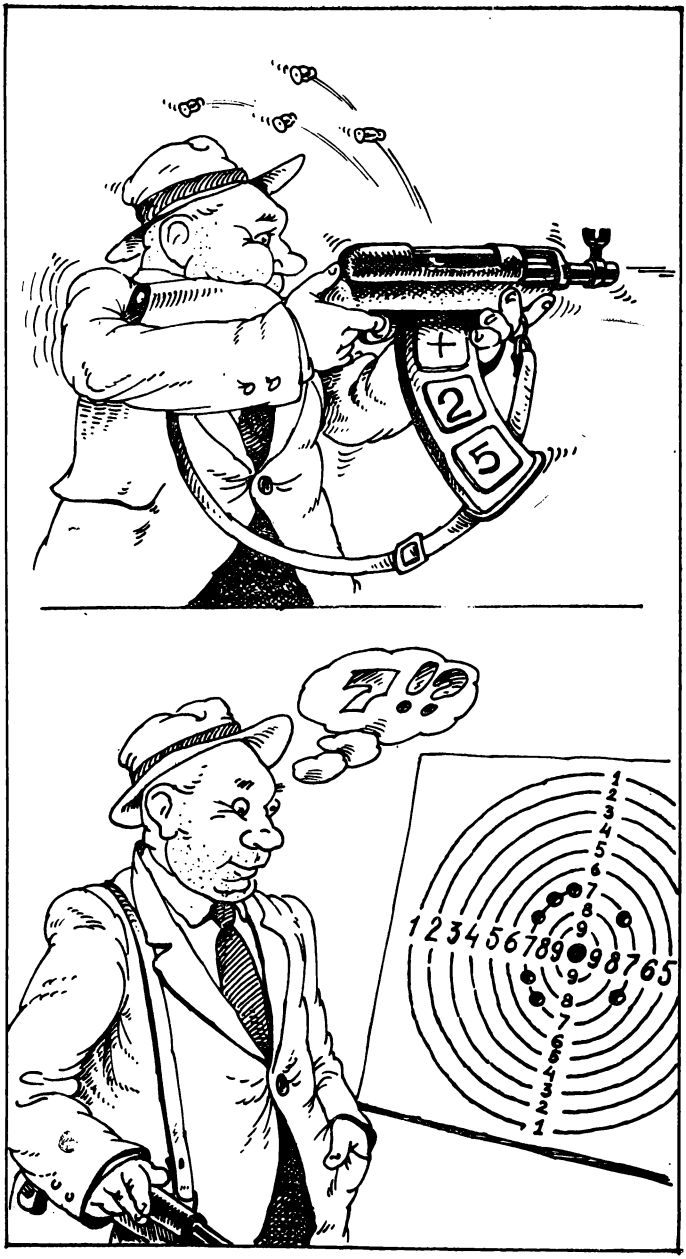
туру. Быстро обращаться можно только к верхушке стека, т. е. в данном случае может быть выполнена только команда 1 над парой ее операндов. После этого место в верхушке стека занимает команда 2. Если организовать оперативную подготовку команд и данных и представить их в виде стека, расположив в порядке исполнения, то программа будет исполняться очень быстро, так как обращение к верхушке стека занимает ничтожно малое время. Ситуацию можно сравнить со стрельбой из автоматического оружия. *Магазин*, а именно так называется стек, набивается командами с соответствующими операндами и «выстреливается» в порядке расположения «зарядов». Но если оказалось, что какой-то заряд не подготовлен, например не успел поступить один из операндов, то магазин «заедает». Чаще всего его вынимают, т. е. процесс сбрасывается со счета в оперативную память.

В заключение опишем некоторые из основных характеристик, по которым будем сравнивать конкретные комплексы. Самая важная характеристика с точки зрения параллельного программирования — число процессоров. Однако не всегда однозначно можно сказать, что какая-то система имеет столько-то процессоров.

Во-первых, многие системы могут существовать в нескольких конфигурациях. Так, система ПС-2000 может иметь до $8k$ процессоров, где $k = 1, 2, \ldots, 8$. Система ILLIAC IV разрабатывалась как 64×4-процессорная, но по экономическим соображениям в ней была только одна матрица из 64 процессоров. Таким образом, при описании системы мы будем указывать не число, а диапазон возможного числа процессоров.

Во-вторых, не всегда понятно, что считать процессором. Так, можно ли считать процессором отдельный элемент конвейера в конвейерных системах? Это оказывается неудобным, так как функции конвейера бывают распределены между довольно большим числом специализированных устройств. Возникает вопрос, что же считать элементом конвейера? При рассмотрении конвейерных систем мы будем указывать, как правило, число различных конвейеров и их функции.

Важной характеристикой МВК является *объем памяти*. По мере возможности мы будем называть объемы сверхоперативной, оперативной и виртуальной памяти. Объем выражается в специальных единицах: К и М.


+
2
5
1 2 3 4 5 6 7 8 9 9 8 7 6 5

К — кило — равно 1024, М — мега — равно 1024^2. Например, 8 Мбайтов означает $8 \cdot 1024^2$ байтов.

Характеристикой, для улучшения которой собственно и были созданы МВК, является *быстродействие*, или *производительность МВК*. Эта характеристика наиболее сложна в смысле ее объективного определения и вычисления. Она обычно измеряется в числе операций, производимых в секунду времени. Однако дело в том, что:

а) различные операции занимают различное время; например, время деления бывает на порядок больше, чем время сложения;

б) даже одинаковые операции, но реализованные над разными типами данных, занимают разное время; например, сложение целых чисел выполняется быстрее, чем действительных с фиксированной запятой, которое, в свою очередь, быстрее, чем для чисел с плавающей запятой;

в) при одинаковых типах данных у различных МВК длина машинного слова различна, она колеблется от 12 до 64 разрядов; нередко операция над более коротким словом требует меньшего времени;

г) при одинаковых типах данных и одинаковом размере слова время операции зависит от длины вступающих в операцию операндов; для коротких операндов оно меньше;

д) время очень существенно зависит от цикла памяти, в которой находятся операнды.

В условиях такой многопараметрической неопределенности был предложен ряд практических методов более или менее объективной оценки быстродействия. Обычно они вовлекают в рассмотрение испытания МВК на некоторых смесях задач — универсальных для универсальных ЭВМ или специальных — для проблемно-ориентированных процессоров. По возможности мы будем показывать, в каком именно смысле понимается быстродействие того или иного МВК.

Было нетрудно заметить, что ЭВМ типа МКОД и ОКМД имеют наибольшее быстродействие при обработке участков программ, представляющих собой длинные серии однотипных операций. Для конвейерных ЭВМ оказывается надолго загруженным конвейер, а для матричных имеется возможность полностью использовать всю матрицу процессоров. При скалярных

вычислениях, т. е. тогда, когда тип операций постоянно меняется, ЭВМ этих классов сразу резко теряют в производительности. Ориентация на обработку длинных серий однотипных операций определила еще одно их общее название — *векторные* ЭВМ. Им будет посвящен следующий раздел.

Конвейерные и матричные ЭВМ

История развития многопроцессорных комплексов такова, что ЭВМ того или иного класса первоначально разрабатывались, так сказать, в чистом виде. Они служили для отработки основных концепций, проверки заложенных в них идей и изготовлялись, как правило, в одном или нескольких экземплярах. В качестве примеров рассмотрим наиболее известные конвейерные системы STAR-100 и Cray-1.

Система STAR-100 (String ARray computer) разрабатывалась фирмой Control Data Corp (CDC) с 1965 по 1973 г. Стоимость разработки оценивается в 15 млн дол; производительность — до 100 млн опер/с.

Основной архитектурной особенностью системы является наличие трех конвейерных устройств. Первое устройство служит для сложения и умножения с плавающей запятой. Второе — для сложения с плавающей запятой, умножения и деления с фиксированной запятой, а также для извлечения корней. Третий конвейер служит для коротких специальных операций над 12-разрядными словами. Время цикла (такта) каждого конвейера 40 нс. Основную вычислительную работу проделывают первые два конвейера, поэтому производительность подсчитывается только на них. Так как эти конвейеры обеспечивают получение каждые 40 нс двух результатов по 64 разряда, или четырех по 32 разряда, то производительность системы равна 50 млн опер/с для 64-разрядных операций. Конвейер для аддитивных операций сложения и вычитания с плавающей запятой имеет длину 4, для умножения с плавающей запятой — 8. Таким образом, время выполнения отдельных операций сложения и умножения 160 и 320 нс соответственно. Если считать, что в среднем операция занимает 200 нс и программа «плохо векторизована», т. е. в ней нет больших последовательностей однотипных операций, все операции не векторного

типа, а скалярного, то производительность должна бы упасть всего в пять раз. Но на самом деле производительность падает гораздо ниже. Это происходит из-за того, что на настройку конвейера тратится очень большое время — 76 тактов. Поэтому в подобных системах прибегают к разного рода ухищрениям — делают отдельные специализированные процессоры для скалярных операций, выполняют упреждающий просмотр программы с целью выявления однотипных операций, группировку операций при трансляции — векторизуют программу (см. подраздел «Общие замечания и примеры»).

Если говорить о практически достигаемой производительности STAR-100, то по многим оценкам производительность 50 и 100 млн опер/с никогда не была достигнута. В лучших случаях для хорошо векторизованных задач достигалась производительность 30 и 60 млн опер/с. Поэтому усредненно принято считать, что производительность STAR-100 — около 50 млн опер/с на хороших задачах. Это, конечно, весьма высокая производительность, если учесть, что производительность современных универсальных больших ЭВМ редко превосходит 1 млн опер/с.

Память системы включает 512К 64-разрядных слова, т. е. имеет объем около 4 Мбайтов. Цикл памяти сравнительно высок для подобных систем — 1,28 мкс. Есть возможность виртуальной адресации — виртуальная память имеет объем 4×10^{12} слов. Для «общения» конвейеров имеется быстрая регистровая память емкостью в 256 слов.

Система Cray-1 разработана фирмой CRC (Cray Research Corp.), которую основал главный конструктор фирмы CDC С. Крей в 1972 г. после выхода из нее. Система рассчитана на задачи аэрогидродинамики, в частности для моделирования погоды, хотя является достаточно универсальной и подходит для других задач, включающих массовые векторные вычисления. При разработке Крей исходил из ряда основных принципов. Во-первых, он делал большую ставку на миниатюризацию (вспомним рассуждения об увеличении быстродействия). Поэтому машина при среднем быстродействии 80 млн опер/с занимает общий объем около 1 $м^3$, выполнена в виде 12 стоек высотой два метра. Во-вторых, Крей был сторонником сосредоточения

в одних руках всей разработки: от логических схем до элементной базы. Это позволило ему в процессе разработки провести единую линию конструирования. В-третьих, машина выполнена на проверенной элементной базе, что очень существенно сказалось на степени надежности и «сбалансированности» работы системы.

Система имеет 12 функциональных конвейерных устройств, объединенных в 4 спецпроцессора: для адресных, скалярных, векторных операций и операций с плавающей запятой. Все 12 конвейеров могут работать параллельно и имеют такт 12,5 нс. Особенностью конвейеров является то, что они короткие, это позволяет сократить время на разгон конвейера; они могут группироваться в цепочки, передавая информацию между собой через регистры без отсылки в память. Оперативная память системы имеет скорость 1 Мбайт с циклом около 50 нс. Регистровая память имеет в совокупности емкость около 500 байтов и цикл 6 нс.

Рассмотрим теперь три системы типа ОКМД: ILLIAC-IV STARAN и ПС-2000. Напомним, что эти системы характеризуются наличием нескольких (иногда очень большого количества) однотипных процессоров (в этом смысле они являются однородными), работающих от общего устройства управления. Таким образом, на каждом такте все процессоры выполняют одну и ту же команду.

Наиболее ранней и самой известной среди систем типа ОКМД является матричный спецпроцессор ILLIAC-IV (ILLInois Array Computer), разработанный Иллинойским университетом совместно с фирмой Бэрроуз (Burroughs). Принципиальная схема одного из квадрантов процессора изображена на рис. 1.6.

Вначале предполагалось, что процессор будет состоять из четырех квадрантов по 64 процессорных элемента. Однако в процессе разработки из-за непредвиденного увеличения ее стоимости было произведено перепроектирование, и в действительности система содержала лишь один квадрант. Процессоры соединены каналами передачи данных каждый со своим левым, правым, верхним и нижним соседом. Размер слова 64 разряда. Каждый процессор имеет локальную память в 2 Кслов, 4 быстрых регистра и вполне универсальный набор команд. Таким образом, он является небольшой ЭВМ. Цикл локальной памяти 300 нс. Об-

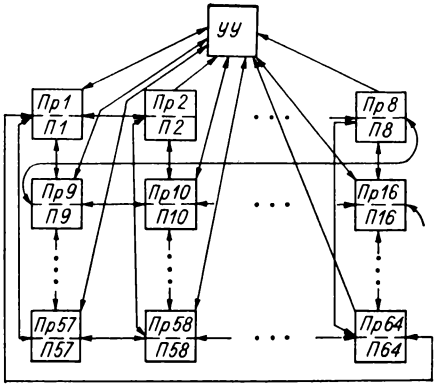


Рис. 1.6

щая для всех процессоров память — дисковая с циклом 20 мс, объемом $2 \cdot 10^9$ битов. Следующий уровень — барабанная лазерная память объемом 10^{12} битов. Эта память представляет собой барабан, покрытый тонкой металлической пленкой. При записи лазер выжигает в ней микроскопические отверстия. Таким образом, в память можно записывать только один раз и стирать из нее нельзя. Она служит для хранения архивов и больших системных программ.

Общее устройство управления выполняет, кроме обычных функций расшифровки и рассылки команд, функции обмена данными между процессорами, защиты памяти, маскирование и т. д. Под *маскированием* понимается возможность дать указание произвольной группе процессоров не выполнять операцию какого-либо такта. Этот механизм позволяет организовать обработку условных переходов над массивами.

Стоимость разработки ILLIAC-IV оценивается в 40 млн долларов. Одна из таких машин была установлена в Калифорнийском отделении NASA — Национального центра космических исследований. Высокая стоимость разработки и специализация только на матричные вычисления сделали ее непопулярной для широкого пользователя. В настоящее время научные программы с ILLIAC-IV выполнены и все машины демонтированы.

В системах типа ОКМД имеется одно требование — общий поток команд, и не накладываются никакие ог-

Таблица 1.1

№ п/п	Фамилия	Год рождения	Должность	Ученая степень	Зарплата
1	Иванов	1940	Зав. лаб.	Канд. техн. наук	320
2	Петров	1945	Мл. научн. сотр.	Канд. физ.-мат. наук	180
3	Сидоров	1950	Инженер	Без степени	120

раничения на число входящих в нее однородных процессоров. Таким образом, число этих процессоров и их возможности могут колебаться в больших пределах от универсальной быстродействующей ЭВМ до преобразователя, производящего некоторые простейшие операции, например логическое сложение и умножение. При этом простых преобразователей может быть большое число. В вырожденном случае каждый преобразователь может производить только элементарные операции выборки и засылки, т. е. осуществлять лишь функции памяти. Однако на большом квадранте параллельно действующих исполнительных устройств эти возможности могут дать новый эффект — появится механизм выборки из массива таких запоминающих устройств группы элементов по каким-то признакам, которыми характеризуется содержимое. В этих случаях говорят об *ассоциативной выборке* и *ассоциативной памяти* в противоположность адресуемой памяти.

Простейший пример использования ассоциативной выборки дает работа с таблицами. Пусть в таблицу сведены данные отдела кадров о сотрудниках какого-либо института (табл. 1.1). Если нам нужны фамилии всех сотрудников 1945 г. рождения, то обычная память предполагает последовательный поиск, в то время как ассоциативная память за один такт выводит все номера строк путем выставления некоторого признака в тех строках, в которых во второй колонке стоит «1945». Запросы к ассоциативной памяти могут быть и сложнее. Например, можно вывести фамилии всех инженеров — не кандидатов наук, зарплата которых находится между 120 и 160 руб., и т. д.

Наличие развитой ассоциативной памяти является главной особенностью так называемых ассоциативных

систем. Одной из наиболее популярных ассоциативных систем является система STARAN. Процессор STARAN состоит из набора ассоциативных матриц — от 1-й до 32-х размером 256×256 битов каждая. Имеется возможность ассоциативной выборки как по строкам, так и по столбцам матрицы. Каждой строке матрицы соответствует простой процессорный элемент, имеющий свой канал ввода-вывода. Все элементы работают «под руководством» одного устройства управления, которое имеет собственную память и обладает возможностью маскировки. Между собой и с внешними устройствами матрицы соединяются через буферную память объемом 2—4 Кслов с циклом 150 нс. Система STARAN работает под управлением мини-машины PDP-11, чья память объемом 16—32 Кслов также используется процессором через устройство управления. Кроме задач типа быстрого ассоциативного выбора система может решать и другие интересные задачи. Так, одна из первых систем STARAN следила за авиадвижением в районе аэропорта Филадельфии. Она могла держать под контролем одновременно 144 самолета, находящихся от аэропорта в радиусе 88 км.

Оценка производительности систем является делом весьма сложным, так как совершенно неясно, что принять в них за операцию. Однако сравнительный анализ на ряде задач с другими системами позволил сделать вывод о том, что быстродействие STARAN около 300 млн опер/с.

На основании опыта решения многих задач была выполнена модернизация системы STARAN. В новой системе STARAN-E были существенно увеличены емкость и скорость работы памяти.

В заключение остановимся более подробно на отечественном матричном процессоре ПС-2000 (рис. 1.7), машина принята Государственной комиссией в конце 1980 г.

Основной компонентой системы является совокупность процессоров (Пр), которых может быть в зависимости от конфигурации $8i$, $i = 1, 2, \ldots, 8$. Минимальная конфигурация — 8 процессорных элементов, а далее система может наращиваться модулями по 8 процессоров до 64 процессорных элементов. Каждый процессорный элемент — универсальная машина, имеет достаточный выбор быстрых регистров и локальную опе-

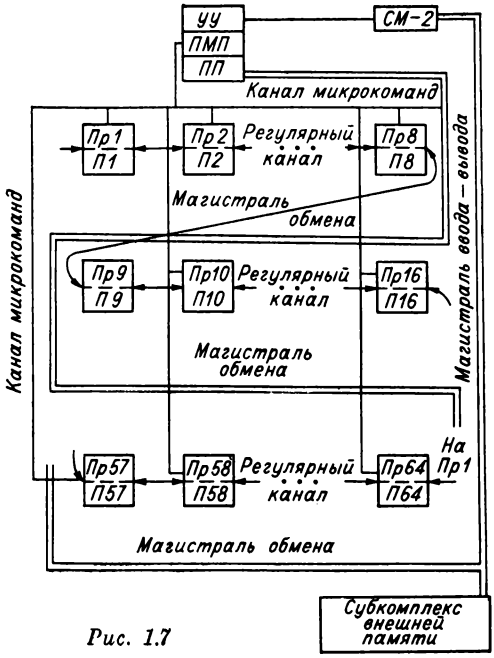


Рис. 1.7

ративную память (П) объемом либо 4 Кбайтов, либо 16 Кбайтов. Цикл памяти в первой конфигурации 640, во второй 960 нс. Производительность одного процессора 3 млн опер/с, в формате 24-разрядного слова, причем это для операций сложения и вычитания с фиксированной запятой. На операциях умножения и деления с плавающей запятой производительность падает почти на порядок.

Таким образом, считается, что в максимальной конфигурации ПС-2000 достигает производительности 200 млн так называемых коротких операций в секунду, а средняя ее производительность на «хороших» задачах — 5—10 млн опер/с. Все процессоры работают под управлением общего устройства управления (УУ). Система является *микропрограммируемой*. Это значит, что по отношению к ней имеет смысл говорить не об одном, а о двух уровнях машинных команд: *микрокоманд* и команд. Микрокоманды оперируют в терминах регистров, ячеек, отдельных команд запоминающих элементов, адресов и т. д. Программирование в микро-

командах для такой сложной многопроцессорной системы, как ПС-2000, является весьма тонким делом и доступно только опытным системным программистам, хорошо знающим архитектуру системы. Они пишут в микрокомандах нижний уровень матобеспечения системы — микропрограммы.

Каждая микропрограмма реализует некоторую «популярную» функцию, например $\sin(x)$, деление с плавающей запятой и т. д. и оформляется как команда процессора. Набор команд, включающий сотни команд за каждой из которых стоит микропрограмма, поставляется пользователю вместе с системой. Программирование в командах оперирует уже в понятиях более высокого уровня, и программисту не обязательно знать очень точно структуру системы. Достаточно иметь общее представление. Для целей микропрограммирования УУ имеет две памяти: ПМП — память микропрограмм (они же — реализация команд) и ПП — память программ, написанных в командах. При вычислении УУ обращается к очередной команде программы, находит реализующую ее микропрограмму и «раскручивает» эту микропрограмму, рассылая ее микрокоманды по процессорам. Процессоры же могут воспринимать только микрокоманды, но не команды. Для обеспечения быстрой реализации микропрограмм память ПМП делается с максимально низким циклом. Поэтому из-за высокой стоимости эта память сравнительно небольшого объема. В ПС-2000 память ПМП имеет объем 16К 64-разрядных микрокоманд.

Если учесть, что в среднем микропрограмма занимает 10—20 микрокоманд, то в память ПМП может быть записано около тысячи микропрограмм, т. е. пользователю может быть представлено около тысячи команд высокого уровня. На самом деле, как правило, таких команд общематематического характера будет предоставлятся около 200. Предполагается, что остальные пользователь сформирует сам в зависимости от своей предметной области. Память ПП для программ имеет объем 4К либо 16К 24-разрядных слов, что достаточно для хранения в ней большой (до 1 тыс. строк) программы. В этой же памяти хранятся некоторые общие для всех процессорных элементов данные.

Устройство управления рассылает микрокоманду на все процессоры по специальному каналу микрокоманд,

В соответствии с микрокомандой все или некоторые процессорные элементы могут выполнить операцию над регистрами одного для всех процессоров наименования, или регистром и ячейкой памяти с одним для всех процессоров номером, или номером, «регулярно» зависящим от номера процессора, так называемый фигурный доступ к памяти. Например, можно обратиться к ячейке с адресом i, где i — номер процессорного элемента. Важной особенностью процессора является то, что команды обработки управления могут быть совмещены. Так, сложение может быть совмещено с пересылкой информации соседнему процессору. Кроме того, могут быть совмещены обработка и «подкачка» данных в локальную память, если при обработке к ней нет обращений.

Между собой процессоры соединены двумя способами. Быстрая передача информации осуществляется через так называемый регулярный канал, которым каждый процессор соединен с двумя своими соседями, а последний процессор соединен с первым. Таким образом, все процессоры соединены регулярным каналом в кольцо, и в этом смысле система не является матричной, а является, скорее, векторной или линейной. Если вспомнить матричный процессор ILLIAC-IV, то там в дополнение есть регулярные соединения с верхним и нижним соседом. Таким образом, сеть регулярных каналов образует матрицу. По регулярному каналу за один такт можно «сдвинуть» одно слово информации на один процессор влево или на один процессор вправо. Некоторые сдвиги при желании можно замаскировать, по этому же каналу можно передать информацию от произвольного i-го к произвольному j-му процессору, затратив не более $N/2$ тактов, где N — число процессоров. Регулярный канал может разбиваться на два, четыре и более подколец. Минимальное кольцо — 8 процессоров.

Вторым способом процессоры могут коммутировать через общую *магистраль обмена*. Она позволяет передать информацию УУ либо выбранного микропрограммным способом процессорного элемента во все либо в адресованные процессорные элементы. По этой же магистрали возможна пересылка из выбранных микропрограммно процессорных элементов в УУ.

Система ПС-2000 не может работать автономно. Она работает в комплексе с мини-машиной, например СМ-2 и находится у этой машины на правах периферийного устройства. СМ-2 имеет два процессора, поэтому весь комплекс СМ-2 — ПС-2000 представляет собой неоднородную вычислительную систему. Вся информация и все программы в ПС-2000 идут через СМ-2. Между ними есть канал небольшой пропускной способности, поэтому в ПС-2000 эффективно решаются только такие задачи, в которых не требуется частый обмен информацией с внешней памятью. Кроме памяти СМ-2 система может использовать так называемый субкомплекс внешней памяти, с которым может соединяться напрямую, минуя СМ-2. Он состоит из 1—2 дисков объемом 29М байтов и процессора ввода-вывода.

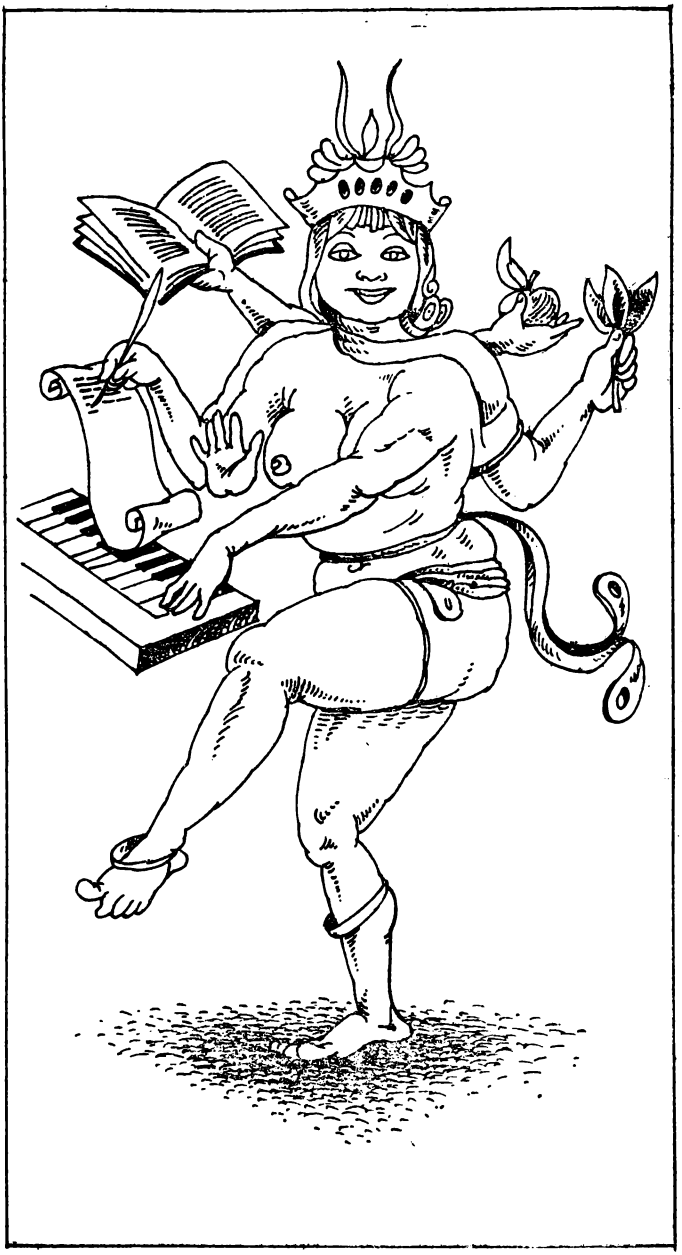
Система ПС-2000, как, впрочем, и другие рассмотренные в разделе системы, не является универсальной. Она ориентирована на классы задач, которые требуют обработки больших массивов по регулярным алгоритмам. Современным МВК универсального назначения посвящается следующий раздел.

Современные многопроцессорные комплексы

Истоками такого типа архитектуры, как вычислительные комплексы типа MKMD, являются МВК фирмы Бэрроуз.

Американская фирма Бэрроуз (Burroughs) — одна из самых известных в плане технического экспериментирования и введения новых архитектурных решений. Достаточно вспомнить созданный ее процессор ILLIAC-IV. С начала 60-х годов фирма начала разрабатывать серию высокопроизводительных ЭВМ, использующих новые архитектурные принципы. Наиболее известными моделями этой серии были B5500, B6700, B7700 и последние — B6800, B7800.

Рассмотрим основные характеристики и особенности системы B6700, хотя ее производительность в несколько раз ниже, чем последних двух систем. Система B6700 была выпущена фирмой в 1971 г. В ее состав входят от 1 до 3 центральных процессоров с производительностью до 1 млн опер/с каждый, на практике до 0,5 млн. Кроме того, может осуществляться параллельный ввод-вывод, для которого может быть подключено



до 3 процессоров. Оперативная память с циклом обращения от 500 до 1500 нс может набираться модулями по 16К или 64К до 1М 48-разрядных слов. Система имеет разветвленную сеть периферийного оборудования с максимальным числом терминальных устройств 128. В их число входят память, графопостроитель, АЦПУ и т. д.

Одно из важных архитектурных решений машин серии Бэрроуз — высокий уровень машинного языка. Если традиционные команды машинного языка оперируют с регистрами и ячейками памяти, задают над их содержимым элементарные операции: суммирование, сдвиг, пересылку и т. д., то команды машин серии Бэрроуз имеют уровень операторов языка Алгол, а в некоторых случаях даже более высокий. Так, они включают команды работы с процедурами, строками, стеком сверхоперативной памяти и т. д. Высокий уровень языка позволяет писать более короткие и обозримые пользовательские программы в *автокоде*, быстро реализовать трансляторы с языков высокого уровня, при этом время трансляции намного меньше, чем в трансляторах на традиционные машинные языки. Ряд команд МВК Бэрроуз реализован аппаратно, другие команды высокого уровня имеют микропрограммную реализацию.

Еще одной важной особенностью машин этой серии является введение и эффективное использование тегов. *Тегом* называется группа разрядов машинного слова, которая не участвует в операциях, а служит для указания *типа* информации, записанной в этом слове. По внешнему виду содержимого слова традиционной ЭВМ никак нельзя понять, является ли оно, скажем, командой, числом с плавающей запятой или символьным значением. В машинном слове МВК Бэрроуз такая информация указывается в разрядах тега. Это позволяет облегчить поиск и намного снизить число ошибок из-за неправильного обращения к памяти. В модели B6700 под тег отведено три разряда. Стоимость разработки системы по различным источникам от 1,5 до 9 млн дол.

Отечественные ЭВМ стековой архитектуры представляют машины серии Эльбрус: Эльбрус-1 и Эльбрус-2. Первая партия машин Эльбрус-1 была выпущена в 1978 г., а в 1980 г. комплекс был сдан Государственной комиссии. В состав МВК может входить от одного до десяти центральных процессоров с производи-

тельностью 1—1,5 млн опер/с. Формат слова — 32, 64 и 128 разрядов; емкость оперативной памяти — от 576 до 4608 Кбайтов в зависимости от комплексации, цикл — 1200 нс. Система ввода-вывода может содержать до четырех автономно работающих процессоров ввода-вывода. К системе можно подключать спецпроцессор СВС, имеющий ту же систему команд, что и машины БЭСМ-6 (интегральная БЭСМ-6). Его производительность — до 2 млн опер/с.

Следующая модель — Эльбрус-2 — имеет производительность около 10 млн опер/с на процессор и число процессоров от 1 до 10.

Одним из ранних архитектурных решений МВК явились ОВС — однородные вычислительные системы. В каком-то смысле ОВС являются развитием матричной архитектуры, так как состоят из одинаковых процессорных элементов, соединенных регулярным образом. Отличие заключается в том, что каждый процессор имеет свое УУ. Это обеспечивает повышенную эффективность при решении задач с ветвящимися процессами, а также упрощает задачу распределения ресурсов в системе при обслуживании потока задач различных размеров. В качестве элементарного процессора ОВС берется, как правило, микропроцессор или мини-машина.

Вычислительные комплексы типа MKMD в целом имеют преимущества над конвейерными и матричными ЭВМ. Однако еще больший эффект можно получить, если объединить в единый комплекс все типы архитектур. Идеальный вариант архитектуры представляет собой комплекс, в котором в асинхронном режиме могут работать и взаимодействовать несколько типов процессов: скалярные, матричные, спецпроцессоры, ориентированные на отдельные классы задач и достигающие на них сверхвысокой производительности. При этом каждый из процессоров является конвейерным. Такая организация МВК позволяет использовать преимущества всех типов архитектур.

Один из примеров МВК смешанной архитектуры — система ПС-3000 (рис. 1.8). Система ПС-3000 включает четыре универсальных процессора, работающих над общей памятью в режиме SIMD. Каждая пара этих процессоров делит между собой специализированный векторный процессор, ориентированный на обработку длинных серий однотипных операций. Предполагается сле-

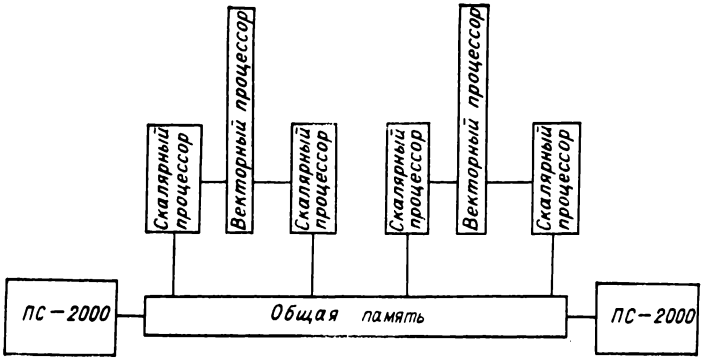


Рис. 1.8

дующая организация вычисления. В то время как первый процессор занят обработкой своих скалярных вычислений, т. е. нерегулярных участков программ, состоящих из смесей различных операций, второй процессор использует векторный процессор для обработки своих векторных данных. А затем они меняются ролями — первый из процессоров «захватывает» векторный процессор, а второй осуществляет обработку своих скалярных участков. Кроме того, к системе могут быть подключены два комплекса ПС-2000. Таким образом, объединяются три типа архитектур.

В настоящее время наибольшие успехи в развитии высокопроизводительной вычислительной техники достигнуты за рубежом. Рассмотрим некоторые из комплексов, выпускаемых фирмами США и Японии. Сводная информация о них приведена в табл. 1.2. Прокомментируем ее.

Прежде всего заметим, что ведущее место в производстве сверхвысокопроизводительных средств попрежнему занимает фирма CRC. Ею давно превзойден миллиардный рубеж, а новая анносируемая Креем разработка — система Cray-4 — будет иметь производительность 128 млрд опер/с. Выпуск этой машины намечен на начало 90-х годов. Однако фирма CRC уже не имеет такого монопольного положения на рынке суперЭВМ, как это было раньше. Во-первых, уже сейчас у нее имеется много конкурентов, в частности фирма ETA System. Ее модель ETA 10-E превосходит по быстродействию выпускаемые в настоящее время комплексы фир-

Таблица 1.2

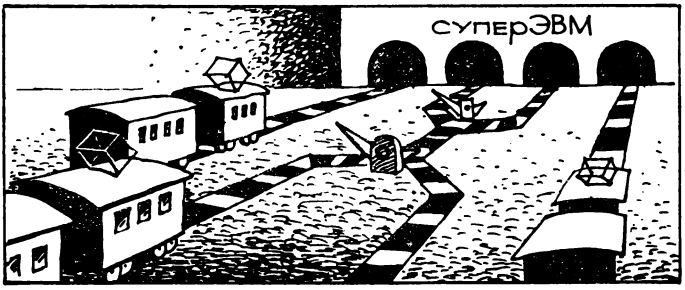
ЭВМ	Фирма	Максимальное число процессоров	Максимальное быстродействие (млн опер/с)	Цена, тыс. дол.	Число поставленных машин (к началу 1988 г.)
X-MP	CRC	4	1200	2500—16000	123
Cray-2	CRC	4	1800	20 000	11
ETA 10-P	ETA System	2	750	995	1
ETA 10-E	ETA System	8	6857	более 5500	2
VP-30E	Фуджитсу	1	220	—	16
VP-400	Фуджитсу	1	1700	—	2
S-810/5	Хитачи	1	160	—	20
S-810/10	Хитачи	1	315	—	20
IBM 3090	IBM	6	—	230	200
SX-1ER	NEC	1	330	—	14
C 120	Convex	1	20	249	260
Matrix 1	Saxpy	32	1000	900—1800	1
SCS-40	SCS	1	44	595	30
Vortex	Sky	1	10	10—50	200
GP 1000	BB&N	256	125	75—2500	100
Elex 32	Flexible	20	80	110—600	16
Series 10 000	Apollo	4	140	70—80	—

мы CRC. Во-вторых, из фирмы CRC вышел один из ее главных конструкторов Стив Чен. Он и еще около 45 бывших сотрудников фирмы создали собственную фирму SSI (Supercomputer Systems Inc.) и намереваются выпустить свою машину с производительностью 128 млрд опер/с раньше, чем появится Cray-4. Наконец, в-третьих, появилось много фирм, которые не ставят целью создание сверхвысокопроизводительных систем, однако выпускают компактные и дешевые комплексы, имеющие достаточно высокую производительность для решения многих задач.

Одна из наиболее важных характеристик МВК с экономической точки зрения — это стоимость одной операции, т. е. отношение цены к производительности. Из табл. 1.2 видно, что для машин фирмы CRC эта характеристика равна примерно 10 дол. В то же время для таких суперминиЭВМ, как Vortex и Series 10000 фирмы Apollo, она в 10—20 раз ниже.

Далее, анализируя информацию табл. 1.2, можно заметить, что в то время как американские фирмы выпускают в основном многопроцессорные комплексы, японские фирмы (Фуджитсу, Хитачи, NEC) ограничиваются однопроцессорными ЭВМ. Высокую производительность они достигают за счет высокопроизводительной элементной базы. Таким образом, у них есть большой неиспользованный резерв. Кстати, стоимость японских суперЭВМ не указана, так как они продают, как правило, машинное время, а не сами машины. Стоимость времени — от 30 до 80 млн иен в месяц.

В табл. 1.2 указаны фирмы, выпускающие дешевые небольшие, но достаточно производительные мини-суперЭВМ. Таких фирм сейчас около двадцати. Следует полагать, что возрастающая конкуренция на рынке ЭВМ приведет к появлению вычислительных машин, спроектированных на принципиально новых архитектурных принципах и элементной базе.



2. Как задать параллельные вычисления

Итак, пусть у нас имеется многопроцессорный вычислительный комплекс. Для того чтобы он начал решать предлагаемую задачу, нужно этот комплекс, а правильнее сказать задачу, соответствующим образом запрограммировать. Всю совокупность вопросов, возникающих при создании программ для МВК, изучает область науки, которая называется *параллельное программирование.*

Общие замечания и примеры

Если пытаться дать прямую расшифровку термина «параллельное программирование», то его можно понимать в узком смысле слова и в широком. Эта двойственность возникает из-за того, что сам термин «программирование» имеет два смысла: узкий и широкий. «Программирование» в узком смысле — это написание программ, предназначенных для исполнения на ЭВМ. Соответственно «параллельное программирование» в узком смысле — это написание программ для ЭВМ в специальном параллельном языке программирования. Параллельный язык содержит операторы для задания «размножения» вычисления. Как только процесс вычисления достигнет такого вида оператора, он распадается на несколько процессов, каждый из которых может исполняться на отдельной ЭВМ или процессорном элементе. Таким образом, необходимым условием параллельного вычисления по параллельной программе является многопроцессорность ЭВМ.

«Программирование» в широком смысле — это тот термин, который употребляется во фразах «программа концерта», «линейное» и «математическое программирование», «программное обучение» или «программированное выращивание урожая». Он используется в случае, когда есть некоторая система (уравнение, группа учеников, семена и поле), на которую мы можем влиять и наблюдать реакцию. Пусть мы достаточно хорошо изучили эту систему и хотим теперь достичь какой-то цели. Тогда в случае достаточно стабильного «поведения» системы мы можем составить «программу» в широком смысле слова, т. е. последовательность или систему предписаний для выполнения некоторых действий, определяющую оптимальный режим взаимодействия с системой, режим, наилучшим образом способствующий достижению цели. Таким образом, «параллельное программирование» в широком смысле непосредственно не связано с ЭВМ и означает разработку некоторой параллельной системы предписаний.

На практике, когда говорят о параллельном программировании, подразумевают нечто среднее между узким и широким смыслами этого понятия. А именно имеют в виду все, что связано с параллельным решением задач на ЭВМ: вычислительные методы, параллельные алгоритмы, операционные системы, а также организацию параллельных вычислительных комплексов.

Термин «параллельное программирование» в указанном «промежуточном» толковании имеет ряд близких по значению выражений. Среди русских синонимов наиболее часто используются также «асинхронное программирование» и «теория вычислительных систем». Их оттенки выражают различные аспекты проблемы параллельной обработки. Так, первые два — это программирование в целях дальнейшей параллельной обработки, а третий — это уже сама обработка, но если говорить об этих терминах, как о названиях областей кибернетики, то они определяют приблизительно одну и ту же область.

Как у всякой науки, у параллельного программирования можно найти источники, т. е. области, из которых оно возникло и которые стимулировали ее быстрый рост. Основным источником, несомненно, являются нужды программирования для многопроцессорных ЭВМ. Составление оптимальных параллельных программ тре-

бует решения многих задач, касающихся выявления параллелизма в последовательных программах, распределения заданий по процессорам, исключения конкуренции над ресурсами и памятью и т. д. Для решения этих задач часто требовалось абстрагироваться от некоторых деталей и работать с формальными моделями программ безотносительно реализующей вычислительной системы. Это стимулировало развитие специальной области теории алгоритмов — теории параллельного программирования, где типичной задачей стала задача минимизации временно́й сложности параллельного алгоритма. Большое влияние на формирование этой области оказали также некоторые классические разделы кибернетики такие, как недетерминированные автоматы. С их помощью обычно задается управление в моделях программ.

Третьим источником стала необходимость создания средств для моделирования систем и процессов. В современной науке и технике как при синтезе, так и при анализе сложных систем создаются модели этих систем. Поскольку сложные системы имеют, как правило, параллельно функционирующие компоненты, то и модели имеют такие компоненты. Таким образом, чтобы зафиксировать модель в некотором языке, нужно воспользоваться средством описания параллельности и взаимодействия параллельных процессоров.

Более подробно источники и характер задач, относящихся к параллельному программированию, рассмотрим на примерах. Пусть в некоторый замкнутый сосуд попадает газ, неоднородный по температуре, и начинается адиабатический процесс его перемешивания. Нужно определить температуру газа после перемешивания. Это как раз тот случай, когда требуется правильно поставить задачу, так как, учитывая адиабатичность, окажется проще решить задачу в другой постановке — определить среднюю температуру газа в сосуде до перемешивания. Предполагается, что у нас есть возможность замерить температуру газа одновременно в любых точках сосуда. Пусть $a(p, t)$ — температура газа в точке p в момент t. Двум упомянутым нами постановкам задачи соответствуют два метода.

Первый метод заключается в том, чтобы дождаться, пока газ перемешается, после чего искомое значение a_0 будет равно установившейся в сосуде температуре. Вто-

рой метод — замерить значения температур в достаточно «чистой» совокупности точек (p_i, $i = 1, \ldots, n$) и взять среднее арифметическое. Таким образом, в первом случае для вычисления используется формула

$$a_0 = \lim_{i \to \infty} a_i = \lim_{t_i \to \infty} a(t_i, p_0),$$

где p_0 — произвольная точка; во втором случае — формула

$$a_0 = 1/n \sum_{i=1}^{n} x_i = 1/n \sum_{i=1}^{n} a(t_0, p_i),$$

где t_0 — исходный момент времени, в который производится замер. Итак, для решения одной и той же задачи имеются два различных пути.

Если теперь стоит задача ускорить получение решения, то очевидно, что при использовании первого метотода мы ничего не сможем сделать в этом направлении. Пока газ не станет достаточно однородным, а это выразится в близости соседних значений a_i, нельзя установить его окончательную температуру. Получение же результата вторым методом зависит не от поведения газа, а лишь от скорости измерения и выполнения арифметических операций. Предположим для определенности, что измерение не требует времени, а сложение и деление занимают по одному такту. Тогда для вычисления a_0 обычным методом потребуется n тактов: $n - 1$ — сложение и одно деление. Это как раз та ситуация, когда неудачный выбор алгоритма уничтожает потенциальную параллельность задачи. Однако, если организовать вычисление суммы специальным образом, то можно сократить время вычисления. Предположим для простоты, что $n = 2^m$. Тогда за один такт можно выполнить все сложения вида

$$x_i + x_{i+1}, \; i = 1, 3, 5, \ldots, n - 1.$$

Пусть значение суммы $x_i + x_{i+1}$ приписывается переменной x_i. За следующий такт можно получить все суммы

$$x_i + x_{i+2}, \; i = 1, 5, 9, \ldots, n - 3, \text{ и т. д.}$$

Если выполнен k-й такт одновременного сложения $x_i + x_{i+2^k-1}$ $i = 1{,}1 + 2^k, 1 + 2 \cdot 2^k, \ldots, (n-2) + 1$, то значения сумм следует приписать к переменным x_i

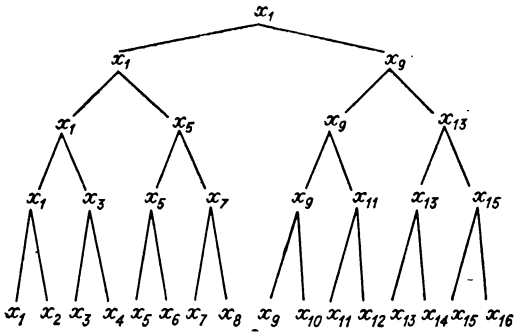


Рис. 2.1

и перейти к $k+1$-му такту. Схема организованного таким образом вычисления при $n = 2^4$ изображена на рис. 2.1. Она представляет собой полное двоичное дерево с числом ярусов $m+1 = 5$. Общее число тактов, требуемое для осуществления всех операций сложения, равно m, следовательно, общее число тактов для вычисления a при n точках будет равно $]\log_2 n + 1[$ или асимптотически — $O(\log n)$.

Итак, второй метод вычисления при специальной организации счета дает существенный выигрыш во времени. Приведенный пример является типичным для задач параллельного программирования на этапе выбора или разработки вычислительного метода.

После нахождения приемлемого метода вычисления возникает задача «фиксации» его в некотором языке программирования. Нетрудно заметить, что выбранный нами метод параллельного вычисления суммы не может быть записан, скажем, в таком популярном языке, как Фортран. Таким образом, прежде всего встает задача разработки подходящих языковых средств для записи параллельных алгоритмов. Программным аспектам параллельной обработки посвящена третья глава. Здесь же мы лишь заметим, что для записи параллельных сложений нашего примера может быть использована следующая языковая конструкция, где n уже произвольно:

$$\textbf{for } k = 1, \lceil \log n \rceil \textbf{ do}$$
$$\textbf{for } i = 1, \quad 1 + 2^k \lceil n/2^k - \frac{3}{2} \rceil, \quad 2^k \textbf{ do par} \qquad (1)$$
$$x(i) := x(i) + x(i + 2^{k-1}),$$

где $\lceil a \rceil$ — ближайшее целое, бо́льшее либо равное a. В записи (1) внешний фортрановский цикл задает последовательность переходов от яруса к ярусу дерева (см. рис. 2.1). Внутренний оператор **do par** предписывает одновременное суммирование и присваивание значений переменным $x(1)$ от 1 до $1 + 2^{k} \left\lceil \frac{n}{2} - \frac{3}{2} \right\rceil$ с шагом 2. Окончательный результат накапливается в ячейке $x(1)$.

Последующие этапы решения задачи связаны непосредственно с ЭВМ. В первую очередь требуется оттранслировать нашу программу на машинный язык с адекватным перенесением параллелизма. Это значит, что действия, которые определены как параллельные в (1), должны выполняться одновременно и на машинном уровне.

При разработке систем коммутации, осуществляющих связь между подсистемами, принципов взаимодействия процессоров и обращения к данным МВК, также возникают задачи, которые принято относить к проблематике параллельного программирования. Первая из них — создание адекватного языка моделирования для конструируемого комплекса. Такой язык должен отражать основные компоненты комплекса в структурном, функциональном и имитационном смыслах. Первое обозначает то, что должны быть развитые средства описания соединений элементов в структуру; второе — программная машина должна выдавать на входных данных тот же результат, что и описываемая реальная; третье — работа программной машины, т. е. последовательность обрабатываний элементов программы должна отражать работу реальной проектируемой машины.

Имея язык моделирования при разработке того или иного звена МВК, можно активно использовать машинный эксперимент. Проиллюстрируем это на простом примере. Одна из типичных конфигураций МВК представляет собой несколько процессоров, работающих над общей памятью. Работа каждого процессора состоит из трех чередующихся процедур: считывание из памяти, счет, запись в память. Если считывание из памяти может вестись несколькими процессорами одновременно, то запись в каждый момент времени, по техническим причинам, может осуществлять только один процессор. Таким образом, МВК должен иметь механизм взаи-

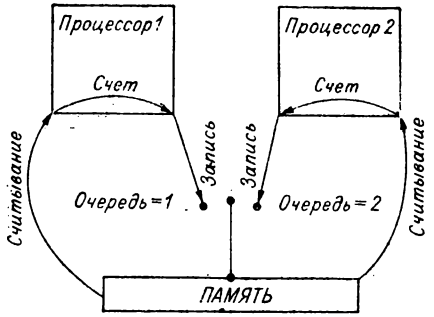


Рис. 2.2

монсключения процессоров по записи. Попробуем спроектировать такой механизм для случая двух процессоров, пользуясь некоторым простейшим языком моделирования. Первая идея, которая приходит в голову,— поставить переключатель, соединяющий память с каналом записи то одного, то другого процессора, как показано на рис. 2.2.

Программа, изображающая функционирование такой системы, выглядит следующим образом:

процесс 1:	процесс 2:
считывание;	*считывание*;
счет;	*счет*;
M1: **если** *очередь* = 2	*M2*: **если** *очередь* = 1
то на *M1*	**то на** *M2*
иначе *запись*;	**иначе** *запись*;
очередь := 2;	*очередь* := 1;
на *процесс* 1;	**на** *процесс* 2;

(2)

На i-м процессоре реализуется i-й процесс, $i = 1, 2$. Считывание и счет ведутся независимо и параллельно. Когда наступает время записи в память, проверяется значение переменной *очередь*, изображающей положение ключа: *очередь* $= i$ означает замыкание канала записи i-го процессора. Если канал замкнут, то производится запись, после чего ключ «перебрасывается на другую сторону», чтобы дать возможность произвести запись и другому процессору. Если же канал разомкнут, то процесс «крутится» на метке, ожидая своей очереди. Таким образом происходит взаимное исключение при записи.

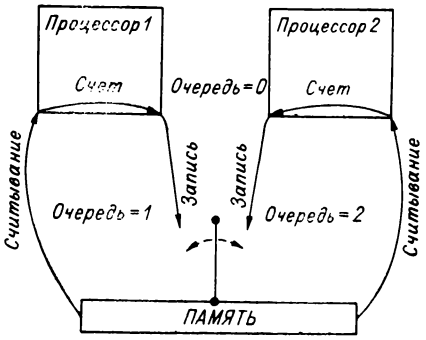


Рис. 2.3

Предположим, что наш язык моделирования реализован и программу (2) можно пропустить на некоторой машине. Для механизма взаимоисключения несущественно, как работают процедуры: считывание, счет и запись. Важно только, что они требуют время, которое, вообще говоря, непредсказуемо изменяется от одного выполнения процедуры к другому. Более существенно то, что, распечатав последовательность значений переменной *очередь*, мы увидим, что они строго чередуются: 121212... . Таким образом, запись в память процессоры производят также поочередно. В нашем случае механизм взаимоисключения и соответствующая программа очень просты, поэтому к такому выводу можно было бы прийти из обычного анализа программы, не прибегая к машинному эксперименту. Однако уже в несколько более сложных ситуациях аналогичные выводы сделать непросто.

Итак, значения переменных чередуются, а это означает, что наше решение не является вполне удовлетворительным. Пусть, например, на считывание и запись каждый процессор расходует один такт времени; на счет первый процессор тратит также один такт, а второй — десять тактов. Тогда получится, что первый процессор будет в основном простаивать. «Перебросив» ключ второму процессору, он через два такта будет готов к очередному «сеансу записи», но второй процессор предоставит ему канал только через двенадцать тактов. Чтобы исправить это положение, попробуем ввести для ключа нейтральную позицию: *очередь* = 0 (рис. 2.3),

Приведем программу, отражающую такую ситуацию:

Очередь := 0;

	процесс 1;	*процесс* 2;
	считывание;	*считывание*;
	счет;	*счет*;
M1:	**если** *очередь* = 2	*M2*: **если** *очередь* = 1
	то на *M1*;	**то на** *M2*; (3)
	очередь := 1;	*очередь* := 2;
	запись;	*запись;*
	очередь := 0	*очередь* := 0;
	на *процесс* 1;	**на** *процесс* 2;

Сейчас, перед записью, процессор проверяет положение ключа, и если он замыкает записи другого процессора, то ожидает на метке. Иначе он «притягивает» ключ к себе, производит запись и «отталкивает» ключ в нейтральное положение. Такая конструкция блока взаимного исключения исправляет недостаток, заключающийся в строгой периодичности состояний ключа, но, как будет показано ниже, обладает другими недостатками. Если бы каждый раз при реализации того или иного технического решения мы строили реальный экспериментальный блок, а не программный, то это обошлось бы нам значительно дороже.

Итак, естественным образом возникает проблема описания одновременно протекающих процессов в некотором языке программирования. Для процессов различной природы нужны различные языковые средства. В следующих разделах мы рассмотрим некоторые из них, разумеется в наиболее простом, «модельном» виде.

Средства программирования высокого уровня

Получить язык параллельного программирования двумя путями: расширить введением параллельных средств некоторый последовательный язык, например Фортран, или разработать полностью новый параллельный язык. Несмотря на то что первый путь связан с обычными для таких случаев трудностями нарушения цельной языковой концепции, внедрением другой, в некотором смысле противоположной концепции, он выглядит предпочтительнее, так как сохраняется весь программный продукт, написанный в последовательном языке;

пользователю не нужно полностью переучиваться, а можно только постепенно «доучиваться» новым операторам. Поэтому далее мы будем вести речь о параллельных средствах, как о средствах, расширяющих некоторый последовательный язык.

Какие же языковые конструкции необходимо ввести в языки программирования в целях задания эффективных параллельных вычислений? В этой главе мы остановимся практически на всех видах таких средств, но только в наиболее простых их вариантах.

1. **Средства для параллельного ветвления.** В зависимости от характера ветвящихся участков они делятся на средства массового и «единичного» ветвления. Другой параметр классификации — семантика ветвления — определяет поведение этих ветвей по отношению друг к другу. Запущенные параллельно ветви могут взаимодействовать: задерживать друг друга, обмениваться информацией и т. д. Среди них выделяются средства высокого уровня, позволяющие сформулировать взаимодействия в близких к пользователю терминах: имен переменных, операторов, ветвей, подпрограмм.

Нами был дан подробный пример взаимодействия двух процессоров над общей памятью. В более широкой постановке можно рассмотреть дисциплину взаимодействия произвольного числа исполнительных устройств над произвольным набором ресурсов. В разд. 2.3 вводятся языковые средства низкого уровня, позволяющие описать эти взаимодействия.

До сих пор во всех рассуждениях о языках программирования речь шла только о процедурных средствах. Такие средства задают явное управление в программе: предписывают, какой оператор выполняется следующим, когда нужно «разветвить» вычисление, когда обменяться информацией. Сейчас все большую популярность приобретают так называемые непроцедурные языковые конструкции. Такого рода конструкции не определяют явно порядок вычисления. Они задают в какой-то форме либо целый спектр порядков, либо описывают лишь конечный результат. Это позволяет реализовать такие конструкции параллельно. Наибольшее развитие непроцедурные операторы получили в языке АПЛ. Некоторые из элементов этого языка описаны в разд. «Взаимодействия процессов», тогда как в разд. «Асинхронные средства программирования» описаны

непроцедурные средства другого типа — в них готовность к выполнению определяется истинностью некоторых условий.

2. **Средства задания массового ветвления.** В разд. «Как устроены современные ЭВМ» были рассмотрены примеры, требующие для параллельной обработки объявления целого массива параллельных ветвей. Массивы вычислений в последовательных языках задаются циклами. Поэтому оператор задания массива параллельных вычислений часто называют параллельным циклом. Синтаксически он описывается разработчиками языка обычно похожим на последовательный цикл. Например, если в последовательном языке имеется циклическая конструкция

$$\textbf{for } i = l,\ r,\ h \textbf{ do } T(i),$$

задающая последовательное выполнение итераций

$$T(l),\ T(l+h),\ T(l+2h),\ \ldots$$

(тело $T(i)$ исполняется при различных подстановках параметра i от l до r с шагом h, $h=1$ в записи может быть опущено), то при параллельном расширении естественно ввести конструкцию

$$\textbf{for } i = l,\ r,\ h \textbf{ do par } T(i),$$

задающую одновременное выполнение этих же итераций.

В данном случае имеем *цикл типа арифметической прогрессии*. Вообще говоря, индексное множество может задаваться многими другими способами. Понятие «одновременное выполнение» может уточняться также по-разному. Поэтому для общего случая примем следующее определение оператора параллельного цикла: **for** $i =$ ⟨индексное множество⟩ **do** ⟨тип параллельности⟩ $T(i)$. Здесь ⟨индексное множество⟩ есть либо

а) тройка l, r, h, что соответствует циклу типа арифметической прогрессии, либо

б) вектор $(i_1, \ldots, i_n)$, что соответствует циклу типа *перечисления*: задается выполнение группы итераций, либо

в) логическое выражение (предикат $P(i)$ в смысле расширяемого языка, зависящее от i; в этом случае предписывается одновременное выполнение группы

итераций

$$\{T(i): | i \in \{1, 2, 3, \ldots\} \wedge P(i) = 1\}.$$

Этот цикл напоминает последовательный цикл типа **пока.** Дополнительно от предиката требуется, чтобы множество было конечным и чтобы эффективно определялась его мощность, либо левая и правая границы, между которыми оно лежит.

Конструкция ⟨тип параллельности⟩ в записи цикла есть одно из следующих служебных слов: CONC, AUTON, SIM, SINCH(J), EXCL. Каждое из этих слов задает определенный тип поведения параллельных ветвей или, другими словами, тип семантики оператора параллельного цикла. Рассмотрим их более подробно.

Тип параллельности CONC (concurrent) определяет независимое выполнение всех ветвей при взаимодействии через общую память. Если нет вмешательства средств синхронизации, то каждая ветвь выполняется так, как будто она работает одна. Если какая-то ветвь в момент t делает присваивание переменной x, то в любой момент t_1 такой, что между t и t_1 нет засылок в/из других ветвей, любая другая ветвь, считывая x, считывает именно это значение. Так они обмениваются информацией. Тип CONC ориентирован на асинхронные МВК с общей памятью.

Тип параллельности AUTON (autonomus) отличается от CONC тем, что все переменные локализованы в своих ветвях. Выполнение ветвей происходит независимо, но если одна ветвь вырабатывает x, а другая считывает x, то эти x разные. Считывается последнее значение x, присвоенное этой переменной в той же самой ветви. Для обмена информацией между ветвями необходимо использовать специальные операторы обмена. Такая семантика ориентирована на асинхронные вычисления над разделенной памятью. Семантика типа SIM (simultaneous) определяется тогда, когда имеется некоторый транслятор, преобразующий последовательность операторов $T(i)$ в последовательность машинных команд. При этом требуется, чтобы независимо от i эта последовательность была одна и та же. Последнее требование автоматически обеспечивается, если в последовательности нет операторов передачи управления.

При выполнении перечисленных условий цикл реализуется следующим образом. Вначале во всех ветвях

одновременно считывается информация для первых команд этих ветвей. Затем во всех ветвях одновременно выполняется эта команда. Потом одновременно рассылаются вычисленные данные, при этом должно быть обеспечено, чтобы две ветви не производили засылки в одну и ту же переменную. Далее в той же последовательности исполняется вторая команда ветвей и т. д. Таким образом, эта семантика существенно ориентирована на процессоры с единым потоком команд ОКМД.

Может оказаться, что тело параллельного цикла $T(i)$ также является циклом, но последовательным. Таким образом, в каждой параллельной ветви исполняется некоторая последовательность итераций. По некоторым причинам может потребоваться синхронизация выполнения ветвей через один или несколько кортежей итераций. Часто такая ситуация возникает при распараллеливании последовательных циклов. Для целей синхронизации такого рода применяется тип SYNCII(J). Рассмотрим, например, цикл

$$\textbf{for } i = 1{,}16 \textbf{ do } x(i) := x(6i - 27).$$

Все итерации этого цикла нельзя выполнить параллельно, так как между ними есть зависимость. Например, 5-я итерация зависит от 3-й. Цикл можно выполнить «кортежами», например, по четыре итерации в каждом:

1-й кортеж	2-й кортеж	3-й кортеж	4-й кортеж
$x(1) := x(-221);$	$x(5) := x(3);$	$x(9) := x(27);$	$x(13) := x(51)$
$x(2) := x(-15);$	$x(6) := x(9);$	$x(10) := x(33);$	$x(14) := x(57)$
$x(3) := x(-9)$	$x(7) := x(15);$	$x(11) := x(39);$	$x(15) := x(63)$
$x(4) := x(-3);$	$x(8) := x(21)$	$x(12) := x(45);$	$x(16) := x(69)$

Такой порядок выполнения можно записать в следующем виде:

$$\begin{aligned} &\textbf{for } k = 1{,}4 \textbf{ do SYNCII} \\ &\quad \textbf{for } i = k,\ 16{,}4 \textbf{ do} \\ &\quad x(i) := x(6i - 27) \end{aligned} \qquad (1)$$

Если же у нас, по существу, стековая машина, то для нее удобнее такая форма записи:

$$\begin{aligned} &\textbf{for } i = 1{,}4 \textbf{ do} \\ &\textbf{for } j = 4(i - 1) + 1,\ 4i \textbf{ do CONC} \\ &x(j) := x(6j - 27) \end{aligned}$$

Здесь запись **do SYNCH** обозначает, что делается синхронизация после выполнения в каждой ветви очередной итерации. Таким образом, ветви ожидают, пока не выполняются все итерации 1-го кортежа, затем запускаются итерации 2-го кортежа. После их выполнения запускаются итерации 3-го кортежа и т. д. Между выполнениями кортежей происходит обмен информацией, в нашем случае между 3-й и 5-й итерациями. Если у нас всего два процессора, то следует завести только две ветви. Тогда для сохранения информационных связей достаточно производить синхронизацию через два кортежа, что выражается записью

$$\begin{aligned} &\textbf{for } k = 1, 2 \textbf{ do SYNCH} \\ &\textbf{for } i = k, 16, 2 \textbf{ do} \\ &x(i) := x(6i - 27) \end{aligned} \qquad (2)$$

Наконец, последний тип параллелизма EXCL (exclusively) не является в буквальном смысле типом параллелизма, так как определяет последовательное выполнение всех итераций, но в произвольном порядке. Он может быть использован в случае, когда ресурсы не позволяют выполнять одновременно две или более итерации, и в то же время итерации могут оказаться готовыми к выполнению (в определенном смысле) в произвольном порядке. Например, в некоторой задаче реального времени в произвольном порядке для этих итераций могут прийти входные данные, определяющие их готовность.

Перечисленные типы параллелизма могут использоваться и в различных непротиворечивых сочетаниях. Например, AUTON SIM обозначает синхронное выполнение без обмена информацией.

Есть и другие возможности задания массового ветвления. Так, в языке IVTRAN, созданном для процессора ILLIAC-IV, используется оператор **do** M **for ALL** $(i_1, \ldots, i_n)/S$, задающий повторение последующего участка программ до метки M для всех наборов $(i_1, \ldots, i_n)$, удовлетворяющих n-местному предикату S. Наша форма цикла также довольно просто обобщается при допущении, что индексное множество состоит из векторов индексов.

В качестве средств «единичного» ветвления в языках используются различные виды параллельных «ско-

бок» из служебных слов: **parbegin**, **parend**; **cobegin**, **coend**; **start**, **halt**; ... В этих конструкциях первый примитив открывает участок параллельности, а второй — закрывает. При этом обычно налагается требование того, что параллельный участок закрывается только тогда, когда все его ветви завершились. Каждая из ветвей может содержать свои участки параллельности, таким образом они образуют произвольную иерархию.

Рассмотрим более подробно вариант примитивов для задания ветвления, определяемый служебными словами **fork**, **join** — разветвить, соединить. Представим эту пару в следующем синтаксисе:

fork ⟨тип параллельности⟩ ⟨ветвь⟩, ⟨ветвь⟩, ...

..., ⟨ветвь⟩ **join**.

Тип параллельности здесь имеет тот же смысл, что и в операторе параллельного цикла. Однако следует отметить, что такие типы, как SIM и SYNCH(y), на практике, видимо, использоваться не будут, так как они употребимы лишь в случае идентичных ветвей, а идентичные ветви удобнее задавать оператором цикла. Правда, есть возможность придать этим примитивам более свободный смысл. Так, если каждая ветвь состоит из единственного присваивания, то тип SIM может обозначать одновременное, но не обязательно синхронное вычисление правой части и одновременное присваивание, как в языке Tranquil. Запись ⟨ветвь⟩,... ..., ⟨ветвь⟩ обозначает, что далее идет произвольная последовательность ветвей (не менее одной ветви). А ⟨ветвь⟩ есть либо метка ветви, либо последовательность операторов. Ветви, описываемые метками, имеют вид

N :⟨оператор⟩; ⟨оператор⟩; ...; ⟨оператор⟩ end N

при этом все метки перечислены в описании атрибутов программы

branch M_1, M_2, ..., MN.

Такие ветви можно использовать повторно, обращаясь к ним по именам.

Ветвь, которая является последовательностью, заключенной между **fork** и **join**, описывается «своим появлением» и должна быть повторяема при всех после-

дующих к ней обращениях. Приведем простой пример использования этого аппарата ветвления. В последовательной программе

integer x, y, u, v, w
ввод (x, y);
$u := x + y$
$v := x * y$;
$w := x/y$
вывод (u, v, w)

все присваивания могут быть выполнены одновременно. Перепишем ее в следующем виде:

integer x, y, u, v, w
branch M
$M : v := x * y$
end M
ввод (x, y);
fork CONC
$u = x + y$,
M,
$v := x/y$
join
вывод (u, v, w).

Здесь одна ветвь объявляется описанием, и обращение к ней производится по метке, а оставшиеся две выписываются явно между **fork** и **join**.

Если нехватка ресурсов вынуждает нас делать взаимоисключения некоторых ветвей, то помимо примитива EXCL как типа параллельности можно использовать **excl** как описатель взаимозаключающих ветвей. Появление **excl** $(M11, \ldots, M_1N_1), \ldots, (MK_1, \ldots, MKN_k)$

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
1	17	2	18	3	19	4	20	5	21	6	22	7	23	8	24
1	9	17	25	2	10	18	26	3	11	19	27	4	12	20	28
1	5	9	13	17	21	25	29	2	6	10	14	18	22	26	30
1	3	5	7	9	11	13	15	17	19	21	23	25	27	29	31
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

в сегменте описаний программы гарантирует, что ни одна группа ветвей (M_i1, ..., M_iN_i), $i = 1$, ..., K, не совместится при выполнении. Это средство особенно полезно при исключении ветвей на разных уровнях иерархии.

Рассмотрим еще один несложный пример применения языковых средств для параллельной обработки. Он касается задачи моделирования так называемой процедуры полной тасовки. Проще всего эта процедура иллюстрируется с помощью колоды карт. Пусть у нас имеется обычная преферансная колода карт, состоящая, как известно, из 32 карт. Однократное действие тасовки умелый игрок совершает в следующем порядке. Колода делится ровно на две части, и вторая из частей вкладывается в первую таким образом, что между каждой парой соседних карт попадает некоторая разделяющая их единственная карта. После этого за первой картой последует карта с номером 17, за второй — с номером 18 и т. д. Рассмотрим табл. 2.1. Первая ее строка показывает исходный порядок карт в колоде, вторая строка — порядок карт после однократной тасовки, третья — после двукратной и т. д. (Заметим, кстати, что после пятого действия карты в колоде устанавливаются в первоначальном порядке. Этим свойством пользуются иногда карточные шулеры.)

Описанная процедура, хотя и иллюстрируется таким полулегальным способом, имеет большое практическое значение и может быть использована во многих областях. Например, в генной инженерии ее механизм может быть использован при получении новых молекулярных цепочек. В той же вычислительной технике

Таблица 2.1

17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32
9	25	10	26	11	27	12	28	13	29	14	30	15	31	16	32
5	13	21	29	6	14	22	30	7	15	23	31	8	16	24	32
3	7	11	15	19	23	27	31	4	8	12	16	20	24	28	32
2	4	6	8	10	12	14	16	18	20	22	24	26	28	30	32
17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32

по типу полной тасовки осуществляется коммутация в некоторых вычислительных комплексах. Все это определяет актуальность изучения данного механизма и, в частности, важность его моделирования. Поскольку скорость моделирования увеличивается при распараллеливании, актуален вопрос о параллельной организации процесса моделирования.

Самый простой способ параллельного моделирования — использование параллелизма типа SIM. Рассмотрим следующую программу, изображенную для наглядности в несколько отличном от обычного виде:

fork SIM

for $i = 1,31,2$ **do** SIM
$x(i) := x([i/2 + 1])$

for $i = 2,32,2$ **do** SIM
$x(i) := x(i/2 + 16)$

join SIM

Она задает синхронное выполнение двух ветвей, каждая из которых представляет собой цикл из 16 итераций, имеющий тип также SIM. Переменная $x(i)$ служит для запоминания номера карты в i-й позиции. Вся работа, осуществляемая данной программной конструкцией, выражается в следующей серии одновременных присваиваний:

$x(1) := x(1)$	$x(2) := x(17)$
$x(3) := x(2)$	$x(4) := x(18)$
$x(5) := x(3)$	$x(6) := x(19)$
$x(7) := x(4)$	$x(8) := x(20)$
$x(9) := x(5)$	$x(10) := x(21)$
$x(11) := x(6)$	$x(12) := x(22)$
$x(13) := x(7)$	$x(14) := x(23)$
$x(15) := x(8)$	$x(16) := x(24)$
$x(17) := x(9)$	$x(18) := x(25)$
$x(19) := x(10)$	$x(20) := x(26)$
$x(21) := x(11)$	$x(22) := x(27)$
$x(23) := x(12)$	$x(24) := x(28)$
$x(25) := x(13)$	$x(26) := x(29)$
$x(27) := x(14)$	$x(28) := x(30)$
$x(29) := x(15)$	$x(30) := x(31)$
$x(31) := x(16)$	$x(32) := x(32)$

Заметим, что возможно только полностью одновременное выполнение всех итераций. Отдельное выполнение

любой из них приводит к потере одного из значений.

Естественно поставить задачу: можно ли осуществить все указанные перестановки с помощью языковых конструкций типа CONC, если есть возможность использовать дополнительные ячейки памяти? Самое простое решение заключается в том, что вводится новый массив $y(i)$, $i = 1, \ldots, 32$, в котором запоминается порядок карт в перетасованной колоде, а затем его значения присваиваются тем же элементам массива x. Реализующая этот режим тасовки программа выглядит так:

$$\begin{array}{l}\textbf{for } i = 0,\ 62,\ 2 \textbf{ do CONC}\\ y((i-1)\ (\mathrm{mod}\ 31)+2) := x(i/2+1);\\ \textbf{for } i = 1,\ 32 \textbf{ do CONC}\\ x(i) := y(i)\end{array} \qquad (3)$$

(здесь $n(\mathrm{mod}\ m)$ обозначает число n, взятое по модулю m). Эта программа осуществляет полную тасовку за два такта.

Кстати, имея пять дополнительных массивов по 32 элемента, можно в программе одной языковой конструкцией описать весь процесс тасовки, представленной табл. 2.1. Если предположить, что для всех значений i

$$y(0,\ i) = x(i),$$

то эта конструкция имеет следующий вид:

$$\begin{array}{l}\textbf{for } j = 1,5 \textbf{ do}\\ \textbf{for } k = 0,\ 62,\ 2 \textbf{ do CONC}\\ y(j,\ (k-1)\ (\mathrm{mod}\ 31)+2) := y(j-1,\ k/2+1).\end{array}$$

Дело обстоит хуже, если дополнительных ячеек меньше, чем 32. Правда, сыграв на том, что в первой и последней ячейках значения не изменяются, можно обойтись тридцатью дополнительными ячейками. Но если их меньше, чем 30, то проблема не так проста. Дело в том, что если переписать в массив y все, кроме одного значения x, например, значения $x(16)$, то при обратной переписи в массив x это значение уничтожится, в данном случае на месте $x(16)$ запишется $x(24)$.

Опишем более тонкий подход к решению поставленной задачи. Проанализируем пути, по которым в схеме тасовки одно значение переходит на место другого. Так, значение $x(2)$ попадает на место $x(3)$, значение

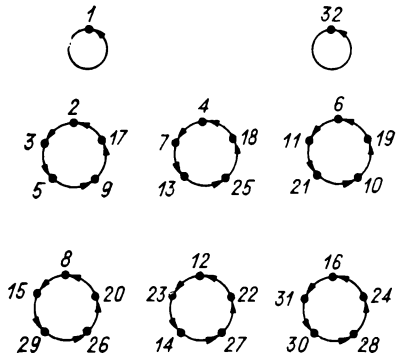


Рис. 2.4

$x(3)$ идет в ячейку $x(5)$, $x(5)$ занимает место в $x(9)$, которое, в свою очередь, перемещается на место $x(17)$, а $x(17)$ попадает на место $x(2)$. Если изобразить пути перемещения переменных графически, то получится шесть циклов длины пять и две изолированные петли для $x(1)$ и $x(32)$ (рис. 2.4).

Основная идея параллельного метода тасовки заключается в том, что значения переменных в каждом из циклов меняются последовательно, но все циклы осуществляют свои изменения одновременно. Однако если начать вычисление с присваивания новых значений в ячейки массива x, то значение одной из ячеек x неминуемо потеряется. Поэтому для каждого цикла вводится дополнительная ячейка $u(i)$, $i = 1, \ldots, 6$, в которую «прячется» одно из значений x. Целиком соответствующая языковая конструкция выглядит так:

fork CONC

$u(1) := x(2);$	$u(2) := x(4);$
$x(2) := x(17);$	$x(4) := x(18);$
$x(17) := x(9);$	$x(18) := x(25);$
$x(9) := x(5);$	$x(25) := x(13);$
$x(5) := x(3);$	$x(13) := x(7);$
$x(3) := u(1),$	$x(7) := u(2),$
$u(3) := x(6);$	$u(4) := x(8);$
$x(6) := x(19);$	$x(8) := x(20);$
$x(19) := x(10);$	$x(20) := x(26);$
$x(10) := x(21);$	$x(26) := x(29);$
$x(21) := x(11);$	$x(29) := x(15);$
$x(11) := u(3),$	$x(15) := u(4),$

$u(5) := x(12);$ $u(6) := x(16);$

$x(12) := x(22);$ $u(16) := x(24);$

$x(22) := x(27);$ $x(24) := x(28);$

$x(27) := x(14);$ $x(28) := x(30);$

$x(14) := x(23);$ $x(30) := x(31);$

$x(23) := u(5);$ $x(31) := u(6);$

join CONC

Переприсваивания переменным $x(1)$ и $x(32)$ мы опустили. В этой записи заголовок **fork** CONC относится ко всем шести участкам присваивания. Общее время выполнения конструкции — 6 тактов. Предлагаем читателю самостоятельно разработать соответствующие языковые конструкции в случае, когда дополнительных переменных меньше шести.

Семафоры

В предыдущем разделе мы рассмотрели средства синхронизации параллельных ветвей, определяемые типом семантики параллелизма. Они позволяют организовать взаимодействие ветвей, но взаимодействие это регулярное, описываемое в заголовке и фиксирующее определенный режим синхронизации и обменов, либо его отсутствие. Для других менее регулярных или вообще не регулярных единичных взаимодействий используют другие средства. В разд. 3.3, 3.4 мы рассмотрим некоторые из них.

Изучим средства синхронизации низкого уровня. Обычно они служат для обеспечения взаимного исключения процессов при делении общих ресурсов. У нас уже был пример такой задачи, когда два процессора конкурировали при обращении к общей памяти (см. разд. 2.1). Был также пример языковой конструкции высокого уровня — примитив EXCL, позволяющий исключить параллельное протекание процессов. Но EXCL должен быть как-то реализован в машинных понятиях на уровне ячеек памяти. А как показало рассмотрение примера конкуренции над общей памятью — это не такая уж простая задача. Наше второе решение задачи (см. рис. 2.3) описывается программой (3). Эта программа, используя промежуточное состояние ключа

очередь-0, уже не задает периодичной работы процессоров: 121 212... Однако в ней есть другой недостаток. Предположим, *очередь* находится в состоянии 0, а оба процессора одновременно провели считывание, счет и проверили состояние переменной *очередь*. После этого они одновременно переходят на операторы присваивания и пытаются записать в переменную *очередь сразу* 1 и 2. А это, естественно, сделать нельзя ни в программе, ни на физической модели. В программе произойдет попытка записи в ячейку сразу двух значений, а на модели — одновременная попытка «притянуть» ключ сразу двумя процессорами.

Можно усовершенствовать программу, вводя два ключа, имитируемые двумя переменными, например *c1* и *c2*:

```
                    c1 := c2 := 0;
процесс 1;                          процесс 2;
считывание;                         считывание;
счет;                               счет;
M1: если c2 = 1                 M2: если c1 = 1
    то на M1;                       то на M2;
    c1 := 1                         c2 := 1
    запись;                         запись;
    c1 := 0                         c2 := 0;
    на процесс 1;                   на процесс 2;
```

В этой программе для каждого канала записи существует свой ключ *c1*, *c2*, принимающий значение 1 — замкнут, 0 — разомкнут. В случае нейтрального положения ключей сейчас не будет попытки записи в одну и ту же переменную: единица запишется в разные ячейки *c1*, *c2*. Но зато после этого оба канала окажутся замкнутыми и начнется одновременная запись в память, исключения которой мы добиваемся.

Попробуем исключить такую ситуацию, поменяв местами проверку замкнутости соседнего канала и замыкания собственного канала. В программе это выразится перестановкой 6-й строки на 4-ю позицию:

```
                    c1 := c2 := 0
процесс 1;                          процесс 2;
считывание;                         считывание;
счет;                               счет;
c1 := 1                             c2 := 1
```

```
M1: если c2 := 1          M2: если c1 := 1
    то на M1;                 то на M2;
    запись;                   запись;
    c1 := 0;                  c2 := 0;
    на процесс 1;             на процесс 2;
```

Сейчас не может быть одновременной записи, так как процесс перед записью проверял канал соседа, и если он замкнут, то «крутится» на метке. Но зато если оба процесса замкнули свои каналы одновременно, они будут ждать друг друга и ни один из них не начнет запись. Такая ситуация называется *взаимная блокировка*. Для выхода из нее, по-видимому, время от времени надо размыкать каналы, даже если и не производилась запись. Такое решение дает следующая программа:

```
                    c1 := c2 := 0
    процесс 1;                    процесс 2;
    считывание;                   считывание;
    счет;                         счет;
M1: c1 := 1;                  M2: c2 := 1;
    если c2 = 1                   если c1 = 1
    то {c1 := 0; на M1}           то {c2 := 0; на M2}
    иначе запись;                 иначе запись;
    c1 := 0                       c2 := 0;
    на процесс 1;                 на процесс 2;
```

(в фигурные скобки заключен основной оператор). В ней каждый процессор, замкнув свой канал, «оглядывается» на соседа, и если у него также замкнут канал, то размыкает свой канал и дает соседу некоторый интервал времени, чтобы произвести запись. Ситуация напоминает попытку двух абонентов связаться друг с другом по телефону. Если они одновременно поднимают трубку (замыкают канал) и слышат короткие гудки (канал соседа замкнут), то произвольным образом кладут трубку на рычаг (размыкают канал) и через какое-то время поднимают ее снова. Понятно, что после нескольких попыток связь все-таки произойдет. Таким образом, приведенная программа, а значит, и соответствующее устройство решают поставленную задачу, но только не для синхронных вычислений. Если на обоих процессорах считаются полностью идентичные программы, а процессоры имеют общее устройство управления, то замыкания и размыкания каналов

будут выполняться также синхронно. Снова возникнет ситуация взаимной блокировки.

Правильное решение этой задачи для произвольных типов вычислителей показано ниже. Оно весьма громоздко, использует два типа ключей и множество проверок, что снижает эффективность использования памяти и процессорного времени. По этим причинам его следует признать не вполне удовлетворительным.

```
                    c1 := c2 := 0;
                    очередь := 1
   процесс 1;                          процесс 2;
   считывание;                         считывание;
   счет;                               счет;
a1: c1 := 1;                        a2: c2 := 1;
M1: если c2 = 1                     M2: если c1 = 1
    то                                  то
    {если очередь = 1                   {если очередь = 2
    то на M1                            то на M2
    иначе c1 := 0                       иначе c2 := 0;
b1: если очередь = 2                b2: если очередь = 1
    то на b1                            то на b2
    иначе на a1}                        иначе на a2}
иначе запись;                       иначе запись;
очередь := 2;                       очередь = 1;
c1 := 0;                            c2 := 0;
на процесс 1;                       на процесс 2;
```

Простое правильное решение задачи взаимного исключения предложено Дейкстрой. Оно использует новый тип переменных и, следовательно, новый тип устройства для их запоминания и чтения. Эти переменные принимают целые значения и называются *семафорами* из-за сходства их роли в программе с ролью железнодорожных семафоров в управлении движением. Над семафорами определены две операции V и P:

$V(c)$ есть $c := c + 1$;
$P(c)$ есть m: **если** $c \neq 0$, **то** $c := c - 1$, **иначе на** m **все.**

Операция V — это обычное прибавление единицы. Операция P — это вычитание единицы, если число не становится отрицательным, и задержка, пока c не станет больше нуля,— в противном случае. Однако относительно этих операций существует одно очень важное соглашение: операции V и P объявляются неделимыми.

Скажем, при обычном сложении оператор $c := c + 1$ может рассматриваться как последовательность действий — сначала считывание значения переменной c из памяти, затем прибавление 1 к этому значению и, наконец, присваивание полученного результата переменной c. При выполнении же V-операции компоненты этой операции неразделимы, она неразложима на более простые действия.

При использовании семафоров программа задачи взаимного исключения записывается совсем просто:

очередь := 1;

процесс 1: *считывание*;	*процесс 2*: *считывание*;
счет;	*счет*;
P(*очередь*);	*P*(*очередь*);
запись;	*запись*;
V(*очередь*);	*V*(*очередь*);
на *процесс 1*;	**на** *процесс 2*;

В этой программе каждый из процессов, прежде чем начать запись, «закрывает» семафор *очередь*, и тогда другой процесс не может начать запись, пока семафор не откроется. А «откроется» он после окончания записи тем процессором, которым семафор был закрыт.

Фрагменты программ, в которых может находится не более одного процесса, называется *критическими интервалами*. В нашем примере критическим интервалом является выполнение процедуры запись. В нее не могут войти два процессора одновременно.

Семафоры, появившиеся в программировании в связи с необходимостью управления параллельными процессами, широко применяются и при программировании других моделирующих задач. Например, они могут моделировать автоматическое открытие и закрытие своих собратьев — железнодорожных семафоров. На больших железнодорожных станциях управление движением поездов, распределение локомотивов и вагонов по запасным путям, оптимальная комплексация составов являются очень трудными задачами. С этими задачами ЭВМ справляется сейчас лучше человека. Железнодорожные участки моделируются в ЭВМ критическими интервалами, железнодорожные семафоры — программными семафорами, а перемещения составов — различными процессами. Поскольку семафоры гарантируют, что в

один и тот же критический интервал не могут войти два процесса, то оказывается невозможным и столкновение поездов — на один и те же рельсы не могут заехать одновременно два поезда.

В терминах семафоров было найдено решение многих задач, возникающих при организации взаимодействия параллельных процессов при ограничениях на ресурсы. Однако общепризнанным недостатком семафоров является их «ненадежность»: бесконтрольное использование семафоров запутывает структуру управления параллельной программы, велика вероятность ошибки при описании сложных взаимодействий. В последние годы большее внимание уделяется технологии программирования. В частности, Дейкстрой была развита концепция структурированного программирования, в основе которой лежит дисциплина использования строго ограниченного набора языковых средств в строго регламентированных сочетаниях. В то же время его семафоры не обеспечивают структурированности параллельных программ. В связи с этим после появления семафоров сделаны попытки повысить уровень синхронизирующих примитивов и добиться структурированности параллельных программ. В большинстве случаев предлагаемые механизмы заключались в «привязке» средств синхронизации к другим программным объектам (семафор в этом смысле независим). Это позволило применять неявные, скрытые механизмы синхронизации, которые реализуются или могут реализоваться через семафоры. Объектом привязки в механизме *условных* критических интервалов являются те общие ресурсы, через которые необходима организация взаимодействия. Они представлены переменными, которые явно описываются как *общие* переменные.

Процесс может использовать общие переменные только внутри структурированного оператора, называемого условным критическим интервалом:

$$\textbf{region } v \textbf{ when } B \textbf{ do } S,$$

где B — условное выражение; S — оператор; v — переменная. Процесс входит в критический интервал и вычисляет значение B. Если оно — истина, то процесс продолжает вычисление оператора S. В противном случае процесс уходит из критического интервала и задерживается до тех пор, пока другой процесс не завер-

шит работу в критическом интервале. В этом случае задержанный процесс может вновь войти в критический интервал и вычислить B. Этот цикл продолжается, пока B не станет истинным, после чего S будет выполнен.

Однако использование общих переменных не всегда дает удовлетворительное решение. В ряде задач может возникнуть ситуация, когда некоторый процесс будет «отталкиваться» другими процессами и никогда не сможет войти в критический интервал. Для исключения таких коллизий Дейкстра предложил следующее: вместо того чтобы устраивать децентрализованное взаимодействие n процессов со своими критическими интервалами, нужно выделить эти интервалы в $n+1$-й процесс, который был назван *секретарем* и который взял на себя всю организацию взаимодействия процессов. Такая структура процессов может быть обобщена на несколько уровней иерархии. Подобные концепции централизации были реализованы также другими авторами. В целом техника взаимозаключения на машинном уровне является вполне изученной областью параллельного программирования и нашла свое воплощение в архитектуре практически всех МВК, рассмотренных в гл. 2.

Взаимодействие процессов

Перейдем теперь к рассмотрению средств взаимодействия высокого уровня. В разных языках вводились средства для синхронизации и так или иначе все они связывались с ожиданием некоторого события. Мы рассмотрим единственный оператор **wait**, с помощью которого, используя различные виды событий, можно моделировать большинство из языковых синхропримитивов. Синтаксис оператора **wait** очень прост: **wait** (⟨событие⟩). Синтаксическую единицу ⟨событие⟩ определим довольно широко. Предполагается, что в расширяемом или конструируемом нами языке есть понятие метки объекта, по меньшей мере метки ветви. Будем также предполагать, что могут быть помечены операторы, в том числе составные, и подпрограммы. Кроме того, будем предполагать, что в языке есть логические выражения вида $x = y$, $x \geqslant y$, основанные на элементарных логических выражениях, как минимум, и полу-

ченные с помощью применения обычных логических выражений связок. Элементарным событием назовем метку и элементарное логическое выражение. Событие есть либо элементарное событие, либо получено из других событий с помощью логических операций $\wedge$, $\vee$, $\supset$, $\neg$ (а для некоторых языков и кванторов $\forall$, $\exists$). Таким образом, событие есть расширение понятия логического выражения включением в качестве элементарных подвыражений меток объектов. В этом определении можно формулировать события в терминах взаимоотношения величин, выполнения или невыполнения некоторых частей программ, конъюнкций и дизъюнкций этих условий.

Семантика оператора **wait** такова. Если в некоторой ветви управление передается на оператор **wait**, то вычисляется значение события. При этом элементарное событие-метка становится истинным, если оператор, помеченный ею, завершил работу хотя бы один раз. (В конкретных языках можно вводить дополнительные предположения относительно помеченных операторов, которые вычисляются несколько раз. Например, ввести элементарное событие $M * i$, обозначающее выполнение оператора i раз.) Если событие ложно, то вычисление задерживается на операторе **wait**, и через некоторый промежуток времени вычисление условия повторяется. Чтобы избавиться от этого перевычисления, в процессоре может быть предусмотрен механизм, который ставит в известность оператор **wait** о том, что событие наступило. Как только событие оказывается истинным, вычисление переходит на следующий за **wait** оператор ветви.

Проиллюстрируем работу оператора на примере. Пусть дана программа

$$\begin{array}{l} \textit{ввод}\ (x,\ y,\ u) \\ u := x + y; \\ v := x * y; \\ w := u * x; \\ z := u/y \\ \textit{вывод}\ (u,\ v,\ w,\ z) \end{array} \qquad (4)$$

Ее можно выполнить двумя ветвями: 1-я ветвь: $u := x + y$; $w := u * x$; 2-я ветвь: $v := x * y$; $z := u/y$. Но второй оператор 2-й ветви зависит от u, которое вычисляется в 1-й ветви. Если вычисление ведется над общей памятью, то для сохранения порядка передачи

информации, определенного программистом, необходимо ввести метку и вставить оператор ожидания:

```
        ввод (x, y, u);
        fork
M:      u := x + y;                    (5)'
        w := u * x,
        v := x * y;
        wait (M);
        z := u/y;
        join
        вывод (u, v, w, z)
```

Приведенное определение оператора **wait** может обеспечить правильную синхронизацию в других, достаточно сложных случаях. Пусть в программе (5)' присваивание $u := x + y$ зависит от условия, т. е. программа имеет вид

```
        ввод (x, y, u);
        если x ≠ y, то на L
        u := x + y;
        v := x * y;
L:      w := u * x;
        z := u/y;
        вывод (u, v, w, z)
```

Ее можно реализовать параллельно:

```
        ввод (x, y, u)
        fork
        если x ≠ y, то на L1
M:      u := x + y;
L1:     w := u * x,
        если x ≠ y, то на L2;
        v := x * y;
L2      wait (M ⋀ x ≠ y);
        z := u/y;
        join
        вывод (u, v, w, z)
```

Здесь оператор **wait** задает ожидание одного из событий: либо $x \neq y$, и вычисление минует оператор M, тогда можно взять значение переменной u с вводного оператора; либо выполняется M, и значение берется с него.

Все наши примеры годились для процессов над общей памятью. Если вычисление реализовано над разделенной памятью, то для передачи данных необходимо воспользоваться операторами обмена.

Наиболее полно проблема обмена проработана в таких языках как Ада, и семействе языков для ОВС. Приведем в других обозначениях и в упрощенном виде основные языковые конструкции. Их стержнем является пара примитивов **передать, принять:**

передать $(i_1 : x_1;\ i_2 : x_2;\ \ldots;\ i_m : x_m)$,
принять $(j_1 : y_1/z_1;\ j_2 : y_2/z_2;\ \ldots;\ j_n : y_n/z_n)$,

здесь i_k, j_l — метки ветвей или их номера, исчисляемые в естественном порядке, а x_k, y_l — имена массивов. Если в некоторой ветви i встречается первый из операторов, то ветвь переходит в состояние ожидания до тех пор, пока во всех ветвях i_k управления не пришло к какому-нибудь оператору **принять,** в списке $j_1, \ldots, j_n$ которого имеется $j_l = i$ и при этом $y_l = x_k$. После этого происходит переписывание массива $x_k = y_l$ на место массива z_l. Массивы должны быть согласованы по размерностям и могут, в частности, вырождаться в простые переменные. Допускаются вырезки массивов (см. разд. 3.5). Для примера программы (5) при организации ветвей над разделенной памятью получим параллельную программу

ввод (x, y, u);
  fork
  $u := x + y$;
передать $(1 : u)$;
  $w := u * y$;
  $v := x * y$;
принять $(1 : u/v)$;
  $z := u/y$;
  join
вывод (w, v, w, z)

Сейчас в 1-й ветви вычисление не может двигаться вперед, пока не выполнится оператор **передать.** Если все-таки нужно выполнение последующих операторов, то оператор **передать** может быть ответвлен во внутреннюю ветвь. Оператор **принять** задает ожидание и инициирует перепись, которая в данном случае ведется из ячейки u в ячейку v.

При работе с иерархией ветвей при передаче данных между разными уровнями иерархии можно пользоваться составными номерами или именами ветвей. Так, М.2.1 будет обозначать ветвь, выделенную в следующей конструкции звездочкой:

```
        branch N, M
N:  . . .
    :
    :
    end N;
M:  . . .
    fork
     N
     :
     :
    fork
     *
     N
     :
     :
    join
     :
     :
    join
    end M
```

Для регулярных обменов могут быть предусмотрены сокращенные варианты примитивов **передать, принять.** Так, **передать** $(x_1, \ldots, x_m)$ обозначает передачу массивов x_i во все ветви, а **принять** $(y_1/z_1, \ldots, y_n/z_n)$ обозначает принятие массивов $y_1, \ldots, y_n$ из всех ветвей. В этом случае должно быть согласование массивов: один и тот же участок массива не должен вводиться сразу из нескольких ветвей. Для других типов обменов могут быть введены другие регулярные операторы, например, передающие из каждой i-й ветви в $(i+1)$-ю ветвь, во $2 * i$-ю ветвь (полная тасовка) и т. д.

Асинхронные средства программирования

Все описанные ранее параллельные операторы были связаны с понятием параллельных ветвей, каждая из которых задает последовательный порядок исполнения или последовательное исполнение групп по нескольку операторов. Такие типы программирования называют еще соответственно параллельно-последовательными или последовательно-параллельными. В противополож-

ность им можно рассмотреть более общий и гибкий тип программирования — асинхронное.

В асинхронной программе нет понятий параллельной ветви, параллельно исполняемых операторов, передачи управления и других элементов явного управления, идущих от последовательных программ. Включение операторов в асинхронной программе обусловлено выполнением некоторых условий готовности. Эти условия связаны с каждым фрагментом, динамически проверяются и разрешают или не разрешают выполнение данного фрагмента (оператора), в результате чего формируется вычислительный процесс. Условия готовности берут на себя всю организацию вычисления, так что в асинхронных программах нет разделения средств управления на те, что формируют параллельные ветви, и на средства синхронизации. Если программа имеет иерархическую структуру, но каждый «составной» фрагмент организован внутри по такому же принципу. Таким образом, асинхронный принцип программирования существенно отличается от последовательно-параллельного. Относительные достоинства всех подходов зависят от внутренней структуры программируемых задач и типа реализующих вычисления систем.

Рассмотрим два типа параллельных асинхронных программ: так называемые асинхронные, или А-программы, и программы в языках единственного присваивания. Аппарат А-программ можно реализовать над любым последовательным языком, в котором есть понятие логических условий. Центральным понятием А-программы является модуль: А-программа есть последовательность модулей или, лучше сказать, иерархическая система модулей. Синтаксис модуля следующий:

⟨модуль⟩ : := ⟨простой модуль⟩ | ⟨составной модуль⟩
⟨составной модуль⟩ : := (⟨список входов⟩)
⟨список обрамленных модулей⟩
вывод (⟨список выходов⟩)
⟨обрамленный модуль⟩ : := ⟨спусковая функция⟩ → ⟨модуль⟩
⟨спусковая функция⟩ : := ⟨логическое выражение⟩
⟨вход⟩ : := ⟨локальная переменная⟩ : ⟨внешняя переменная⟩
⟨выход⟩ : := ⟨внешняя переменная⟩ := ⟨локальная переменная⟩

Простой модуль является «атомарной» структурной единицей; он может быть, например, оператором при-

сваивания или вызовом процедуры. Если *спусковая функция* некоторого обрамления представляет собой логическую константу *истина*, то она может быть опущена. Пусть M — составной модуль, образованный списком обрамленных модулей $M_1, \ldots, M_n$. Тогда его локальная память, образованная переменными, входящими в спусковые функции модулей M_i, правые части входов модулей M_i (или первые части операторов присваивания), левые части входов модулей M_i (или левые части присваиваний). Локальная память модуля M является одновременно общей внешней памятью модулей $M_1, \ldots, M_n$. Таким образом, память в А-программе иерархически структурирована в соответствии со структурированием модулей. Память может содержать наряду с простыми переменными-ячейками и данные сложной структуры, например стеки, очереди и т. д.

Правила управления с помощью спусковых функций такова. Каждый модуль проходит последовательность состояний: пассивное, проверяемое, активное. Когда модуль пассивен, он не участвует в вычислениях. Когда он проверяется, находится значение его спусковой функции. Если спусковая функция равна единице, то модулю разрешается включиться. При этом спусковая функция не заставляет, а именно разрешает включение. Если нет свободного процессора, то модуль может подождать. Срабатывание модуля влечет переход к выполнению подмодулей низких уровней, а в конце концов — к изменению значений некоторых переменных. Одни из них, являющиеся информационными, участвуют в функции переработки данных. Другие — управляющие — влияют на значение спусковых функций модулей программы и участвуют в организации вычисления. Одна и та же переменная может играть обе роли. Иногда функция обработки информации и функция управления разделяются: модуль представляется в виде последовательности двух подмодулей, один из которых работает только над информационными переменными, а другой зависит от информационных и управляющих переменных, но изменяет только управляющие. В этом случае первый из них называется собственно модулем, а второй — управляющей функцией. Приведем простой пример А-программ, весьма похожих друг на друга:

а) *ввод* (x, y, z)
$x < y \to x := x + 1;$
$y < z \to y := y + 1;$
$z < x \to z := z + 1;$
вывод (x, y, z)

б) *ввод* (x, y, z)
$x \leqslant z \wedge x < y \to x := x + 1;$
$y \leqslant x \wedge y < z \to y := y + 1;$
$z \leqslant y \wedge z < x \to z := z + 1;$
вывод (x, y, z)

В примере а) А-программа состоит из единственного составного модуля. Модуль включает три обрамленных оператора и в результате вычислений устанавливает значения выходных переменных равными максимальному из значений входных переменных. Эта программа допускает параллельное исполнение двух из трех операторов. Программа б) функционально эквивалентна предыдущей, но «менее параллельна». В ней в каждый момент времени может работать только один из трех процессоров. Ограничения, добавленные к спусковым функциям, изменили набор возможных вычислительных процессов.

Можно иным способом изменить поведение А-программы, а именно: модифицировать правила ее исполнения. Например, проверки спусковых функций могут быть синхронизированы таким образом, что все те модули, которые были активны, должны перейти в пассивное состояние, и только после этого возможна новая проверка. Легко видеть, что это правило задает синхронное параллельное вычисление. Если, наоборот, и не более чем один модуль может быть в любой момент в активном состоянии, то А-программа становится последовательной программой с семантикой типа EXCL.

Итак, изменения в поведении А-программ могут быть сделаны двумя способами: 1) добавлением явных ограничений к спусковым функциям или 2) добавлением неявных ограничений к правилам исполнения. В обоих случаях наблюдается полезная «монотонность»: новая информация, подходящим образом добавленная к А-программе, уменьшает недетерминизм исполнения. Следовательно, поиск результата может быть организован пошаговым способом: сначала со-

ставляется набор независимых модулей, затем связывающие ограничения добавляются поочередно, пока не будет достигнуто желаемое взаимодействие между модулями.

В заключение изложения механизма спусковых функций приведем запись примера программы (5) в виде А-программы:

ввод $(x, y, u, t := 0; r := 1, V := 1)$
$V = 1 \rightarrow u := x + y;\ t := 1;\ S = 0,$
$S = 1 \rightarrow v := x * y;\ S := 0$
$t = 1 \rightarrow w := u * x;\ t = t + 1;$
$V = 0 \bigwedge S = 0 \rightarrow z := u/y;\ V = 3;$
вывод (u, v, w, z).

В этом примере спусковые функции двух первых модулей тождественно истинны, переменная t играет роль управляющей переменной: $t = 0$ говорит о том, что переменная u еще не перевычислена, а $t = 1$ — о том, что u уже перевычислена и можно включать модули, зависящие от нее.

Непроцедурные языковые средства

Как уже было замечено, языковые средства, в которых явно не указан порядок исполнения, называются *непроцедурными*. Языки, содержащие такие средства, называются также непроцедурными. Так, а-язык, рассмотренный в предыдущем разделе, является непроцедурным. Другие примеры дают язык ЛИСП и упоминаемый нами язык TRANQUIL. В последнем языке допускаются выражения, операндами которых являются массивы. Можно, например, записать $X := Y + Z$, где X, Y, Z — массивы размерности 10×10. Элементы результирующего массива определяются так: $X(i, j) = Y(i, j) + Z(i, j)$. При реализации этого матричного сложения так или иначе должны быть выполнены все сто сложений, но порядок их не фиксирован, так как в этой операции важен лишь конечный результат.

Таким образом, у реализующей системы есть выбор, в каком порядке выполнять сложения. И если вычислитель параллельный, естественно выполнить их параллельно на разных процессорах, стараясь уменьшить совокупное время выполнения. Точно так же и в случае других непроцедурных средств выбирают для реализации максимально или оптимально параллель-

ный режим. В этом заключается основная связь непроцедурности и асинхронности. Идя дальше, некоторые авторы ввели меру сравнительной непроцедурности языковых конструкций как включение множеств реализующих процессов. Пусть L_1 и L_2 — две языковые непроцедурные конструкции, задающие одну и ту же функцию, $P(L_i)$ — множество вычислительных процессов, реализуемых конструкцией L_i. Тогда L_1 называется более непроцедурной, чем L_2, если $P(L_2) \subseteq \subseteq P(L_1)$.

В наиболее завершенной форме непроцедурные матричные операции представлены в языке АПЛ. Направление развития языковых представлений в нем весьма естественно и подобно развитию языковых форм других наук, например математики. В математике вначале оперировали с матрицами и векторами в полной форме. Например, при сложении матриц писали:

$$\begin{pmatrix} a_{11} & a_{12} & \dots & a_{1m} \\ a_{21} & a_{22} & \dots & a_{2n} \\ \vdots & \vdots & & \vdots \\ a_{m1} & a_{m2} & & a_{mn} \end{pmatrix} + \begin{pmatrix} b_{11} & b_{12} & \dots & b_{1n} \\ b_{21} & b_{22} & \dots & b_{2n} \\ \vdots & \vdots & & \vdots \\ b_{m1} & b_{m2} & & b_{mn} \end{pmatrix},$$

но затем перешли к матричным обозначениям и стали записывать просто $A + B$. Для суммирования элементов массива $x_1 + x_2 + \dots + x_n$ использовали также специальный символ $\sum_{i=1}^{n} x_i$ и т. д. Подобные «сокращения» приняты в АПЛ. Рассмотрим основные конструкции этого языка.

Начнем со средств для сборки и перестройки матриц. В АПЛ есть оператор $\rho(m, n)$ который, будучи примененный к любому массиву чисел любой размерности общего объема mn, выстраивает его в матрицу $m \times n$. Например:

$$\rho(2, 3)(2, 4, 5, 6, 7, 8) = \rho(2, 3)\begin{pmatrix} 2 & 4 \\ 5 & 6 \\ 7 & 8 \end{pmatrix} = \begin{pmatrix} 2 & 4 & 5 \\ 6 & 7 & 8 \end{pmatrix}.$$

Если массив большего объема, то его «хвост» игнорируется, если меньшего объема, то результирующий массив дополняется циклическим повторением элемен-

тов. Так,

$$\rho(2, 3)(2, 4, 5) = \begin{pmatrix} 2 & 4 \\ 5 & 2 \\ 4 & 5 \end{pmatrix}.$$

С помощью этого оператора можно, в частности, формировать массивы с буфера ввода, располагать их удобным образом для процессора.

Вторая группа операций — аддитивные операции над матрицами — сложение и вычитание. Если

$$A = \begin{pmatrix} a_{11} & \dots & a_{1n} \\ \vdots & & \vdots \\ a_{m1} & \dots & a_{mn} \end{pmatrix}, \quad B = \begin{pmatrix} b_{11} & \dots & b_{1n} \\ \vdots & & \vdots \\ b_{m1} & \dots & b_{mn} \end{pmatrix},$$

$$A \pm B = \begin{pmatrix} a_{11} \pm b_{11} & \dots & a_{1n} \pm b_{1n} \\ \vdots & & \vdots \\ a_{m1} \pm b_{m1} & \dots & a_{mn} \pm b_{mn} \end{pmatrix}.$$

Особенностью этих операций является то, что они распространяются на случай одного скалярного операнда. Так,

$$\begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix} + 2 = \begin{pmatrix} 3 & 4 \\ 5 & 6 \end{pmatrix}.$$

Логические операции над матрицами определяются аналогичным образом. Например,

$$\begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix} > \begin{pmatrix} 2 & 1 \\ 2 & 5 \end{pmatrix} = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix};$$

$$\begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix} > 3 = \begin{pmatrix} 0 & 0 \\ 0 & 1 \end{pmatrix},$$

где матрицы правой части имеют нули и единицы уже типа «логические». Логические операции являются необходимым сервисом для механизма маскирования в матричных процессорах.

Следующая группа — редуцирующие по разным измерениям операции. Так, сложение всех элементов массива есть тривиальная редукция единичного измерения в нулевое. При $x = (1, 2, 3)$

$$+x = +(1, 2, 3) = 1 + 2 + 3 = 6.$$

При применении к матрице редукция может быть двух видов:

$$+/\begin{pmatrix}1 & 2 & 3\\ 4 & 5 & 6\end{pmatrix} = (1+4,2+5,3+6) = (5,\ 7,\ 9),$$

$$+/\begin{pmatrix}1 & 2 & 3\\ 4 & 5 & 6\end{pmatrix} = \begin{pmatrix}1+2+3\\ 4+5+6\end{pmatrix} = \begin{pmatrix}6\\ 15\end{pmatrix}.$$

Операция может быть обобщена, как и все другие, на произвольные размерности.

В языках типа АПЛ есть средства вырезки части матрицы, и, наоборот, конкатенации матриц. Так, если A — матрица, то:

$A(3:5,1)$ — 3, 4, 5-й элементы первого столбца,

$A(2,:5)$ — первые 5 элементов 2-й строки,

$A(2:4,\ 3:5)$ — подматрица, стоящая на пересечении 2-, 3-, 4-й строк и 3-, 4-, 5-го столбцов.

Конкатенация матриц задается записью

$$\begin{pmatrix}1 & 2\\ 3 & 4\end{pmatrix}\cdot\begin{pmatrix}5\\ 6\end{pmatrix} = \begin{pmatrix}1 & 2 & 5\\ 3 & 4 & 6\end{pmatrix}.$$

Она полезна при построении расширенных матриц, которые используются в линейной алгебре, при сборке таблиц и т. д.

Программирование задач моделирования

Как описываются параллельные процессы в языках моделирования? Один из примеров, на котором мы проиллюстрировали введение семафоров (см. разд. 2.3), относился как раз к моделированию. Поскольку семафорами представляются совместно протекающие и взаимодействующие процессы, они также представляют интерес для целей параллельного программирования. Правда, ситуация с моделированием в каком-то смысле противоположна. Если при параллельном программировании мы пытаемся разложить на параллельные процессы и пропустить на многих процессорах последовательно организованную задачу, то при моделировании мы описываем параллельно протекающие процессы, которые потом необходимо имитировать на последовательной ЭВМ.

Рассмотрим автоматическую мойку автомобилей, обслуживающую в каждый момент времени одну машину, при этом затрачивается, например, 10 единиц времени. Машины прибывают случайным образом и выстраиваются в очередь, но в один и тот же момент не могут прибыть два автомобиля. Имея список моментов прибытия машин, в течение дня можно восстановить весь процесс работы мойки: как интенсивно она была загружена, какая длина очереди была в произвольный момент t, каково среднее время обслуживания и т. д. Можно построить соответствующий график, используя два типа событий — прибытие машин и окончание обслуживания. Все это может сделать программа, если описать в ней необходимые элементы. Обычно для подобного рода задач, как и в нашем случае, модель содержит элементы двух типов — стационарные элементы и так называемые транзакты. Стационарные элементы могут быть либо свободны, либо заняты. Они введены для «обслуживания» элементов другого типа — транзактов, носящих временный динамический характер. Транзактам могут быть присвоены числовые или логические величины, называемые атрибутами. В нашем случае стационарным элементов является мойка, а транзакты — это массив автомобилей, каждый из которых характеризуется временем прибытия. Таким образом, имеется массив атрибутов — времен прибытия.

Приведем, быть может, не самую оптимальную программу, моделирующую работу мойки:

```
        k := 2;
        i := 1;
        t := 0;
    очередь := (  );
    пока t ≠ t(i) выполнять t := t + 1;
M:  ввести (авт (i));
    для t := t шаг 1 до t + 10 выполнять
    если t = t(k)
    то  {очередь := очередь. авт. k;
        k := k + 1};
    вывести (авт (i));
        i := i + 1;
    очередь := очередь/авт(i);
    на M;
```

Индекс i указывает на обслуживаемую в данный момент машину, k — на машину из конца очереди. Переменная t отсчитывает время на модели. Обслуживание начинается в нулевой момент времени, когда очередь пуста. Над ней определены две операции: *очередь. а* — установка элемента a в очередь, *очередь/а* — выход элемента a из очереди. Первый цикл служит для запуска работы — ожидается первая машина. Затем идет цикл, который каждые 10 тактов (время обслуживания автомобиля) устанавливает новоприбывших в очередь. Каждые 10 тактов производится мойка машин, соответствующие изменение k и очереди, начинается обслуживание новой машины.

Это простейший пример, где для моделирования нам, по существу, не пришлось еще применять какие-либо специфические для языков моделирования конструкции. В более сложных случаях используется значительно более широкий спектр средств. Так, объекты могут быть нескольких типов, их можно заводить и уничтожать, могут вводиться специальные отношения, например отношение следования машин в очереди. Очереди могут быть разных дисциплин обслуживания: FIFO — «первый вошел — первый вышел» — это обычная в нашем понимании очередь; FILO — «последний вошел — первый вышел» — это *стек* или *магазин*. Допустим и смешанные дисциплины обслуживания. В языке моделирования вводится понятие времени, и, как правило, можно оперировать тремя типами времен: реальным — для модели, системным — для языка, машинным — для ЭВМ. В языках также обычно присутствует понятие событий, иногда очень сложных, и определяются понятия процессов. Но нас интересует другой вопрос: а можно ли применить для моделирования средства параллельного программирования? Оказывается можно и достаточно эффективно.

Приведем пример записи программ с использованием операторов **fork**, **join**, **wait** и семафора:

$t := 0;$
$i := 1 := 1;$
$k := 2;$
очередь $:= (\ );$
пока $t \neq t(i) \wedge t < T$
выполнять $t := t + 1$
ввести (*авт* (i));

```
                    fork
для j := t + 1 шаг 1        для l := t + 10 шаг 10
до T выполнять              до T выполнять
если j = t(k)                   {вывести (авт (i));
то {P(S);                       i = i + 1;
    очередь := оче-             wait(k ≥ i);
    редь. авт(k);               P(S);
    V(S);                   очередь := очередь/авт(i)
    k := k + 1},                V(S);
                                ввести (авт (i))}
                    join
```

В дополнение в этой программе рассматривается верхняя граница T времени обслуживания. Процесс обслуживания естественным образом делится на два подпроцесса — управление очередью и собственно обслуживание, что и отражено в программе. Первая часть осуществляет установку вновь прибывших машин в очередь, вторая вводит и выводит их из мойки. Критический интервал — обращение к переменной *очередь* — «огражден» семафором S. Сейчас общее для ветвей время идет только до **fork**. После этого в левой ветви время отсчитывает переменная j, а в правой — переменная l. Они никак не согласуются, хотя можно было бы и ввести согласование. При вычислении на ЭВМ второй процесс может «забежать» вперед первого и начать обслуживать машины, которые еще не встали в очередь. Чтобы это исключить, процессы синхронизуются оператором **wait**. Программа может совершенствоваться и далее. Например, если въезд и выезд из мойки может осуществляться одновременно, то процедуры *вывести* и *ввести* 2-й ветви могут быть разложены в две параллельные ветви.

Приведенный пример иллюстрирует дополнительные возможности предложенных в главе средств параллельного программирования.

3. Операционные задачи

С точки зрения программы МВК может рассматриваться как множество различных ресурсов, необходимых для ее выполнения. В каждый момент времени для исполнения программы нужны, вообще говоря, не все, а лишь часть ресурсов МВК. Остальные ресурсы могут быть использованы другими программами. Комплекс специальных программ — *операционная система* — отвечает в МВК за организацию эффективных вычислений, оптимальное распределение ее ресурсов. Обсудим основные проблемы, связанные с организацией работы МВК.

Мультипрограммирование

Первые ЭВМ (рис. 3.1) имели все те основные узлы, что и современные вычислительные системы: центральный процессор (ЦП), оперативную память, каналы для передачи данных между компонентами ЭВМ, внешние устройства (ВУ) памяти, печати, ввода данных. Память была разбита на ячейки, пронумерованы целыми числами 0, 1, 2, ..., N, здесь N — номер последней ячейки памяти. Ячейка состоит из нескольких разрядов, например 8, 32, 48 или 64. Разряд может содержать значение 0 либо 1. Каждая ячейка могла содержать либо число, либо команду ЭВМ. Команда имела, например, такой вид: 85 00124 09348. При этом «85» ЦП трактует как код команды, к примеру, считает, что эта команда СЛОЖИТЬ, а 00124 и 09348 — как номера ячеек, в которых находятся числа — слагаемые. Та-

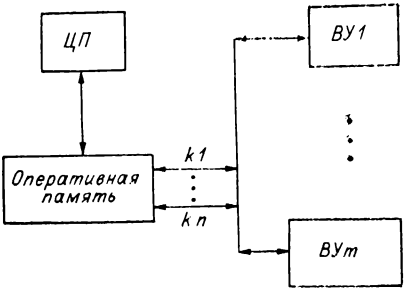


Рис. 3.1

ким образом, по команде 85 00124 09348 ЦП достанет из памяти, из ячеек 00124 и 09348, слагаемые, сложит их и результат (сумму) зашлет снова в память в ячейку 00124. Обычно имеется большой набор различных арифметических логических и управляющих команд. Приведем еще несколько примеров машинных команд, используя вместо кода команды ее название:

а) ПЕРЕСЛАТЬ 0012408743 — переслать содержимое ячейки 08743 в ячейку 00124;

б) ПЕРЕЙТИ-НА 35743 — команда передачи управления на ячейку 35743. Если эта команда находилась в ячейке 10000, то по ней центральный процессор вместо того, чтобы следующей выполнять команду из ячейки 10001, начнет выполнять команду, которую он достанет из ячейки 35743;

в) ПРОВЕРИТЬ 35743 00124 — проверить число в ячейке 00125, и если там нуль, передать управление в ячейку 35743, в противном случае выполнять следующую команду.

Нумерация ячеек в примерах сделана в десятичной системе счисления, хотя на самом деле в ЭВМ всегда используются двоичная, восьмеричная либо шестнадцатиричная системы, т. е. системы по основанию 2^1, 2^3 либо 2^4. Этого здесь достаточно, чтобы продемонстрировать основные понятия, а с другой стороны, позволяет не использовать системы счисления по основанию 2^k, $k = 1, 3, 4$, о которых все имеют представление, но активно владеют которыми лишь профессиональные программисты.

Каждый канал ввода/вывода — это тоже процессор, но специализированный для решения задачи обмена данными. К каналу подключается одно или несколько внешних устройств. Он имеет свою систему команд, таких как установить связь с устройством, начать прием передачу данных в/из ЭВМ, в/из памяти, начи-

ная с ячейки N в количестве k чисел, освободить устройство и т. п. Когда программе необходимо произвести обмен, она составляет *канальную программу* (т. е. программу, составленную из команд канала) и передает ее на исполнение каналу, который сам выполнит обмен, без использования центрального процессора, ЦП во время обмена может продолжать выполнение программы, если это возможно без новых данных.

Кроме того, первые ЭВМ имели еще обычно и огромный пульт управления, похожий несколько на пульт управления космическими кораблями, как их нам показывают в фантастических фильмах. На пульте многими рядами располагались индикаторы, тумблеры, кнопки, переключатели, с пульта программист мог вводить в ЭВМ программу, запускать ее на счет, останавливать в нужных местах, смотреть и изменять содержание любых ячеек и т. д. Опытные программисты могли работать за пультом в две и даже в четыре руки и внешне иной раз напоминали пианистов во время концерта. Таким образом, программисты имели *непосредственный доступ* к ЭВМ, имея в своем распоряжении все ресурсы машины. Это сильно облегчало отладку программ, но оборудование ЭВМ использовалось крайне неэффективно, даже тогдашние маломощные ЭВМ успели бы сделать сотни тысяч или миллионы операций за время, которое программист тратил за пультом, управляя ЭВМ «вручную».

Тогда управление процессом вычислений в ЭВМ поручили операционной системе (ОС) и родилось *мультипрограммирование* — режим работы ЭВМ, при котором в максимальной степени используются ресурсы ЭВМ. Суть его иллюстрирует следующий пример (рис. 3.2). В ЭВМ находятся две готовые к исполнению программы. Одна из них, например первая, «захватывает» ЦП (ОС выделяет ЦП первой программе) и начинает исполняться до момента t_1, когда ей потребуется начать ввод данных с внешней памяти через канал $k1$. Первая программа не может работать, пока не будут введены новые данные, поэтому она освобождает ЦП и переходит в состояние ожидания конца обмена, а ЦП начинает выполнение программы 2. Когда в момент t_2 завершится ввод данных по каналу $k1$ для программы 1, исполнение программы 2 прервется, и ЦП

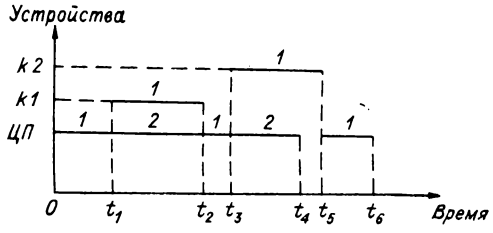


Рис. 3.2

продолжит исполнение программы 1. В момент $t_3 > t_2$ программа 1 начнет вывод данных по каналу $k2$, а ЦП продолжит исполнение программы 2 с прерванного места. В момент t_4 программа 2 завершится и ЦП будет простаивать до момента t_5, когда кончится обмен программы 1, она начнет считаться и завершится в момент t_6.

В этом примере программы 1 и 2 совместно, поочередно используют общий ресурс — ЦП (*разделяют* ЦП). Аналогичным образом могут разделяться и другие ресурсы: каналы, оперативная память, внешние устройства, файлы на магнитных дисках.

Существует много различных вариантов режима мультипрограммирования в зависимости от критерия, по которому оценивалось качество работы ОС. Можно выделить два основных варианта: пакетная обработка и разделение времени. Пока оборудование ЭВМ было дорогим, использовался в основном режим пакетной обработки. В этом режиме переход от выполнения одной программы к выполнению другой делался из соображений достижения максимальной производительности ЭВМ посредством разделения всех ресурсов. Задача ОС заключалась в том, чтобы избежать простоев наиболее дорогостоящего оборудования, в первую очередь центрального процессора. Интересы программиста, пользователя ЭВМ, напрямую не учитывались, терялся непосредственный доступ к ЭВМ, как следствие, сильно увеличивалось время на отладку программ и получение результатов. В пакетном режиме программа, получившая ЦП, использует его до тех пор, пока она в нем нуждается, например, пока она не начнет обмен. Дорогое оборудование ЭВМ (ЦП, память) используется хорошо, недостатком является то, что одна из программ может надолго «захватить» ЦП и блокировать все прочие. Кроме того, программист теряет

непосредственный доступ к ЭВМ, что крайне отрицательно сказывается на производительности его труда.

С удешевлением компонентов ЭВМ, повышением мощности ЦП стал возможен другой вариант мультипрограммирования. В режиме разделения времени каждой программе отводится квант времени ЦП для ее выполнения. По исчерпанию кванта времени выполняющаяся программа *А* прерывается и начинает исполняться другая (процессор переключается на другую программу), и так по кругу обслуживаются все программы. Затем квант времени ЦП получает вновь программа *А* и продолжает исполнение с прерванного места. Таким образом, все программы исполняются как бы одновременно, и несколько программистов будто бы имеют непосредственный доступ к ЭВМ (*квазипараллельное исполнение*).

Аппаратные средства поддержки мультипрограммирования

Для того чтобы ОС могла организовать мультипрограммную обработку, в ЭВМ необходимы специальные устройства. Прежде всего это *счетчик времени*. Счетчик часто выполняется в виде ячейки памяти, содержимое которой увеличивается либо уменьшается на единицу по сигналу генератора синхроимпульсов. Сигнал может поступать, к примеру, каждую микросекунду (1 мкс = $= 10^{-6}$ с). В режиме пакетной обработки счетчик времени используется для контроля зацикливания программы. Делается это так. Каждая программа перед началом исполнения сообщает ОС, что она будет использовать ЦП не более, к примеру, трех часов. Тогда ОС следит за тем, сколько времени центрального процессора использовала программа, и если по истечении трех часов программа не завершится, ОС «выбросит» ее из машины и сообщит об ошибке.

При пакетной обработке несколько программ могут одновременно находиться в памяти, некоторые из них могут, к примеру, отлаживаться, содержать ошибки и из-за этого портить тексты других программ, например, записав в ячейку, где была команда, какую-нибудь абракадабру. Таких случаев, конечно, не должно быть, и в ЭВМ встроили механизм *защиты памяти*. Механизмы эти бывают очень разные, мы рассмотрим защиту

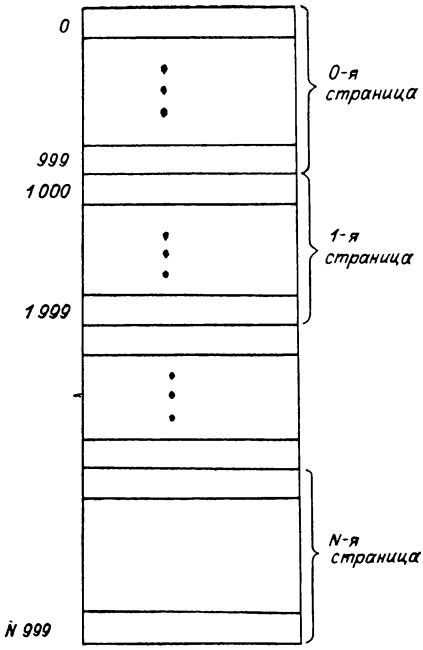


Рис. 3.3

по ключу, работает она так. Вся память ЭВМ разбивается на страницы (рис. 3.3), каждой странице сопоставляется специальная ячейка (программисту она не доступна) — ключ защиты страницы. Перед началом выполнения программы ОС присваивает ей уникальный код, к примеру 5. Этот код не имеет никакая другая программа, находящаяся в памяти ЭВМ. Этот же код 5 записывается в ключи всех тех последовательных страниц памяти, которые отведены программе для исполнения. Аппаратура ЭВМ при исполнении программы следит за тем, чтобы все ячейки памяти, к которым происходят обращения, располагались в страницах с ключом 5. При нарушении этого правила исполнение программы прерывается. Другой, гораздо более изящный способ распределения и защиты памяти рассмотрен в разд. 3.4.

Наличие в ЭВМ *системы прерываний* позволяет ЦП переключаться с выполнения одной программы на другую, реагировать на изменения ситуации в ЭВМ, на внешние условия. *Прерывание* — это сигнал, меняющий состояние ЦП. Всего ЦП при исполнении программ может находиться в двух состояниях: *подчиненном и привилегированном.* В подчиненном состоянии выполняются программы пользователей, в привилегированном — программы ОС, в этом состоянии не действуют никакие ограничения, например можно обращаться к любой ячейке памяти.

Механизм прерывания заключается в следующем. По сигналу прерывания прекращается выполнение те-

кущей программы и осуществляется переход к специальной подпрограмме, которая называется *программой прерывания*, ЦП переходит в привилегированное состояние. Все эти действия выполняются аппаратурой ЭВМ. Программа прерывания последовательно должна:

а) запретить временно другие прерывания, таким образом, все новые сигналы прерывания более не будут вызывать срабатывания аппаратуры, а только запомнятся;

б) запомнить всю информацию, необходимую для продолжения счета, например адрес команды, с которой следует продолжить выполнение прерванной программы;

в) выяснить причину, вызвавшую прерывание, и запустить на счет программу, которая среагирует на это прерывание (*программа обработки прерываний*);

г) восстановить информацию и продолжить счет с прерванного места в подчиненном режиме.

Можно выделить в общем виде четыре основных типа (причины) прерываний:

а) прерывания по обращениям за ресурсами (например, программе требуются канал и внешнее устройство памяти для обмена, по соответствующему прерыванию ОС выделит их программе);

б) прерывания по ошибкам в программе (деление на нуль, слишком большое число не поместилось в ячейке и т. п.);

в) прерывания по сбоям в машине (неисправная ячейка памяти, вышло из строя внешнее устройство);

г) внешние прерывания (нажата кнопка на пульте ЭВМ).

Управление очередями

Описанные аппаратные средства позволяют организовать выполнение программ в режиме мультипрограммирования. Готовые к исполнению программы вводятся в машину и поступают во входную очередь, с каждой программой поступает информация о требуемых ей ресурсах: объеме памяти, времени ЦП, внешних устройствах и т. п. Во входной очереди запоминается также время поступления в нее программы. Программа ОС (*системная программа*) — *планировщик* — получает уп-

равление, когда освобождаются ресурсы ЭВМ (ЦП, память), и решает вопрос, какая программа из входной очереди получит ресурсы и начнет считать.

Существует много разных дисциплин обслуживания очереди. Разумной является дисциплина, когда все программы в очереди обслуживаются одинаково, ни одна не имеет преимуществ перед другой. Достигнуть этого можно, выбирая из входной очереди программы на исполнение в порядке их поступления в очередь (первым пришел — первым обслужен). Реализация такого управления очередью очень проста и недорога. Основной недостаток дисциплины в том, что короткие программы (требующие ЦП на малое время) ждут столько же, сколько и длинные (требующие ЦП на большое время).

Следующая дисциплина основана на времени выполнения программы, а не на времени ее пребывания в очереди. Планировщик выбирает на исполнение ту программу, которой ЦП нужен на меньшее время. Короткие программы быстрее в этой дисциплине попадают на счет, но платится за это дорогая цена: время ожидания программы в очереди становится непредсказуемым. Вновь поступающие в очередь короткие программы постоянно оттесняют длинные («лезут без очереди»), в результате нельзя предсказать, когда же будет исполнена длинная программа: через час? через день? через неделю? А может быть программа будет «замурована» в очереди навечно?

Этого недостатка нет в дисциплине обслуживания очереди с *приоритетами*. В ней каждой программе сопоставляется число — приоритет. Например, программе можно сопоставить приоритет, равный $Пр = 1/T$, где T — время, в течение которого программе нужен ЦП. Ясно, что более короткие программы будут иметь бо́льший приоритет. Планировщик выбирает на исполнение программу с наибольшим приоритетом, либо любую из тех, что имеют одинаковый максимальный приоритет. Чтобы длинные программы не «застревали» в очереди, в приоритете следует учесть время ожидания в очереди, к примеру, считать приоритет по формуле $Пр = 1/T + k \times Тожидания$. Тогда с увеличением времени ожидания будет расти и приоритет, и в конце концов длинная программа также попадет на счет. С помощью коэффициента k можно регулировать степень влияния

ПРОЦЕССОРЫ
ПРИВИЛЕГИРОВАННЫЙ ПРОЦЕСС
ПРОЦЕСС

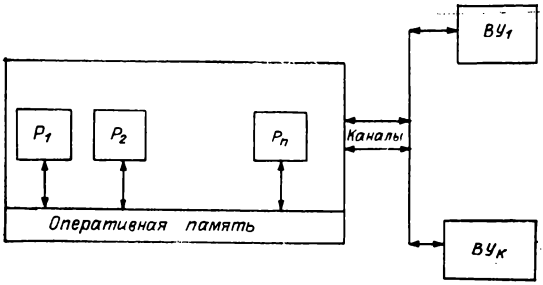


Рис. 3.4

времени ожидания на величину приоритета, в частности можно положить $k = T_{ожидания}$.

Существуют, однако, случаи, когда одного критерия (например, приоритета) недостаточно для выбора программы на счет. Такая ситуация возникает, к примеру, если один из пользователей платит за использование ЭВМ на 30 % больше других с тем, чтобы его программы раньше были просчитаны независимо от того, короткие они или длинные. В этом случае заводится еще одна *привилегированная очередь*, и всякая программа, попавшая в нее, имеет бо́льший приоритет, чем любая программа в общей очереди.

Описанные способы управления входной очередью годятся не только для однопроцессорной ЭВМ, но и для многопроцессорного вычислительного комплекса (МВК) (рис. 3.4). Освободившийся процессор P_m, $1 \leqslant m \leqslant n$, начинает выполнять программу операционной системы планировщик, в результате из входной очереди будет выбрана одна из программ пользователя на исполнение, и процессор P_m либо другой освободившийся к этому моменту процессор начнет ее исполнять. Нужно лишь предусмотреть, чтобы в случае, когда одновременно освободятся два или более процессора, лишь один имел возможность выбрать из входной очереди программу на исполнение. В противном случае может случиться, что одну и ту же программу будут выполнять два разных процессора (два разных экземпляра программы). Как это сделать, описано в разд. 3.5.

Отметим, что планировщик выбирает на исполнение программу из входной очереди не только из соображений очередности. Он учитывает также требуемые про-

грамме ресурсы МВК: нет ведь смысла назначать на исполнение программу, которой не могут быть выделены необходимые ресурсы. По этой причине очередность исполнения программ нарушается. Вообще, ситуация с распределением недостаточных ресурсов в ОС очень жизненна. И в жизни нередко привилегированные «лезут» вне очереди, и, к примеру, отстояв в очереди за авиабилетом, можно вместо билета получить совет зайти через часок.

Входную очередь составляют программы, ожидающие общий ресурс — МВК. Но точно так же в ОС используются очереди для распределения ресурсов самой МВК: памяти, процессоров, каналов, ВУ. Пусть, например, исполняющая программа A_1 начинает обмен и освобождает процессор, его начинает использовать программа A_2, которая в свою очередь через некоторое время начнет обмен и освободит процессор. Штатной в МВК является ситуация, когда исполняется программа A_k, а программы $A_1 - A_{k-1}$ уже закончили свои обмены и готовы к исполнению. В ОС все такие программы составляют очередь готовых к выполнению программ и специальная программа ОС — *диспетчер* — выбирает одну из них в соответствии с некоторой дисциплиной обслуживания и назначает её исполнение на первый же освободившийся процессор МВК.

Для управления очередями к разным ресурсам могут использоваться дополнительные критерии. В частности, из всех высокоприоритетных программ из очереди готовых программ полезно выбирать для исполнения ту, которой меньше осталось времени до завершения, что позволит раньше освободить ее ресурсы. Таким образом, для каждого ресурса ОС заводит свою очередь программ, требующих этот ресурс.

Общая схема работы ОС выглядит так: каждая программа, когда ей требуется некоторый ресурс (какой-либо процессор, память, канал и т. п.), обращается по прерыванию к ОС с запросом на него. Если есть свободный ресурс и очередь программ к нему пуста, ОС выделяет его программе и разрешает ей продолжить исполнение. А если нет — ставит программу в очередь к этому ресурсу, задерживает ее исполнение. Когда ресурс освободится, ОС просмотрит очередь к нему, выберет одну из программ, выделит ей ресурс и разрешит исполняться. Например, если освободился канал *К1*,

то ОС просмотрит очередь к *К1*, в данном случае это будет список всех канальных программ, сформированных разными программами для *К1*, выберет одну из них и запустит на счет в канале *К1*.

Виртуальная (математическая) память

Важным ресурсом МВК является оперативная память, ее необходимо уметь распределять в ОС быстро и эффективно, без простоев использовать. Для этого были разработаны специальные алгоритмы и аппаратура.

Самый простой и недорогой в реализации алгоритм распределения памяти состоит в том, чтобы каждой программе, выбранной из входной очереди, выделять всю необходимую ей память одним сплошным куском (рис. 3.5), в каждом куске целое число страниц, защита памяти выполняется по ключу. Однако у этого метода есть существенный недостаток. Пусть программа *В* завершилась раньше *А* и *С* (см. рис. 3.5). Если во входной очереди не найдется программы, которая могла бы разместиться на месте *В* или в остатке памяти, то свободная память будет простаивать. Из-за этого может простаивать и процессор, если *А* и *С* начнут обмены. (Заметим, кстати, что реально обмены не всегда вызывают освобождение процессора. Параллельные программы пишут обычно так, чтобы начать обмен как можно раньше, а процессор во время обмена продолжает выполнять программу до тех пор, пока это можно делать без поступающих новых данных.) При этом если бы участок памяти, отведенный *С*, сдвинуть вплотную к *А* (см. рис. 3.5), то в остав-

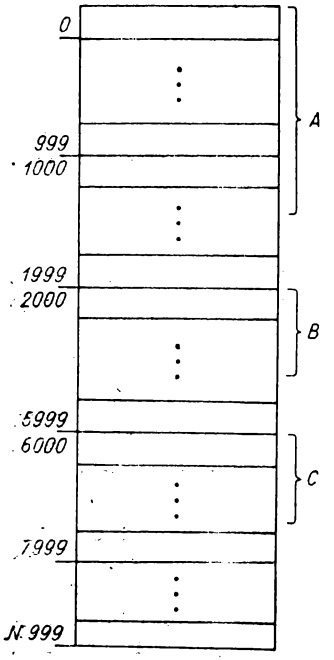


Рис. 3.5

шейся памяти уже смогла бы разместиться некоторая программа из входной очереди. С другой стороны, не всегда необходимо отводить место для всей программы, достаточно загружать ее по частям, каждая часть, отработав, загружает следующую и так до завершения.

Прежде чем переходить к изложению более совершенного алгоритма распределения памяти посредством *переадресации*, рассмотрим коротко процесс разработки программы. Изготовление программы проводится в несколько этапов, на каждом из которых используются свои инструменты. На первом этапе — подготовка текста программы — используется *текстовый редактор*, с помощью которого можно набрать текст программы на некотором языке программирования, например Паскале либо Фортране, вносить в него изменения, запомнить текст на внешнем устройстве памяти, обычно на диске.

На втором этапе — *трансляции* — вызывается на счет специальная программа-транслятор (*транслятор* с языка Паскаль, транслятор с языка Фортран и т. д.), которая преобразует подготовленный текст программы на языке программирования в *объектную* программу, почти готовую к исполнению, сообщает о синтаксических ошибках в тексте. Объектная программа состоит из машинных инструкций (команд, которые могут быть выполнены процессором), однако часть этих инструкций еще не до конца сформирована. На следующем этапе — *редактировании* — специальная программа — *редактор связей* — формирует *машинную* программу, готовую к выполнению. Редактор связей делает много разной работы, и в том числе разрешает внешние ссылки.

Суть здесь в следующем. Пусть, к примеру, в программе *A* есть обращение к подпрограмме (**subroutine**, **procedure**) *B*. Тогда в объектной программе *A*; получившейся после трансляции, в том месте *A*, где было обращение к *B*, будет находиться не полностью сформированная машинная команда ПЕРЕЙТИ-НА *B* (рис. 3.6, *а*), так как место расположения подпрограммы *B* транслятору неизвестно (она к моменту трансляции *A* может быть еще ненаписанной).

Редактор связей, найдя при просмотре *A* команду ПЕРЕЙТИ-НА *B*, находит в библиотеках подпрограм-

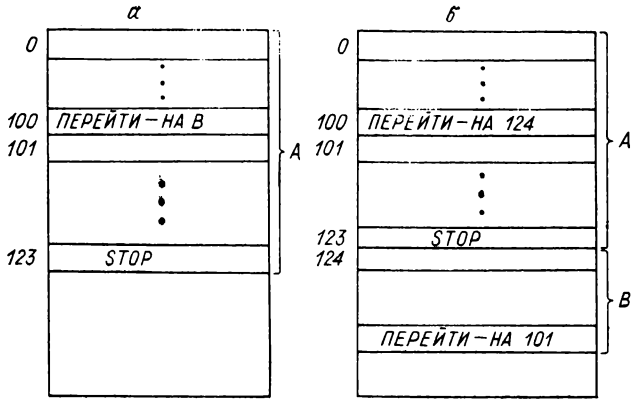


Рис. 3.6

му *B* и размещает ее в памяти сразу после программы *A* (на рис. 3.6, *б* в ячейках, начиная с 124). В этот момент и становится известным адрес нахождения *B* в памяти, и редактор связей заменяет команду-полуфабрикат ПЕРЕЙТИ-НА *B* машинной командой ПЕРЕЙТИ-НА 124. Этап редактирования иногда в явном виде отсутствует, и тогда на трансляцию поступает текст программы вместе с текстами всех вызываемых подпрограмм. Наконец, на последнем этапе машинная программа поступает в машину и исполняется под управлением операционной системы. На каждом этапе исправляются различные ошибки, повторяются они многократно, пока программа не будет отлажена. При формировании программы редактор связей считает, что для размещения программы в МВК всегда есть память, т. е. что МВК имеет как бы неограниченную память. Эта память и называется *виртуальной* (математической) памятью.

После редактирования программа будет готова к исполнению, записана во внешнюю память, но ей может понадобиться L страниц памяти (см. рис. 3.3), причем это число L может быть значительно больше числа N реальных, физических страниц памяти МВК. Задача состоит теперь в том, чтобы эффективно отобразить виртуальную память программы на физическую память МВК. Это отображение делается механизмом *переадресации* — специальной аппаратурой

процессора. Суть его заключается в следующем. Для каждой программы перед началом исполнения заводится *таблица переадресации* (рис. 3.7, *а*). В ее первом столбце — номера математических страниц, второй пуст — физические страницы программе еще не отведены. Кроме того, ОС ведет таблицу физических страниц (см. рис. 3.7, *б*), в которой перечислены свободные физические страницы и страницы, занятые программами. На рис. 3.7, *б* физические страницы 0 и 4 заняты программой *B*; 2 и 6 — программой *C*; страницы 1, 3, 5, ..., *N*-я свободны.

Приняв решение начать исполнение *A*, ОС ищет в таблице физических страниц свободную, нашла, например 5-ю, и записывает в 5-ю физическую страницу нулевую математическую страницу программы (т. е. команды, которые должны содержаться в ячейках 00000—00999). В таблицу переадресации во второй столбец нулевой строки записывается 5 (см. рис. 3.7, *в*), а в таблицу физических страниц в пятую строку во второй столбец — имя программы *A*, занявшей пятую физическую страницу памяти (см. рис. 3.7, *г*). После этого ОС назначает *A* один из свободных процессоров и *A* начинает исполняться с первой команды. Если при этом встречается команда СЛОЖИТЬ 00743 00520, то она выполняется так:

Вначале формируются физические адреса. В адресах 00743 и 00520 последние три цифры есть адрес ячейки внутри страницы, а первые две — номер математической страницы. По таблице переадресации процессор заменяет номер математической страницы на физический (в данном случае заменяет 00 на 05) и получает команду СЛОЖИТЬ 05743 05520 в физических адресах. После этого команда выполняется. Но в программе может встретиться и такая команда: ПЕРЕЙТИ-НА 04270. Однако четвертая математическая страница программы *A* еще не загружена в память (в четвертой строке таблицы переадресации во втором столбце еще пусто, см. рис. 3.7, *в*), по этой причине произойдет прерывание, ОС его обработает следующим образом.

Если в таблице физических страниц найдется свободная страница (в нашем случае это может быть третья), она отдается *A*, четвертая математическая страница *A* загружается в третью физическую, вно-

а

A

Номер математической страницы	Номер физической страницы
0	
1	
2	
3	
4	
5	
L	

б

Номер физической страницы	Имя программы
0	B
1	
2	C
3	
4	B
5	
6	C
N	

A в

Номер математической страницы	Номер физической страницы
0	5
1	
2	
3	
4	
L	

г

Номер физической страницы	Имя программы
0	B
1	
2	C
3	
4	B
5	A
6	C
N	

сятся изменения в таблицу переадресации и таблицу физических страниц (их состояние отображено на рис. 3.7, *д*, *е*), после чего начинает выполняться команда, которую процессор выберет из математической ячейки 04270, а фактически, после переадресации, из физической ячейки 03270.

Если свободной физической страницы не оказывается, ОС имеет две альтернативы. Может быть остановлено выполнение программы *A*, все находящиеся в физической памяти МВС математические страницы *A* запоминаются во внешней памяти (происходит *эвакуация A*), а освобожденные от *A* физические

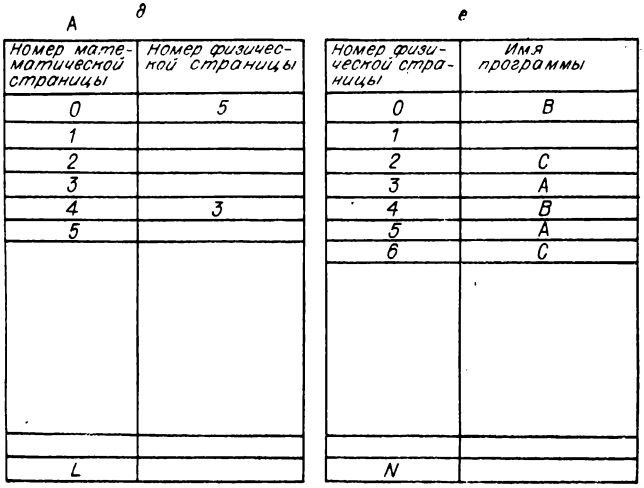


Рис. 3.7

страницы (в нашем примере 3 и 5) поступают в распоряжение ОС, которая может начать исполнение другой программы. А может быть эвакуирована, к примеру, программа B, и занятые ею физические страницы отданы A, причем нет необходимости эвакуировать все страницы B (как и ранее A), а можно и по одной.

Пять обедающих философов

Всякий раз при распределении ресурсов МВК возникают проблемы, связанные с необходимостью «развести» во времени программы, которым необходим один и тот же ресурс. Такие примеры уже встречались ранее в разд. 3.3. Только один процессор из всех свободных процессоров МВК может выполнять системную программу планировщик, которая выбирает из входной очереди на исполнение очередную программу. В противном случае два процессора могут взяться исполнять одну и ту же программу, считывая ее математические страницы каждый в свою физическую память. Ничего хорошего из этого не может получиться.

Точно так же лишь один процессор МВК может выполнять системную программу *диспетчер*, которая решает вопрос о том, какой программе выделить ре-

сурс и продолжить счет. Иначе может случиться, что один процессор решит эвакуировать программу B и продолжить исполнение A, а другой, наоборот, эвакуирует A и продолжит исполнение B. Аналогичная ситуация возникает, когда две программы, выполняющиеся на разных процессорах МВС, одновременно начинают обмен с одним и тем же файлом. В этом случае также необходимо сначала отдать файл в монопольное распоряжение одной программе на все время обмена, а затем передать его другой, тоже монопольно.

Для решения подобных задач в МВК существуют специальные команды, позволяющие работать с семафорами. Напомним, что *семафоры* — это особые переменные, значениями которых могут быть только целые неотрицательные числа, над которыми можно произвести только две команды — операции P и V. Операция $V(s)$ над семафором s добавляет единицу в s, т. е. операция $V(s)$ тождественна $s := s + 1$. Выполнение операции $P(s)$ эквивалентно выполнению программы

$$1 : \textbf{if } s > 0 \textbf{ then } s := s - 1 \textbf{ else goto } 1;$$

Кроме того, операции P и V неделимы, т. е. в любой момент может выполняться не более одной такой операции. Таким образом, если два процессора одновременно начнут выполнять, например, операцию P, аппаратура выполнит (по своему выбору) сначала операцию P одного процессора, а затем другого. На рис. 3.8 показано решение задачи «*читатели — писатели*». По условию задачи несколько *процессов* (исполняющихся программ)-*писателей* записывают в ячейку оперативной памяти (общий буфер) данные, из буфера они считываются процессами-*читателями*. Любое число процессов-*читателей* могут одновременно считывать данные из общего буфера, но только один *писатель* может данные в буфер записывать. Кроме того, *писатели* имеют приоритет перед *читателями*. Кроме программ *читатель* (*reader*) и *писатель* (*writer*) определены переменные, семафоры и установлены их начальные значения. Одновременно в МВК могут выполняться, например, 5 процессов-*читателей* и 3 процесса-*писателей*, т. е. 5 различных экземпляров программы *reader* и 3 различных экземпляра программы *writer*.

```
integer rc, wc
semaphore mutex1, mutex2, mutex3, w, r;
    rc = wc = 0;
    mutex1 = mutex2 = mutex3 = w = r = 1;
reader:                          writer:
begin P(mutex3);                 begin P(mutex2);
P(r);                            wc := wc + 1;
P(mutex1);                       if wc = 1 then P(r);
rc := rc + 1;                    V(mutex2);
if rc = 1 then P(w);             P(w);
V(mutex1);                       write;
V(r);                            V(w);
V(mutex3);                       P(mutex2);
read;                            wc := wc - 1;
P(mutex1);                       if wc = 0 then V(r);
rc := rc - 1;                    V(mutex2);
if rc = 0 then V(w);             end
V(mutex1);
end
```

Рис. 3.8

Программирование взаимодействий с помощью семафоров оказалось трудной, чреватой ошибками работой. Это хорошо видно всем, кто разбирался с примером на рис. 3.8. Для упрощения этой задачи в язык Паскаль встроены особые средства — *условные критические интервалы.* Работают они следующим образом. Ресурсы в программе представляются переменными, которые так и описываются как общие (**shared**), например:

var *v*: **shared;**

Общие переменные могут быть использованы только в специальном операторе, который и называется условным критическим интервалом:

region *v* **when** *B* **do** *S*,

где *B* — условие; *S* — оператор.

Из всех операторов вида **region** *v* ..., которые одновременно могут начать исполняться в разных процессах, лишь один получит возможность его выполнять, остальные будут задержаны и станут в очередь к общей переменной *v*. При выполнении сначала вычисляется значение *B*; если оно истина, то выполняется *S*, после выполнения *S* общая переменная *v* освобождается и следующий процесс получит право вой-

```
var v, w : shared; rr, aw : integer end;
reader: begin
    region v when aw = 0 do rr := rr + 1;
    reading;
    region v do rr := rr — 1
    end;
writer: begin
    region v do aw := aw + 1 await rr = 0;
    region w do writing;
    region v do aw := aw + 1;
    end;
```

Рис. 3.9

ти в критический интервал v. Если же B **ложь**, процесс уходит из критического интервала (освобождает v) и сам становится в очередь к переменной v, и так до тех пор, пока B не станет истинным.

Введен также «симметричный» оператор

region v **do** S **await** B;

Здесь процесс после вычисления S задерживается (не освобождает v) до тех пор, пока B не станет истинным и лишь после этого освободит v. Решение задачи *читатели — писатели* выглядит теперь много проще (рис. 3.9).

Рассмотрим теперь задачу о пяти обедающих философах, суть ее такова. Пять философов, прогуливаясь и размышляя, время от времени испытывают приступы голода. Тогда они заходят в столовую, где стоит стол, на нем всегда пять блюд, между которыми лежат пять вилок. Голодный философ а) входит в столовую, б) садится за стол, в) берет вилку слева, г) берет вилку справа, д) ест, е) кладет обе вилки на стол, ж) выходит из столовой. Необходимо организовать действия философов так, чтобы все они были накормлены и не случилось бы так, что пять философов одновременно войдут в столовую, возьмут левую вилку и застынут в ожидании освобождения правой вилки. Голодная смерть неминуема, если никто из них не захочет расстаться с левой вилкой. Будет не лучше, если они все одновременно положат левые вилки, а затем вновь одновременно же попытаются завладеть двумя вилками. Такая ситуация называется *тупиком* (смертельные объятия, *deadlock*) и в нее

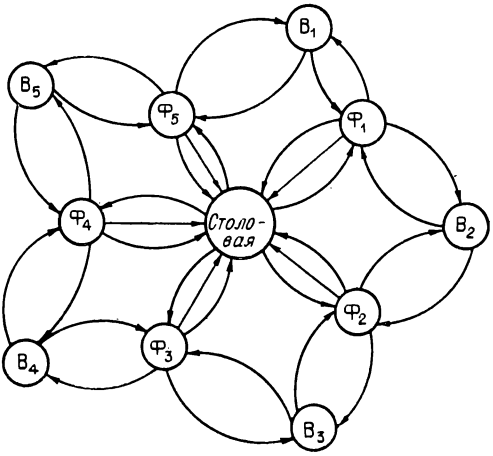


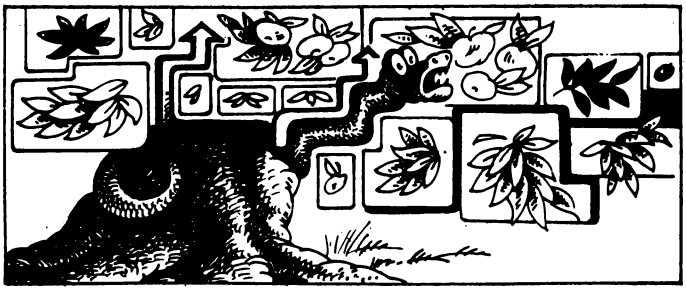
Рис. 3.10

лучше не попадать. Избежать ее можно, если в столовую пускать не более четырех философов, тогда один по крайней мере сможет овладеть двумя вилками. Необходимо также предусмотреть, чтобы два философа одновременно не схватили одну и ту же вилку.

При решении этой задачи следует учитывать самые разнообразные варианты поведения философов. Надо, например, предусмотреть, чтобы какой-нибудь стеснительный философ не умер голодной смертью из-за того, что его вилки постоянно раньше него хватают напористые соседи. Легко представить ситуацию, когда некая банда сговорившихся философов завладеет всеми вилками и, передавая их только в своей среде, уморит голодом всех прочих. Такая бандитская стратегия управления распределением вилок легко может быть и запрограммирована. Чтобы избежать такой ситуации, в систему распределения ресурсов должен встраиваться демократической механизм, позволяющий передать вилки стеснительным философам.

Для реализации этого механизма нужна централизация управления, которая бы обеспечила сбор информации об общем состоянии системы философы — столовая — вилки, что и позволит рационально распределять ресурсы (вилки). В частности, только на основе анализа глобального состояния системы можно выделить в среде философов банду и предотвратить

голодную смерть стеснительных философов. Решить эту проблему можно, например, следующим образом (рис. 3.10): а) фиксировать время обеда каждого философа, б) упорядочить всех философов по убыванию времени последнего полученного им обеда; в) пускать в столовую и выделять вилки прежде всего тем философам, кто дольше всех не обедал, и не пускать в столовую тех, кто пообедал недавно. В качестве времени можно использовать количество обедов, полученных каждым философом. При такой стратегии за каждый промежуток времени все философы пообедают примерно одинаковое число раз (на рис. 3.10, в1—в5 — процессы-вилки, Ф1—Ф5 — процессы-философы).



● Структуры данных

Для представления в программе объектов реального мира используются такие известные структуры данных, как целые и вещественные числа, строки символов, массивы чисел или строк. Их достаточно для решения многих вычислительных задач. При решении же задач невычислительного характера (нахождение номера телефона нужного человека в телефонном справочнике, решение геометрической задачи и т. п.) требуются более сложные структуры данных (записи, файлы, деревья, таблицы) и умение работать с ними.

Простые переменные и массивы

Прежде всего разберемся с понятием переменной. В программировании, в особенности в параллельном программировании, следует различать понятия *переменной алгоритма* и *переменной программы*. Эти понятия иллюстрирует следующий пример. Пусть дан треугольник со сторонами x, y, z, площадью s и углом α, заключенном между сторонами y и z (рис. 4.1, *а*). Если известны длины сторон x, y, z и угол α, то алгоритм вычисления площади s может быть описан так, как это показано на рис. 4.1, *б*. На нем прямоугольники изображают операции (действия, вычислительные акты), внутри которых записаны те вычисления, которые производят операции. Например, операция *а* вычисляет полупериметр треугольника. Каждая операция в программе может быть реализована процедурой или просто последовательностью

операторов языка программирования, таким, скажем, фрагментом для операции a:

$$p := x;$$
$$p := p + y;$$
$$p := (p + z)/2;$$

Переменные на рис. 4.1, *б* изображены точками. В целом эти переменные есть образ треугольника, как он изображен на рис. 4.1, *а*, но в виде, пригодном для машинной обработки. Каждая переменная изображает (называет) некоторое свойство реального объекта — треугольника, но еще ничего не говорит о его количественных характеристиках. Так, переменная x — длина стороны треугольника x, и т. д.

Таким образом, на рис. 4.1, *б* показано машинное представление всех возможных треугольников вида рис. 4.1, *а*. Чтобы изобразить конкретный треугольник, с конкретными длинами сторон, углом, площадью, необходимо переменным (всем или той части, которая полностью определяет, фиксирует объект) сопоставить *значения*. В нашем случае для определения конкретного треугольника достаточно определить значения любых трех переменных, например, сопоставив $x \leftrightarrow$

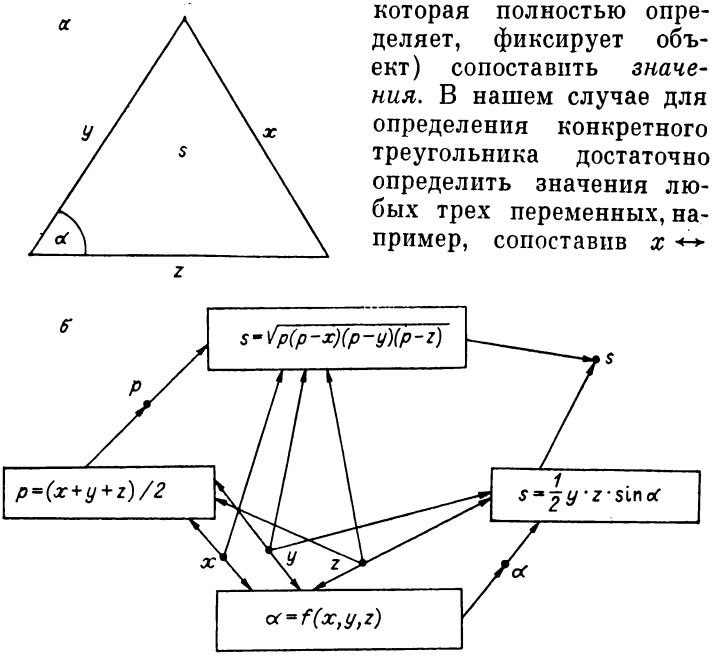


Рис. 4.1

$\leftrightarrow 3$, $y \leftrightarrow 4$, $z \leftrightarrow 5$. Из этих трех переменных могут быть вычислены значения всех других переменных, описывающих конкретный треугольник: площадь s и угол α. Так как каждая переменная есть имя некоторого свойства объекта, то для конкретного треугольника каждой переменной должно быть сопоставлено единственное значение, соответствующее конкретному объекту. В частности, в нашем треугольнике с длинами сторон $x = 3$, $y = 4$, $z = 5$ переменной s может быть сопоставлено (в исходных данных или вычислением) лишь одно-единственное значение $s = 6$. Значение s может быть вычислено двумя разными способами, двумя разными алгоритмами: выполняется операция a и вычисляется значение переменной p, затем операция b, ее результат есть s. Либо выполняется операция d, а затем s. При любом способе вычисления переменной s будет сопоставлено значение 6, и если бы это было другое число, например $s = 10$, можно было бы с уверенностью сказать, что такого треугольника: $x = 3$, $y = 4$, $z = 5$, $s = 10$ не существует в природе. Итак, в алгоритме каждая переменная получает единственное значение, являющееся количественной оценкой того свойства реального объекта, которое обозначает переменная.

Совсем другой смысл имеет переменная программы, которая обозначает ячейку памяти ЭВМ, предназначенной для хранения значения переменной алгоритма. И в этой ячейке могут попеременно храниться значения разных переменных алгоритма. Например, для вычисления площади s треугольника из длин его сторон может быть составлена программа:

```
procedure st(var s, x, y, z: real);
  s := (x + y + z)/2;
  s := sqrt((s * (s - x) * (s - y) * (s - z)));
end;
```

В этой программе переменная программы s последовательно хранит значения переменных алгоритма p (полупериметр) и s (площадь треугольника).

Поступать таким образом в программе приходится из-за необходимости экономить ресурсы ЭВМ (в данном случае память), в представлении же алгоритма делать этого не нужно, здесь важнее лучше понимать алгоритм, а не лучше исполнять его.

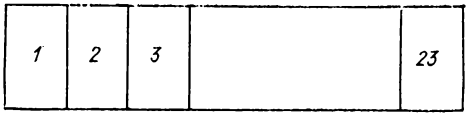


Рис. 4.2

Эффективное использование ресурсов ЭВМ в программе — одна из сложнейших задач программирования. В дальнейшем при конструировании сложных структур данных мы всегда будем иметь дело с переменными алгоритма и лишь иногда будем говорить о способах реализации (представлении в памяти ЭВМ) структур данных, т. е. о переменных программы всегда в этом случае делая особую оговорку.

Хорошо известной структурой данных является *массив*. Массив — это множество переменных, массив имеет собственное имя, например *mac* 1, каждый элемент множества (*элемент массива*) также имеет имя. Оно составляется из имени массива и целочисленных индексов, например: *mac* 1 (1), *mac* 1 (5), *mac* 1 (2 3) — это элементы одномерного массива (используется один индекс). Такой массив можно представить в виде линейно упорядоченного (по возрастанию значения индекса) множества переменных (рис. 4.2):

{*mac* 1 (1), *mac* 1 (2), ..., *mac* 1 (2 3)}.

Для именования элементов двумерного массива используются два индекса (*i*, *j*), например: *mac* 2 (1, 3), *mac* 2 (2, 2), *mac* 2 (3, 2). Такой массив может быть изображен в виде таблицы (рис. 4.3). Все элементы массива с одним и тем же значением первого индекса образуют строку. Например, при $i = 2$ элементы *mac* 2 (2, 1), *mac* 2 (2, 2), *mac* 2 (2, 3) составляют вторую строку, двумерного массива *mac* 2. Элементы массива с одним и тем же значением второго индекса образуют столбец.

Массивы есть и в программе. Там массив *mac* 1 есть последовательность ячеек памяти с именами *mac* 1 (1), *mac* 1 (2), ..., *mac* 1 (2 3). Если массив *mac* 1 именует участок памяти начиная с ячейки 20001, то *mac* 1 (1) именует ячейку 20001, *mac* 1 (2) — 20002, ..., *mac* 1 (2 3) — 200023. Если двумерный массив *mac* 2 именует участок памяти начиная с ячейки

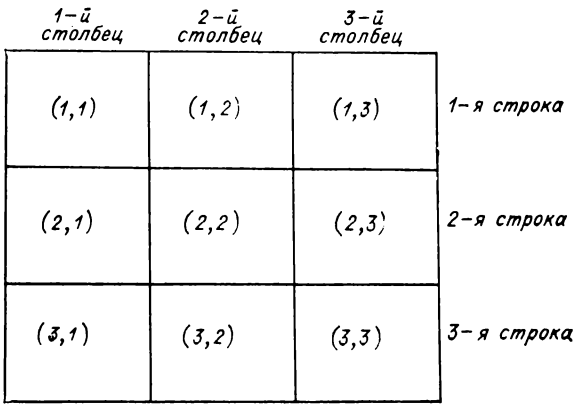


Рис. 4.3

30001, то *mac* 2 (1, 1) именует ячейку 30001, *mac* 2 (1, 2) — 30002, *mac* 2 (1, 3) — 30003, *mac* 2 (2, 1) — 30004, *mac* 2 (2, 2) — 30005, ..., *mac* 2 (3, 3) — 30009, т. е. элементы двумерного массива располагаются в одномерной памяти ЭВМ построчно. Но это не всегда так, в некоторых языках программирования массивы размещаются по столбцам.

Рассмотрим алгоритм печати первых 100 натуральных чисел. Каждое i-е натуральное число, $i = 1, 2, \ldots, 100$, является значением переменной $nat(i)$, *nat* — массив. Переменной $nat(1)$ сопоставляется значение 1, а переменным $nat(j)$, $j = 2, 3, \ldots, 100$,— значение, вычисленное по формуле $nat(j) := nat(j-1) + 1$;. Программа может иметь такой вид:

```
var nat: array [1 ... 100] of integer;
    nat [1]:= 1;
    print (nat [1]);
for   j := 2 to 100 do
begin nat [j] := nat [j] + 1;
      print (nat [j]);
end;
end;
```

Здесь в примере для хранения значения каждой переменной алгоритма отводится своя ячейка. В ре-

альных программах так, конечно, не делается, в данном случае выгоднее хранить значения всех элементов массива последовательно в одной ячейке и составить программу так:

```
var nat: integer;
    nat := 1;
    print (nat);
for i := 2 to 100 do
    begin
        nat := nat + 1;
        print (nat);
    end;
end;
```

Эта программа гораздо более эффективно использует память ЭВМ и даже будет быстрее исполняться в некоторых архитектурах.

Массив является сложной, составной структурой данных, его компонентами являются более элементарные переменные. В частности, элементами массива могут быть вновь массивы. В языке Паскаль двумерный массив можно описывать двумя способами:

1) двумерный описатель:

```
var m: array [1: ... 10], [1 ... 5] of real;
```

2) одномерный массив массивов:

```
var m: array [1 ... 10] of array [1 ... 5] of real;
```

В случае 2) описан одномерный массив, каждый элемент которого в свою очередь есть массив целых чисел. В отличие от сложных структур данных переменные, не имеющие составных частей, называются *простыми.*

Другие сложные структуры данных

Всякая новая структура данных строится, как и любой сложный объект, из компонентов, построенных к текущему моменту. В нашем распоряжении сейчас (как строительный материал) есть простая переменная и массив. Как известно, в языках программирования простая переменная имеет *тип*, характеризующий, в первую очередь, множество значений, которые

могут сопоставляться переменной. Это типы **integer** (целые), **real** (вещественные), **char** (строка символов), **boolean** (логический, имеет всего два значения: *истина* и *ложь*). Тип переменной определяет и выбор операций, которые могут быть произведены над значениями. В операторе $x := y + z$; операция + берется целочисленная, если x и y имеют тип **integer**, и вещественная, если хотя бы одна из переменных x и y имеют тип **real**. Это существенно разные операции, т. е. целочисленное и вещественное сложения выполняются по разным алгоритмам. В языках программирования тип переменной контролируется, т. е. в процессе трансляции проверяется, что переменная получает допустимое значение. Например, переменная, описанная **var** x: **char**;, должна в качестве значения получать только строку символов. Но смысл значения, конечно же, не контролируется, и если описана переменная

var *содержимое-холодильника*: **char**;

то ничто не помешает выполнить такой оператор присваивания:

содержимое-холодильника := 'паровоз';

Массив — это уже сложная (составная) структура данных, его составляющие (компоненты, элементы) есть простые переменные перечисленных типов, либо же вновь массивы (массив массивов). Однако, взяв элемент такого массива массивов (это будет массив) и взяв элемент элемента, мы будем в конечном итоге иметь дело с обычной простой переменной. Хорошо известные арифметические операции не определены над массивами, они могут выполняться только над простыми переменными. Поэтому, сконструировав сложные структуры данных, необходимо сконструировать и операции для их обработки, либо привести алгоритмы, конструкции, с помощью которых можно получить доступ к компонентам. Для массивов, в частности, такой конструкцией является имя компоненты, и любую компоненту двумерного массива *mac* 2 можно выбрать по значениям индексов *mac* 2 (*i*, *j*) и уже к ней применять операции. Для определения массовых вычислений иногда полезно доступ к компонентам задавать заранее фиксированным алгоритмом,


DO 1I=1,10
WRITE(1'10)A

например:

```
begin
  var x: array [1...1000] of array [1...10] of real;
      y: array [1...1000] of array [1...10] of real;
      z: array [1...1000] of array [1...10] of real;
      z := x + y;
end.
```

Здесь операция «+» не определена над массивами массивов x и y. Для решения вопроса о применимости операции «+» в некоторых языках используется алгоритм «просачивания». Делается попытка применить операцию «+» ко всем элементам массивов x и y покомпонентно. В данном случае это невозможно, так как эти элементы — массивы. Тогда пытаются применить «+» ко всем элементам элементов x и y. Это простые переменные, «+» к ним применима и выполняется для всех компонентов. Происходит как бы «просачивание» операции «+» сквозь структуру для простых переменных. В результате выполняется покомпонентное сложение двумерных массивов.

Очень распространенной структурой данных является *запись*. Назовем ее характерные особенности:

1) компоненты записи могут быть неоднородными в отличие от массива, все компоненты которого имеют одну и ту же структуру и типы;

2) каждая компонента записи имеет имя, пример:
книга = (*индекс*, *название*, *автор*, *издательство*, *цена*, *год-издания*).

Этой записи, как и всякой переменной, может быть сопоставлено значение: *книга* = (113781, Теремок, С. Маршак, «Малыш», 0.3, 1988), которое определяет конкретную книгу. На языке Паскаль запись описывается так:

```
type книга = record
               индекс: integer;
               название, автор, издательство: char;
               цена: real (в рублях);
               год-издания: integer;
```

Компоненты записи называются иногда еще полями, например, поле *автор*, поле *цена* в записи *книга*. Обращение к полям записи делается по имени. Например, *книга.автор* есть простая переменная, ей со-

поставляется значение — строка символов, обозначающая имя автора книги, такое сопоставление будем изображать так:

книга.автор := 'С.Маршак';

В языке Паскаль точно так же выглядит оператор, присваивающий значение 'С. Маршак' полю *автор* в записи *книга*. Не следует думать, что при реализации записи значения ее компонент обязательно будут располагаться в соседних ячейках в том порядке, в каком они описаны при определении записи. Это совсем не обязательно.

Каждая запись может уже изображать устройство довольно сложных объектов реального мира, например:

человек = (*Ф. И. О.*, *год-рождения*, *рост*, *профессия*)
завод = (*название*, *адрес*, *расчетный-счет*)
кинофильм = (*название*, *киностудия*, *год-выпуска*, *режиссер*)

Понятно, что при определении записи в нее вносятся лишь те поля (те свойства реальных объектов), которые необходимы для решения задачи. Таким образом, объект в программе изображается очень неполно, фрагментарно.

Кроме конкретных объектов, необходимо уметь изображать и множества таких объектов. Переменная, которой могут в качестве значения быть сопоставлены одно или несколько значений, либо особое *пустое значение*, называется *множеством*. Множество по сравнению с массивом имеет два отличия:

1) элементы множества могут иметь различную структуру;

2) элементы множества не упорядочены.

Рассмотрим пример множества.

содержание-ящика = {**char**, *книга*}.

Фигурные скобки означают, что описывается множество, оно состоит из переменных, структуры которых перечислены внутри фигурных скобок, количество их произвольно. Вот примеры значений множества *содер-*

жание-ящика:

содержание-ящика = {ручка, карандаш, (113781, Теремок, С. Маршак, Малыш, 0.3, 1988), кнопка}.
содержание-ящика = {тетрадь, скрепка}.
содержание-ящика = {(113781, Теремок, С. Маршак, Малыш, 0.3, 1988), (578711, Кортик, А. Рыбаков, Современник, 1.24, 1983)}.

Здесь фигурные скобки использованы и для указания значения множества с именем *содержание-ящика*. Ясно, что {тетрадь, скрепка} и {скрепка, тетрадь} есть одно и то же значение множества *содержание-ящика*.

Заметим, что каждый раз при определении более сложной структуры данных, чем простая переменная, необходимо указывать способ (алгоритм) доступа к компонентам ее значения. Для работы с множествами используются специальные функции (а в языках соответствующие им процедуры для реализации доступа к элементам множества). Например, если переменная *M* есть множество, ее значение {*A*, *B*, *E*, *C*}, в программе описана переменная

var *y*: **char**;

то после выполнения оператора присваивания *y* := *любой* (*M*); значением переменной *y* станет некоторый (неизвестно какой) элемент множества *M*, либо пустое значение, если множество *M* пусто. Можно применять и функцию *другой* (*M*), с помощью которой последовательно перебираются все элементы *M*. Здесь уместно отметить, что различные сложные структуры данных появились не потому, что без них вовсе нельзя обойтись в программировании, но встроенные в различные языки и системы программирования они существенно облегчают разработку программ. И, кстати, массив может, конечно же, рассматриваться как множество, к каждому элементу которого можно обратиться по имени.

В языке СЕТЛ множество является одной из встроенных структур данных наряду с простыми переменными и записями. Множества весьма неэффективно реализуются на современных ЭВМ и по этой причине редко используются в программах. Некоторые

специальные виды множеств, однако же, оказались полезными и эффективно реализуемыми и нашли широкое применение в программировании. Прежде всего имеется в виду *отношение* (*таблица*), все элементы значения которого имеют одну и ту же структуру — структуру записи, например.

1) *Толпа* = {*человек*}, значение множества *толпа* есть множество, которое может содержать только значения записи вида *человек*, например:

толпа = {(И. В. Иванов, 1950, 190, токарь),
(А. П. Сидоров, 1940, 165, плотник),
(С. С. Петров, 1945, 173, слесарь)}.

Значение отношения изображается обычно в виде таблицы:

ТОЛПА

Ф. И. О.	Год рождения	Рост	Профессия
И. В. Иванов	1950	190	Токарь
А. П. Сидоров	1940	165	Плотник
С. С. Петров	1945	173	Слесарь

2) *каталог* = {*книга*}

КАТАЛОГ

Индекс	Название	Автор	Издательство	Цена	Год издания
113781	Теремок	С. Маршак	Малыш	0.3	1988
578711	Кортик	А. Рыбаков	Современник	1.24	1983
819333	Рассказы	В. Шукшин	Просвещение	2.55	1985
393816	Избранное	Л. Толстой	Худ. литература	5.40	1987

Отношения реализуются двумя основными способами. Если отношение целиком может быть размещено в оперативной памяти ЭВМ (т. е. число элементов отношения невелико), то оно определяется как массив записей:

var *книги* [1 . . . 100] **of** *книга*;.

Здесь определена таблица из 100 элементов, каждый имеет ту же структуру, что и запись *книга*. Для определения отношения может быть и особый описатель:

var *толпа*: **set of** *человек*;

Если таблица большая, она размещается во внешней памяти ЭВМ и называется *файлом*. Каждая запись файла может быть считана в оперативную память ЭВМ и, наоборот, записана из оперативной памяти в файл, для этого в языках программирования есть специальные операторы, в языке Паскаль это операторы **put**, **get**, **write**, **read**.

Отношения, реализованные в форме файлов, есть в большинстве языков программирования. *Отношение* — универсальная структура, с помощью отношений удобно и наглядно описываются информационные математические модели реальных явлений, которые в этом случае довольно эффективно реализуются.

Следующий полезный специальный вид множества — *очередь*. Отношения используются в тех случаях, когда порядок использования элементов множества несуществен. Например, если необходимо узнать о предмете разговора в толпе, достаточно взять элемент *любой* (*толпа*) и узнать, в чем дело. А если толпа желающих приобрести товар собирается у магазина, они (чтобы не получилось драки) выстраиваются в *очередь* — линейно упорядоченное множество, с особыми правилами поступления новых элементов в очередь и выбытия элементов из очереди. О любых двух элементах очереди известно, какой из них находится раньше в очереди, и если Иванов располагается в очереди раньше Петрова, а Петров — раньше Сидорова, то Иванов в очереди находится раньше Сидорова. Каждый новый элемент, попадая в очередь, становится последним, т. е. все прочие элементы очереди будут располагаться раньше его. Выбран из очереди может быть только первый элемент, тот, который находится в очереди раньше всех других. Для работы с очередью существуют такие функции, как *поместить-в-очередь*, *выбрать-из-очереди* и т. п. Примеры задач, для решения которых полезно использовать очереди, приведены в гл. 3.

Весьма распространенной структурой является *магазин* (*стэк*) — линейно упорядоченное множество. От

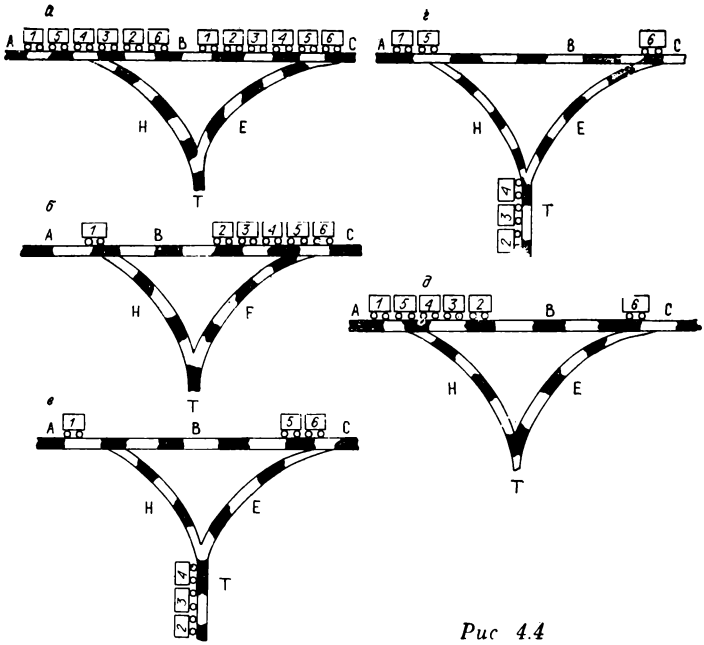


Рис 4.4

очереди магазин отличается одним: вновь поступивший в магазин элемент становится в нем первым, а не последним. Магазин нередко реализуется в аппаратуре ЭВМ, и среди ее команд появляются команды вида *поместить-элемент-в-магазин*, *выбрать-элемент-из-магазина* и т. п.

Использование магазина рассмотрим на задаче сортировки вагонов железнодорожного состава. Пусть имеется сортировочная станция, состоящая из участков путей *A*, *B*, *C*, *E*, *H* и тупика *T* (рис. 4.4, *а*). На участке *C* стоит состав, его вагоны перенумерованы, состав надо перегнать на участок *A*, переформировав по дороге. Вагон *1* должен остаться первым и он просто по участку *B* перекатывается на свое место (см. рис. 4.4, *б*). Вагоны *2—4* в новом составе должны следовать в обратном порядке и они закатываются по участку *E* в тупик *T* (см. рис. 4.4, *в*). Вагон *5* перекатывается по участку *B* на участок *A* и пристыковывается к вагону *1* (см. рис. 4.4, *г*). Далее вагоны *4*, *3*, *2* из тупика *T* по участку *H* перекатываются на уча-

сток A и ставятся в новый состав (см. рис. 4.4, *д*). Остается теперь лишь перекатить вагон *6* по участку B к его новому месту (см. рис. 4.4, *а*). Совершенно очевидно, что тупик T легко реализуется в программе с помощью магазина.

Поиск данных методом деления пополам

Отношения, реализованные в форме файлов или таблиц, используются практически в каждой программе и от умения быстро находить необходимые элементы отношения существенно зависит эффективность программы.

В задачах, при решении которых порядок обработки элементов отношения несуществен, используется последовательный просмотр элементов отношения, начиная с первого и до последнего. Таких задач очень много и, как следствие, практически во всех системах программирования есть последовательные файлы. Их вполне хватает для решения задачи печати всех элементов отношения в том порядке, в каком они размещаются в файле (хватает в том смысле, что алгоритм решения задачи просто реализуется в виде программы и программа получается эффективной).

Однако уже для решения задачи печати отношения *каталог*, элементы которого упорядочены по фамилиям авторов, последовательных файлов недостаточно, необходимо вначале упорядочить файл (произвести сортировку). Понятно, что ускорение поиска нужных элементов множества возможно лишь за счет дополнительных знаний о внутреннем его устройстве. Точно так же по лабиринту можно быстро пройти из точки A в точку B, зная лишь его схему. Упорядочивание как раз и дает такие знания.

Существует много различных алгоритмов упорядочивания, каждый из них по-своему хорош. Наиболее известный из них — алгоритм пузырька. Пусть дано отношение T, содержащее n элементов, элементы T перенумерованы как строки таблицы, i-ю строку обозначим $T(i)$. На T определена функция $K(T(i), T(j))$ для сравнения элементов отношения, и если она меньше нуля, то считается $T(i) < T(j)$, если больше нуля, то $T(i) > T(j)$, если равна нулю, то элементы равны.

Элементы отношения сравниваются обычно по значениям каких-либо полей. Например, в отношении *каталог* элемент (113781, Теремок, Маршак С., Малышев, 0.3, 1988) может считаться меньшим, чем элемент (578711, Кортик, Рыбаков А., Современник, 1.24, 1983), так как в его поле *автор* стоит значение 'Маршак С.', предшествующее в лексикографическом порядке значению 'Рыбаков А.' Обычно для сравнения элементов отношения используется не весь элемент, а лишь часть его (ключ), состоящая лишь из тех полей, значения которых однозначно определяют элемент.

Если договориться, что в каталоге может быть представлена только одна книга каждого автора, то ключом отношения *каталог* может быть поле *автор*. Если несколько разных по названию книг одного автора могут быть представлены в каталоге, то для того чтобы их различать, в качестве ключа следует взять уже два поля: (*название*, *автор*). Функция K определяется на полях ключа отношения. Необходимо помнить, что так как отношение — это множество, то из двух элементов $T(i)$ и $T(j)$ таких, что $K(T(i), T(j)) = 0$, один должен быть удален из отношения. Теперь алгоритм пузырька выглядит следующим образом:

А. Полагается $j := n$.

Б. Сравниваются элементы $T(j)$ и $T(j-1)$, если $T(j-1) > T(j)$, то они меняются в таблице местами, в противном случае остаются на своих местах. Таким образом, меньший, более «легкий» элемент как бы всплывает вверх на одну строку в таблице.

В. Далее последовательно сравниваются пары элементов $(T(j-1), T(j-2)), \ldots, (T(2), T(1))$. В результате в первую строку всплывет минимальный элемент отношения.

Г. Затем сравниваются элементы $(T(n), T(n-1)), \ldots, (T(3), T(2))$, в результате в строку $T(2)$ всплывет минимальный элемент из всех элементов $T(2), \ldots, T(n)$; учитывать $T(1)$ нет необходимости, он и так самый маленький. Сравнения продолжаются до тех пор, пока все элементы не займут свои места.

Нетрудно теперь написать программу на языке Паскаль, реализующую этот алгоритм. Алгоритм незначительно усложняется, если при упорядочивании удалять один из двух равных элементов. Для поиска нужного

элемента в упорядоченном отношении используется алгоритм деления пополам. Алгоритм этот можно проиллюстрировать на решении задачи поимки африканского льва. Поймать льва в клетку очень просто, для этого надо взять Африку и разделить ее решеткой пополам. Затем взять ту половину Африки, в которой находится лев, и вновь поделить ее решеткой пополам. И так до тех пор, пока Лев не будет заключен в клетку.

Аналогичным образом ищется элемент в отношении. Элемент лексикографически упорядоченного по полю *автор* отношения *каталог*, в котором описана книга Маршака С., ищется так. Берется средний элемент, т. е. $T([n/2])$, где $[n/2]$ — целая часть числа $n/2$, и проверяется с помощью функции K, в какой половине должна находиться книга Маршака С., затем эта половина вновь делится пополам, делается проверка и так до тех пор, пока либо элемент, описывающий книгу Маршака С., будет найден, либо будет доказано, что такого элемента нет. Если в отношении *каталог* содержится 2^m элементов, то за m попыток нужный элемент будет найден, либо доказано его отсутствие в отношении.

Пусть теперь в отношении *каталог* допускается несколько элементов, описывающих разные по названию книги одного и того же автора, в качестве ключа отношения используются поля (*автор*, *индекс*), отношение лексикографически упорядочено, причем из двух элементов с одинаковым значением поля *автор* меньшим считается тот, у которого меньшее значение поля *индекс*. Здесь сравнение элементов (вычисление функции K) производится по двум полям: сначала по более старшему (полю *автор*), и элемент с меньшим значением поля считается меньшим; если элементы не сравнимы (значение поля *автор* у них одинаковое, например 'Маршак С.', то сравниваются значения младшего поля ключа — поля *индекс* (предполагается, что все книги любого автора имеют различные значения этого поля). Так может быть сконструирована функция K для любого ключа, а упорядоченное таким образом отношение называется *лексикографически* упорядоченным по этому ключу. Нетрудно видеть, что именно в этом смысле лексикографически упорядочены слова в словаре.

Поиск данных по функции перемешивания

Рассмотрим метод доступа к элементам таблицы по функции перемешивания для решения задач, в которых необходимо многократно пополнять таблицу новыми элементами, находить необходимые и уничтожать ненужные элементы, размер таблицы (максимальное число элементов в ней) зафиксирован.

Таблица в языке Паскаль может быть реализована как массив записей, например:

```
type книга = record
               номер: integer;
               название, автор: char;
               end;
var каталог: array [1...1000] of книга;
```

Здесь *каталог* определяет таблицу из трех столбцов. Ее значением может быть такая, скажем, таблица:

КАТАЛОГ

Номер	Название	Автор
11275	Теремок	С. Маршак
27343	Рассказы	В. Шукшин
13711	Избранное	И. Бунин

Каждая книга в каталоге однозначно определяется своим ключом — полем *номер*. В примере используется пятизначное значение поля *номер*, такими номерами могут быть поименованы 100 000 книг. Но реально в каталоге никогда не бывает, скажем, более 1000 книг (книги стареют, выбывают, новые книги получают новые, ранее не использованные, номера). Поэтому для представления каталога в программе достаточно иметь таблицу на 1000 элементов.

Для того чтобы можно было работать с таблицей, необходимо реализовать метод доступа к элементам таблицы, т. е. реализовать три процедуры: поместить новый элемент в таблицу, найти и считать нужный элемент из таблицы, уничтожить ненужный элемент. Нужный и ненужный элементы указываются значением поля *номер*. Рассмотрим алгоритмы реализации

Таблица 4.1

Номер строки	h	t	Элемент	
1				
2	002		00 002	Д
275	275		11 275	А
276	276		71 276	Е
277				

процедур на примерах, они усложняются по мере усложнения примеров.

Алгоритм внесения нового элемента. Элементы таблицы будут изображаться для краткости так: (11275, А), (27343, В), (13711, С), где символы А, В, С обозначают значения полей *название* и *автор*. На элементах таблицы определяется целочисленная функция перемешивания h. Функция h принимает значения из множества {1, 2, 3, ..., 1000}, здесь 1000 — размер таблицы. В нашем случае, к примеру, в качестве значения функции перемешивания на элементе таблицы можно взять число, состоящее из последних трех цифр значения индекса: $h((11275, А)) = 275$.

Пусть в начальный момент таблица пуста и необходимо в нее последовательно записать элементы: (00002, Д), (11275, А), (71276, Е), (91275, Б), (66277, К), (54275, Н), (49277, М). Так как $h(0002, Д) = 2$, $h(11275, А) = 275$, то элемент (0002, Д) попадет во вторую строку таблицы, элемент (11275, А) — в 275-ю (см. табл. 4.1). В табл. 4.1 появились две дополнительные колонки, необходимые для реализации метода

Таблица 4.2

Номер строки	h	t	Элемент	
1				
2	002		00 002	Д
275	275	277	11 275	А
276	276		71 276	Е
277	275		91 275	Б
278				
		⋮		

доступа. Для следующего элемента h (91275, Б)$=$ $=275$, но 275-я строка уже занята. Начинается поиск свободной строки, для этого просматриваются последующие строки таблицы. Свободной считается строка, в которой поле $h=0$. Строка 276-я занята (в поле непустое значение), а вот следующая — 277-я строка свободна. В нее помещается элемент (91275, Б), а в строку 275 (в ней находится последним поступивший элемент, на котором функция h равна 275), в столбце t появляется ссылка на 277 строку (табл. 4.2). Эта ссылка означает, что в табл. 4.2 есть еще элементы, для которых значение функции h равно 275, и один из них расположен в строке 277. Таким образом, в 277-й строке появился элемент, место которому по значению функции h должно быть в 275-й строке. Следовательно, элемент (66277, К) негде разместить. Поиск свободного места начинается со строки 277, просматриваются последующие строки, пока не будет найдена свободная, либо будет просмотрена вся таблица и в этом случае размещение нового элемента

Таблица 4.3

Номер строки	h	t	Элемент	
1				
2	002		00 002	Д
		⋮		
275	275	277	11 275	А
276	276		71 276	Е
277	275		91 275	Б
278	277		66 277	К
279				
		⋮		

невозможно. При поиске считается, что вслед за последней 1000-й строкой следует 1-я. В примере элемент (66277, К) будет помещен в 278-ю строку (табл. 4.3).

Следующий элемент (54275, Н) размещается так. Функция h (54275, Н) = 275. В строке 275-й хранится элемент, на котором функция $h = 275$ и в столбце t стоит значение 277 (ссылка на место размещения другого элемента с $h = 275$, далее по ссылке идем в строку 277, где также находится элемент со значением $h = 275$). В этой строке $t = 0$, значит, других элементов с $h = 275$ в табл. 4.2 более нет и далее последовательным просмотром ищется свободное место для элемента (54275, Н). В примере он попадает в строку 279 и в строке 277 полагается $t = 279$ (табл. 4.4).

Разместить в табл. 4.2 элемент (99277, М) по описанному ранее алгоритму правильно не удастся. Поэтому алгоритм поиска места для элемента придется вновь усложнять. В данном случае h(99277, М) = 277,

Таблица 4.4

Номер строки	h	t	Элемент	
1				
2	002		00 002	Д
		⋮		
275	275	277	11 275	А
276	276		71 276	Е
277	275	279	91 275	Б
278	277		66 277	К
279	275		54 275	Н
		⋮		

но 277-я строка занята элементом (21275, Б). Кроме поиска свободного места теперь еще контролируется, не встретится ли строка с элементом, на котором функция h также равна 277. В нашем случае такой элемент есть в строке 278 (вошли в цепочку ссылок). Далее, если есть цепочка ссылок ($t \neq 0$), то по цепочке ссылок добираемся до последнего ее элемента и затем ищем свободную строку последовательным просмотром таблицы. В рассматриваемом случае элемент (99277, М) попадет в строку 280, а в строке 278 поле $t = 280$ (ссылка на следующий элемент таблицы такой, что на нем $h = 277$ (табл. 4.5).

В общем виде теперь алгоритм внесения нового элемента выглядит так:

1) вычисляется значение функции h, пусть $h = k$;

2) начиная с k-й строки таблицы ищется последовательным просмотром свободная строка либо вход в цепочку элементов таких, что $h = k$;

Таблица 4.5

Номер строки	h	t	Элемент	
1				
2	002		00 002	Д
		⋮		
275	275	277	11 275	А
276	276		71 276	Е
277	275	279	91 275	Б
278	277	280	66 277	К
279	275		54 275	Н
280	277		99 277	М
		⋮		

3) если найдена свободная строка, значит, элементов с $h = k$ в таблице нет, и новый элемент помещается в свободную строку. Конец;

4) если найден вход в цепочку, по ней продвигаются до последней строки цепочки, далее ищется свободная строка и в нее помещается новый элемент. Конец;

5) если свободная строка не найдена — новый элемент не может быть помещен в таблицу. Конец.

Алгоритм поиска нужного элемента. Пусть необходимо найти элемент с номером 99277. Этот алгоритм теперь совсем просто изложить:

1) вычисляется значение $h = 277$;

2) начиная со строки 277 ищется последовательным просмотром вход в цепочку ссылок, либо свободная строка;

3) если встретилась свободная строка, значит, эле-

Таблица 4.6

Номер строки	h	t	Элемент	
1				
2				
		. . .		
275	275	279	11 275	А
276	276		71 276	Е
277	—1	279	91 275	Б
278	277	280	66 277	К
279	275		54 275	Н
280	277		99 277	М
		. . .		

ментов таких, что на них $h = 277$, в таблице нет вовсе. Конец;

4) найден вход в цепочку, значения поля *номер* элементов цепочки последовательно сравниваются с 99277. Если сравнение произошло, нужный элемент найден (в нашем случае это элемент (99277, М) в строке 280), в противном случае такого элемента в таблице нет. Конец.

В этом алгоритме для доказательства отсутствия нужного элемента в таблице, как правило, не приходится просматривать всю таблицу.

Алгоритм уничтожения ненужного элемента. Пусть из табл. 4.5 необходимо удалить элемент (91275, Б). Алгоритм удаления следующий:

1) ищется элемент (91275, Б), если бы он не был найден, алгоритм удаления на этом бы нормально завершился. В нашем случае нужный элемент найден в 277-й строке;

Таблица 4.7

Номер строки	h	t	Элемент	
1				
2	002		00002	Д
		⋮		
275	275	279	91275	Б
276	276		71276	Е
277	—1	279	91275	Б
278	277	280	66277	К
279	275		54275	Н
280	277		99277	М
		⋮		

2) для уничтожения элемента выполняются действия: а) ссылка $t = 279$ из строки 277 переносится в 275-ю строку, т. е. после первого элемента цепочки теперь будет следовать сразу бывший третий, второй выпал, б) в поле h в 277-ю строку заносится значение $h = -1$ (табл. 4.6), значение $h = -1$ означает, что в строке раньше был элемент, а теперь он уничтожен (см. табл. 4.6). Полагать $h = 0$ нельзя, так как это такой признак свободной строки, который означает, что далее элементов с $h = 277$ в таблице нет, в результате будет потерян вход в цепочку всех элементов с $h = 277$.

Пусть из табл. 4.5 необходимо удалить элемент (11275, А), это первый элемент цепочки элементов, для которых $h = 275$. Удаление производится так:

а) ищется вход в цепочку, это 275-я строка;

б) выбирается второй элемент цепочки (в настоящем случае это элемент (91275, Б) из 277-й строки)

и записывается в 275-ю строку на место первого. В 277-й строке поле $h = -1$, т. е. уничтожается элемент в 277-й строке (табл. 4.7). Если уничтожается последний элемент цепочки, то в предшествующем элементе цепочки поле $t = 0$.

С появлением в таблице свободных строк с $h = -1$ необходимо решать проблему их использования. Они должны учитываться при поиске свободной строки в алгоритме записи в таблицу нового элемента. Возможны и другие оптимизирующие модификации описанных алгоритмов, более того, их очень много. Предлагаем читателю построить несколько таких модификаций для лучшего освоения алгоритмов.

Куда пойти в кино?

С помощью отношений могут быть представлены в ЭВМ самые разнообразные объекты реального мира. Как это делается рассмотрим на простом примере. Предположим, нужно разработать информационную систему, в которую были бы заложены знания о всех кинотеатрах города и фильмах, в них идущих. Терминалы системы можно было бы установить в кассовых залах каждого кинотеатра так, чтобы любой посетитель смог сам узнать, какие кинофильмы, в каком кинотеатре, на каком сеансе демонстрируются. Для решения этой задачи прежде всего необходимо описать в ЭВМ объекты реального мира.

Вначале определяется отношение *кинотеатр*:

кинотеатр = (*название-кинотеатра*, *адрес*, *Ф. И. О. директора*, *телефон*, *число-рядов*, *количество-мест-в-ряде*).

Значение отношения *кинотеатр*, например, такое:

КИНОТЕАТР

Название-кинотеатра	*Адрес*	*Ф. И. О. директора*	*Телефон*	*Число-рядов*	*Количество-мест-в-ряду*
Маяк	Русская, 1а	Иванов И. И.	35-36-87	30	35
Аврора	пр. К. Маркса, 47	Петров П. П.	46-19-02	40	30
Победа	Ленина, 7	Сидоров С. С.	22-68-80	45	28
Пионер	М. Горького, 52	Васильев В. В.	22-22-92	26	20

Предполагается, что в городе нет двух кинотеатров с одинаковым названием и в каждом кинотеатре лишь один зрительный зал.

Следующее необходимое отношение — *кинофильм* — описывается так:

кинофильм — (*название-фильма*, *жанр*, *длительность-демонстрации*, *режиссер*, *год-выпуска*, *студия*).

Одно из допустимых значений этого отношения следующее:

КИНОФИЛЬМ

Название-фильма	*Жанр*	*Длительность-демонстрации*	*Режиссер*	*Год-выпуска*	*Студия*
Бриллиантовая рука	Комедия	1.40	Л. Гайдай	1965	Мосфильм
Рысь выходит на тропу	Детский	1.20	А. Бабаян	1983	Центрнаучфильм
Грачи	Детектив	1.30	К. Ершов	1983	им. Довженко
Жертва коррупции	Политический детектив	1.15	А. Бонпо	1980	Ле фильм де ля Друетт
Обыкновенное чудо	Сказка	3.10	М. Захаров	1984	Мосфильм

Здесь предполагается для упрощения, что каждый фильм выпускается одним режиссером на одной студии.

Следующее отношение — *сеанс* — связывает отношения (понятия) *кинотеатр* и *кинофильм*:

сеанс = (*номер-сеанса*, *название-кинотеатра*, *название-кинофильма*, *дата*, *время-начала*)

Предполагается, что все сеансы во всех кинотеатрах города во все времена перенумерованы и каждый сеанс имеет уникальный номер. Таким образом, поле *номер-сеанса* может служить ключом отношения *сеанс*. С этой целью оно и введено в это отношение. Если бы его не было, в качестве ключа отношения пришлось бы использовать поля (*название-кинотеатра*, *название-кинофильма*, *дата*), т. е. почти все поля отношения *сеанс*. Значение отношения *сеанс* может быть сле-

дующим:

СЕАНС

Номер-сеанса	*Название-кинотеатра*	*Название-кинофильма*	*Дата*	*Время-начала*
27543	Аврора	Грачи	25.03.86	18.40
27544	Аврора	Грачи	25.03.86	22.20
27545	Аврора	Бриллиантовая рука	25.03.86	20.30
28001	Маяк	Жертва коррупции	25.03.86	16.00
28002	Маяк	Грачи	14.01.87	18.00
87560	Победа	Рысь выходит на тропу	25.03.86	12.10
35694	Пионер	Рысь выходит на тропу	17.08.87	14.20
70035	Пионер	Бриллиантовая рука	17.08.87	16.00

Следующее необходимое отношение — *билет*:

билет = (*номер-сеанса*, *номер-ряда*, *номер-места-в-ряде*, *признак*)

В поле *признак* может быть одно из двух значений: {занято, свободно}. Это отношение содержит список всех мест во всех кинотеатрах города на все сеансы. Каждое место отмечено, занято оно или свободно, т. е. продан на него билет или нет. Фрагмент отношения *билет* может выглядеть следующим образом:

БИЛЕТ

Номер-сеанса	*Номер-ряда*	*Номер-места-в-ряде*	*Признак*
27543	21	15	Занято
27543	21	16	Занято
27543	21	17	Свободно
28001	17	9	Занято
28001	17	10	Занято
28001	18	3	Свободно
27544	9	18	Свободно
27544	9	19	Свободно
27544	13	7	Занято

Прежде чем приступить к решению задачи, отношения *кинотеатр*, *кинофильм*, *билет* упорядочиваются по значению своих ключей.

Теперь можно начать отвечать на вопросы посетителей кинотеатров.

1. Какие фильмы идут в кинотеатрах города 25.03.86 г. и в какое время?

Алгоритм поиска ответа на этот вопрос теперь нетрудно сконструировать.

А). Сначала необходимо упорядочить отношение *сеанс* по значениям полей (*дата*, *номер-сеанса*), т. е. два элемента в процессе упорядочивания должны сравниваться сначала по значению поля *дата*, а при несравнении (значения равны) — по значению поля *номер-сеанса*. Таким образом, все элементы отношения *сеанс* с одинаковым значением поля *дата* будут теперь размещаться последовательно.

СЕАНС

Номер-сеанса	*Название-кинотеатра*	*Название-кинофильма*	*Дата*	*Время-начала*
27543	Аврора	Грачи	25.03.86	18.40
27544	Аврора	Грачи	25.03.86	22.20
27545	Аврора	Бриллиантовая рука	25.03.86	20.30
28001	Маяк	Жертва коррупции	25.03.86	16.00
87560	Победа	Рысь выходит на тропу	25.03.86	12.10
28002	Маяк	Грачи	14.01.87	18.00
35694	Пионер	Рысь выходит на тропу	17.08.87	14.20
70035	Пионер	Бриллиантовая рука	17.08.87	16.00

Б. Теперь алгоритмом деления пополам в комбинации с последовательным просмотром отношения выбираются все элементы со значением поля *дата*, равным 25.03.86. Они образуют новое отношение — *ответ на вопрос*. Назовем его *ответ* 1.

ОТВЕТ 1

Номер-сеанса	*Название-кинотеатра*	*Название-кинофильма*	*Дата*	*Время-начала*
27543	Аврора	Грачи	25.03.86	18.40
27544	Аврора	Грачи	25.03.86	22.20
27545	Аврора	Бриллиантовая рука	25.03.86	20.30
28001	Маяк	Жертва коррупции	25.03.86	16.00
87560	Победа	Рысь выходит на тропу	25.03.86	12.10

Далее, посетитель может пожелать познакомиться лишь с сеансами, начинающимися после 16.00 и не позже 22.00. Понятно, что теперь необходимо упорядочить отношение *ответ* 1 по значениям полей (*время-начала*, *номер-сеанса*) и, отобрав необходимые элементы, построить отношение

ОТВЕТ 2

Номер-сеанса	*Название-кинотеатра*	*Название-кинофильма*	*Дата*	*Время-начала*
27543	Аврора	Грачи	25.03.86	18.40
27544	Аврора	Бриллиантовая рука	25.03.86	20.30

Предположим, человек решил посмотреть фильм «Бриллиантовая рука», т. е. выбран элемент (27544, Аврора, Бриллиантовая рука, 25.03.86, 20.30). Теперь следует узнать, а есть ли свободные места на сеанс 27544? Для этого в отношении *билет* выбираются все элементы со значением поля *номер-сеанса*, равным 27544, и формируется отношение:

БИЛЕТ 1

Нотер-сеанса	*Номер-ряда*	*Номер-места-в-ряде*	*Признак*
27544	9	18	Свободно
27544	9	19	Свободно
27544	13	7	Занято

Следующий вопрос может быть таким: а есть ли на этот сеанс два свободных места рядом в одном из рядов в пределах с 5 по 15? В качестве ответа формируется отношение:

БИЛЕТ 2

Номер-сеанса	*Номер-ряда*	*Номер-места-в-ряде*	*Признак*
27544	9	18	Свободно
27544	9	19	Свободно

Если посетитель скажет, что он покупает билеты на эти места, значение поля *признак* этих элементов отношения *билет* 2 становится равным «Занято».

Если человек не знает, где расположен кинотеатр Аврора, можно запросить его описание, и в отношении *кинотеатр* будет найден элемент (Аврора, пр. К. Маркса 47, Петров П. П., 46—19—02, 40, 30), содержащий адрес кинотеатра. Можно запросить и дополнительные сведения о кинофильме, и в отношении *кинофильм* будет найден элемент (Бриллиантовая рука, комедия, 1.40, Л. Гайдай, 1965, Мосфильм).

А вот получить справку о том, как добраться до кинотеатра Аврора, в нашей информационной модели нельзя. И добавить в отношение *кинотеатр* эти данные тоже нельзя. Дело в том, что размер записи по определению записи фиксирован, не может изменяться. И так как к разным кинотеатрам ведет разное количество маршрутов (к одному, например, пять, а к другому десять), то и добавить в отношение *кинотеатр* новые поля с описанием маршрутов невозможно. Ведь даже если в отношение *кинотеатр* будут введены дополнительно двадцать полей для описания транспортных маршрутов, ведущих к кинотеатру (для Авроры, к примеру, пять таких полей описывали бы маршруты, а остальные были резервными), все равно ведь нашелся бы со временем такой кинотеатр, к которому ведет двадцать один маршрут, а значит, не все они были бы описаны в нашей информационной модели и она, таким образом, не соответствовала бы действительности. С другой стороны, заводить много резервных полей весьма и весьма накладно. Если резервное поле не имеет значения, место в памяти ЭВМ ему надо отводить все равно. Тогда нужно новое отношение

кинотеатр-маршрут = (*название-кинотеатра*, *тип-транспорта*, *номер-маршрута*, *название-остановки*).

Фрагмент этого отношения имеет вид:

КИНОТЕАТР-МАРШРУТ

Название-кинотеатра	*Тип-транспорта*	*Номер маршрута*	*Название-остановки*
Аврора	автобус	20	ст. м. Студенческая
Аврора	автобус	47	ст. м. Студенческая
Аврора	троллейбус	6	ст. м. Студенческая
Аврора	трамвай	18	Студенческая
Аврора	трамвай	13	Студенческая
Победа	троллейбус	1	кинотеатр Победа

Аналогично имеет смысл завести и другие подобные отношения, например, *кинотеатр-кинофильм*, *кинорежиссер-кинофильм*. Их структуру разработайте сами в качестве упражнения. Имеет смысл также обдумать дельнейшее развитие модели с целью расширения круга вопросов, на которые система сможет дать квалифицированный ответ.

Конечно, для общения с системой необходим какой-то язык, на котором можно было бы формулировать вопросы (запросы к системе). Не обсуждая проблемы разработки таких языков, приведем пару примеров формализованных запросов.

1. *Ответ* 3 = (*сеанс. название-кинофильма* = Бриллиантовая рука $\wedge$ *сеанс.дата* = 25.03.86).

В соответствии с запросом требуется в отношении *сеанс* найти все элементы со значением поля *название-кинофильма* = Бриллиантовая рука. Знак $\wedge$ (логическое И) требует, чтобы отбирались лишь те элементы, у которых значение поля *сеанс.дата* = 25.03.86. Эти элементы образуют отношение *ответ* 3, которое будет содержать лишь те элементы отношения *сеанс*, которые описывают демонстрации фильма «Бриллиантовая рука» 25 марта 1986 г.

2. *Ответ* 4 = (*ответ* 3. *время-начала* $\geqslant$ 16.00 $\wedge$ *ответ* 3. *время начала* $\leqslant$ 22.00).

Понятно, что требуется сформировать отношение *ответ* 4, состоящее из тех элементов отношения *ответ* 3, в которых описываются демонстрации фильма «Бриллиантовая рука» 25 марта 1986 г. в любом из кинотеатров города, начинающиеся после 16.00 и не позже 22.00.

● Закрывая книгу

В книге собраны очень и очень коротенькие рассказы об очень и очень немногих вопросах современного программирования и суперЭВМ. Конечно, эта тема неисчерпаема, в программировании всегда есть много нового, удивительного и необычного. Ну разве не интересно было бы узнать, как устроены нейронные ЭВМ, организация которых основана примерно на тех же принципах, что и работа человеческого мозга? Исследования по созданию нейронных ЭВМ резко интенсифицировалась за последние три года. Или понять, как думает ЭВМ, как она решает задачи. Всякий школьник, не только двоечник, мог бы с пользой для себя пообщаться с ЭВМ, которая не только сама легко решает любую геометрическую задачу, но и может задать наводящие вопросы, подсказать некоторые решения, рассказывает о наилучших способах решения. На таких принципах построены экспертные системы, системы проектирования сложных объектов.

Рассмотренные в книге вопросы редко популяризируются, и авторы надеются, что она позволит читателю составить более полное представление о программировании, хотя, конечно же, все равно оно будет фрагментарным.

Авторы будут удовлетворены своей работой не только тогда, когда книга вызовет интерес к проблемам параллельного программирования, но даже и в том случае, если им не удастся окончательно запугать читателя трудностями освоения программирования, тем более, что это и не входило в их намерения.

Знание программирования и основных алгоритмов организации и обработки данных, умение самостоятельно решать задачи с использованием различных ЭВМ,— это вторая грамотность, по выражению академика А. П. Ершова: она становится ныне таким же необходимым навыком культурного человека, как и чтение, письмо.

Авторам остается лишь пожелать читателям быстрее и лучше овладеть искусством программирования.

● Словарь терминов

Автокод — язык низкого уровня для программирования в машинных командах

Адрес ячейки — имя ячейки

Ассемблер — то же, что автокод

Блокировка (взаимная) — ситуация, когда два или более процесса захватили лишь часть необходимых им для продолжения выполнения ресурсов ЭВМ и остановились, так как каждый не имеет всех нужных ресурсов, чтобы дать возможность закончиться хотя бы одному из них

Ввод данных — передача данных из внешней памяти в оперативную

Внешняя память — медленная память, обычно размещается на магнитных дисках и лентах (внешних устройствах ЭВМ), данные из внешней памяти становятся доступными процессору после их переписывания в оперативную память

Вывод данных — передача данных из оперативной памяти во внешнюю либо на печать

Задача реального времени — задача, решение которой бессмысленно, если оно получено по истечении определенного для этой задачи времени

Канал — устройство для передачи данных между различными компонентами ЭВМ

Конкатенация — соединение элементов

Критический интервал — фрагмент параллельной программы, в котором может находиться не более одного процесса.

Магазин — очередь с дисциплиной обслуживания, при которой элемент, первым попавший в очередь, будет обслужен последним

Машинная команда — команда, которую исполняет процессор

Мультипрограммирование — режим организации работы ЭВМ, при котором в ней одновременно может исполняться несколько задач

Непроцедурные операторы языка — операторы, не задающие порядка выполнения

Однородная вычислительная система — параллельная ЭВМ с однотипными процессорами

Оперативная память (ОП) — быстрая память, непосредственно доступная процессору

Операционная система — специальная программа, управляющая всеми вычислительными процессами в ЭВМ

Память — устройство ЭВМ, предназначенное для запоминания данных, обычно это множество ячеек

Параллельная ЭВМ (многопроцессорный вычислительный комплекс) — ЭВМ, имеющая более одного процессора

Последовательная ЭВМ — ЭВМ, имеющая единственный процессор.

Предикат — логическое выражение

Производительность ЭВМ — количество операций, выполняемых ЭВМ в течение одной секунды

Процесс — исполняющаяся программа (фрагмент параллельной программы)

Процессор — устройство для выполнения команд ЭВМ

Разделение ресурсов — совместное, поочередное использование ресурсов

Семафор — специальное устройство в ЭВМ и соответствующая конструкция в языке параллельного программирования, обеспечивающие доступ к разделяемому ресурсу только одному процессу

Синхронизация — согласование во времени выполнения вычислительных операций

Специализированные ЭВМ — ЭВМ, ориентированные на эффективное решение узкого класса задач

Спусковая функция — логическая функция, истинное значение которой разрешает процессу начать выполняться (но не приказывает сделать это!)

Стэк — то же, что магазин

Сумматор — устройство процессора для выполнения операции сложения

Такт реального времени — время от момента ввода исходных данных до момента получения решения

Теги — дополнительные разряды ячейки для определения типа данных, программисту не доступны

Тип данных — в языках программирования обычно бывают типы **integer**, **real**, **integer**, **array**, **real array** и т. п. Например, если теги указывают, что в ячейке находится целое число, то команда «+», которая должна сложить два вещественных числа, не сможет использовать значение этой ячейки (будет прервано ее выполнение). Заметим, что в ЭВМ сложение вещественных и сложение целых чисел выполняются разными командами, хотя в языках программирования сложение обозначается одним и тем же символом «+»

Умножитель — устройство процессора для выполнения операции умножения

Устройство управления (УУ) — устройство ЭВМ, управляющее работой всех остальных ее устройств

Цикл памяти — время, требуемое для записи в память некоторого значения и считывания его из памяти

Ячейка — элемент памяти, хранит обычно одно значение

● Список сокращений

ВП — внешняя память
ВУ — внешнее устройство (диски, магнитные ленты, дисплеи, принтеры и т. д.)
К — число, равное 1024; запись вида 8К означает число $8 \times 1024 = 8192$, читается: Ка либо Кило, например, 8К байтов — восемь Килобайтов
М — число $К \times К = 1024^2$; читается: эМ, либо Мега, например, 5М байтов — пять Мегабайтов
МВК — многопроцессорный вычислительный комплекс
МКМД — множественный поток команд — множественный поток данных
мкс — микросекунда, 10^{-6} с
МКОД — множественный поток команд — одиночный поток данных
нс — наносекунда, 10^{-9} с
ОВС — однородная вычислительная система
ОКМД — одиночный поток команд — множественный поток данных
ОКОД — одиночный поток команд — одиночный поток данных
ОП — оперативная память
опер/с — операция в секунду
ПМП — память микропрограмм
Пр — процессор
УУ — устройство управления процессором
I/O — ввод/вывод (Input/Output)

Оглавление

Научно популярное издание

Вальковский Владимир Александрович
Малышкин Виктор Эммануилович

ЭЛЕМЕНТЫ СОВРЕМЕННОГО ПРОГРАММИРОВАНИЯ И СУПЕРЭВМ

Редактор издательства
Л. В. Попкина
Художник
И. Б. Елистратов
Художественный редактор
С. В. Марковская
Технический редактор
Г. Я. Герасимчук
Корректоры
Л. Л. Михайлова, И. А. Абрамова

ИБ 34701

Сдано в набор 29.04.90 Подписано к печати 04.12.90. Формат 84×108¹/₃₂. Бумага книжно-журнальная. Обыкновенная гарнитура. Высокая печать. Усл. печ. л. 7,6. Усл. кр.-отт. 8. Уч.-изд. л. 8,6. Тираж 40 000 экз. Заказ № 175. Цена 90 коп.

Ордена Трудового Красного Знамени издательство «Наука», Сибирское отделение. 630099 Новосибирск, ул. Советская, 18.

4-я типография издательства «Наука». 630077 Новосибирск, ул. Станиславского, 25.

СИБИРСКОЕ ОТДЕЛЕНИЕ
ИЗДАТЕЛЬСТВА «НАУКА»

готовит к выпуску в 1991 году
следующую книгу:
ИЛЬИН В. П. ВЫЧИСЛИТЕЛЬНАЯ
ИНФОРМАТИКА:
ОТКРЫТИЕ НАУКИ

Книга знакомит читателя с феноменами компьютеризации человеческой деятельности. Рассказывается о становлении новой науки, возникшей на стыке вычислительной техники, программирования и вычислительной математики, о задачах и принципах математического моделирования процессов и явлений. Описывается динамика развития и интеллектуализации современных ЭВМ. Излагаются основы программного обеспечения, отображения алгоритмов на архитектуру ЭВМ, методология вычислительного эксперимента.

Для широкого круга читателей.

В Аннотированном тематическом плане выпуска литературы на 1990 г. (Научно-популярные и серийные издания) книга В. П. Ильина занимает позицию № 62.