# СТРУКТУРЫ ЭВМ С МНОЖЕСТВЕННЫМ ПОТОКОМ КОМАНД

В соответствии с классификацией Флинна (см. § 1.2) к *многопроцессорным ЭВМ* относятся ЭВМ с множественным потоком команд и множественным потоком данных (МКМД-ЭВМ). Основы управления вычислительных процессом в таких ЭВМ изложены в [4, 12, 13], элементы теории управления — в [14].

В основе МКМД-ЭВМ лежит традиционная последовательная организация программы, расширенная добавлением специальных средств для указания независимых, параллельно исполняемых фрагментов. Такими средствами могут быть *операторы* FORK и JOIN, скобки parbegin ... parend, оператор DO FOR ALL и др. Параллельно-последовательная программа достаточно привычна пользователю и позволяет относительно просто собирать параллельную программу из обычных последовательных программ. МКМД-ЭВМ имеет две разновидности: ЭВМ с общей и индивидуальной памятью. Структура этих ЭВМ представлена на рис. 3.1.

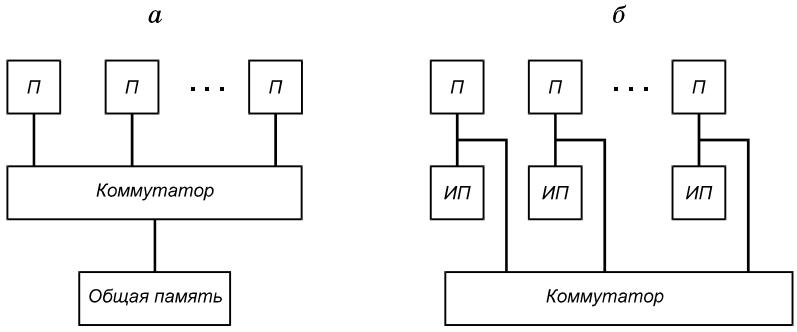



Рис. 3.1. Структура многопроцессорной ЭВМ с общей (*а*) и индивидуальной (*б*) памятью. Здесь:
П — процессор, ИП — индивидуальная память

В качестве коммутатора на рис. 3.1 может использоваться любой коммутатор из описанных в § 2.3. Общая память для повышения пропускной способности обычно содержит более одного блока.

Главное различие между МКМД-ЭВМ с общей и индивидуальной (локальной, распределенной) памятью состоит в характере адресной системы. В машинах с *общей памятью* адресное пространство всех процессоров является единым, следовательно, если в программах нескольких процессоров встречается одна и та же переменная $X$, то эти процессоры будут обращаться в одну и ту же физическую ячейку общей памяти. Это вызывает как положительные, так и отрицательные последствия:

1. Наличие общей памяти не требует физического перемещения данных между взаимодействующими программами, которые параллельно выполняются в разных процессорах. Это упрощает программирование и исключает затраты времени на межпроцессорный обмен.

2. Несколько процессоров могут одновременно обращаться к общим данным и это может привести к получению неверных результатов. Чтобы исключить такие ситуации, необходимо ввести систему синхронизации параллельных процессов, что усложняет механизмы операционной системы.

3. Поскольку при выполнении каждой команды каждым процессором необходимо обращаться в общую память, то требования к пропускной способности коммутатора этой памяти чрезвычайно высоки, что и ограничивает число процессоров в системах с общей памятью величиной 10...20.

В системах с индивидуальной памятью каждый процессор имеет независимое адресное пространство и наличие одной и той же переменной $X$ в программах разных процессоров приводит к обращению в физически разные ячейки индивидуальной памяти этих процессоров. Это приводит к необходимости физического перемещения данных между взаимодействующими программами в разных процессорах, однако, поскольку основная часть обращений производится каждым процессором в собственную память, то требования к коммутатору ослабляются и число процессоров в системах с распределенной памятью и коммутатором типа гиперкуб может достигать нескольких десятков и даже сотен.

В настоящем параграфе будет рассмотрена система управления вычислительным процессом в МКМД-ЭВМ с общей памятью, а в § 3.2 — управление вычислениями в МКМД-ЭВМ с индивидуальной памятью в каждом процессоре.

**Типы процессов**. Для описания параллелизма независимых ветвей в программах для МКМД-ЭВМ, написанных на ЯВУ, обычно используют операторы FORK, JOIN или аналогичные им по функциям.

Пример программы, использующей эти операторы, представлен в § 1.1.

Введем понятия *процесса*, *ресурса* и *синхронизации*, которые понадобятся далее. А.Шоу [12] определяет процесс следующим образом: “процесс есть работа, производимая последовательным процессором при выполнении программы с ее данными”. Это означает, что процесс является результатом взаимодействия элементов пары <процессор, программа>. Одна программа может порождать несколько процессов, каждый из которых работает над своим набором данных. Процесс потребляет ресурсы и поэтому является единицей планирования в многопроцессорной системе.

Ресурс является средством, необходимым для развертывания процесса. Ресурсом может быть как аппаратура (процессоры, оперативная память, каналы ввода-вывода, периферийные устройства), так и общие программы и переменные. В мультипрограммных и мультипроцессорных ЭВМ наиболее употребительными общими переменными являются системные таблицы, описывающие состояние системы и процессов, например, таблицы очередей к процессорам, каналам, ВнУ; таблицы распределения памяти, состояния семафоров и др.

Синхронизация есть средство обеспечения временной упорядоченности исполнения взаимодействующих процессов.

Существует два основных вида зависимости процессов: зависимость по данным и зависимость по ресурсам.

В общем случае зависимость по данным имеет место между множеством процессов. Пусть два множества процессов связаны таким образом, что каждый процесс из множества $\{L\}$ получает данные от всех процессов множества $\{M\}$. Пример информационного графа такой зависимости представлен на рис. 3.2. Тогда условие запуска множества процессов для этого примера можно записать так:

**if**(JOIN1 $\wedge$ JOIN2 $\wedge$ JOIN3) **then** FORK L1, L2

то есть запуск процессов осуществляется по логической схеме “И” после выполнения всех операторов JOIN.

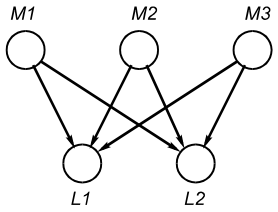


Рис. 3.2. Взаимодействие синхронных процессов

Будем называть такие процессы *синхронными*. Отличительной особенностью синхронных процессов является наличие связного ИГ и вызванный этим обстоятельством запуск по схеме “И”. Иногда такие процессы именуют сильносвязанными.

Другой разновидностью процессов являются *асинхроннные.* Такие процессы не имеют связного ИГ, поэтому иногда называются слабосвязанными (через ресурс). Взаимодействие между ними осуществляется в ситуации, когда несколько таких процессов на основе конкуренции пытаются в одно и то же время захватить единственный ресурс. Это могут быть задачи независимых пользователей, которые одновременно обращаются к каналам ВнУ, требуют объемов физической памяти. Асинхронными являются все системные процессы, работающие с общими таблицами очередей к ресурсам, состояний процессов.

Синхронные процессы, например, *М*1, *М*2, *М*3 на рис. 3.2 могут выполняться двояко: либо им во время запуска отводятся собственные ресурсы (место в памяти, процессор и т. д.) и тогда они независимо выполняются на своих “площадях”; либо эти процессы запускаются как асинхронные на одном и том же наборе ресурсов, конкурируя за их захват.

Поскольку асинхронные процессы пытаются одновременно захватить единственный ресурс, то задача синхронизации в этом случае состоит в том, чтобы обеспечить последовательное во времени использование ресурса процессами, то есть выполнить задачу взаимоисключения процессов. Это соответствует запуску процессов по логической схеме “Исключающее ИЛИ”.

Рассмотрим некоторые методы и средства синхронизации в МКМД-ЭВМ. Сначала коснемся вопроса о средствах, реализующих методы синхронизации. Все процессы между точками синхронизации выполняют обычные команды однопроцессорной ЭВМ. Следовательно, средством выполнения процессов между точками синхронизации является аппаратура процессора. Опера

ции же синхронизации (FORK, JOIN, взаимное исключение) из-за большого объема выполняемой ими работы не могут быть реализованы только в виде аппаратных команд и выполняются также как подпрограммы операционной системы, в совокупности составляющие раздел ОС “Управление процессами и ресурсами”.

**Взаимодействие асинхронных процессов на основе семафоров**. Рассмотрим следующий пример [12]. Пусть имеется система с общей памятью и двумя процессорами, каждый с одним регистром.

В первом процессоре выполняется процесс $L1$, во втором — $L2$:

$L$ 1: ... $X$: = $X$ + 1; ...
$L$ 2: ... $X$: = $X$ + 1; ...

Процессы выполняются асинхронно, используя общую переменную $X$. При выполнении процессов возможно различное их взаиморасположение во времени, например, возможны следующие две ситуации:

$L1$: $R1 := X$; $R1 := R1 + 1$ ; $X := R1$; ... (3.1)
$L2$: ..... $R2 :=$ X ; $R2 := R2 + 1$; $X := R2$;

$L1$: $R1 := X$; $R1 := R1 + 1$; $X := R1$; ... (3.2)
$L2$: ..... $R2 := X$; $R2 := R2 + 1$; $X := R2$;

Пусть в начальный момент $X = V$. Нетрудно видеть, что в (3.1) второй процессор производит чтение $X$ до завершения всех операций в первом процессоре, поэтому $X = V + 1$. В случае (3.2) второй процессор читает $X$ после завершения всех операций первым процессором, поэтому $X = V + 2$. Таким образом, результат зависит от взаиморасположения процессов во времени, что для асинхронных процессов определяется случайным образом. Значит, результат зависит от случая, что недопустимо. Ясно, что должно учитываться каждое приращение $X$.

Выход заключается в разрешении входить в *критическую секцию* (КС) только одному из нескольких асинхронных процессов. Под критической секцией понимается участок процесса, в котором процесс нуждается в ресурсе. При использовании критической

секции возникают различные проблемы, среди которых можно выделить проблемы состязания (гонок) и тупиков. Условие состязания возникает, когда процессы настолько связаны между собой, что порядок их выполнения влияет на результат. Условие тупиков появляется, если два взаимосвязанных процесса блокируют друг друга при обращении к критической секции. Рассмотрим сначала решение проблемы состязаний.

Если бы КС были короткими, например, длительностью в одну команду, тогда можно было решить задачу взаимного исключения аппаратным способом. Для этого надо было бы на время выполнения такой команды запретить обращение в общую память командам от других процессоров. Однако, КС могут иметь произвольную длительность, поэтому необходимо общее решение данной проблемы.

Это решение предложил Дейкстра [12] в виде семафоров. *Семафором* называется переменная $S$, связанная, например, с некоторым ресурсом и принимающая два состояния: 0 (запрещено обращение) и 1 (разрешено обращение). Над $S$ определены две операции: $V$ и $P$. Операция $V$ изменяет значение $S$ семафора на значение $S + 1$. Действие операции $P$ таково: 1) если $S \neq 0$, то $P$ уменьшает значение на единицу; 2) если $S = 0$, то $P$ не изменяет значения $S$ и не завершается до тех пор, пока некоторый другой процесс не изменит значение $S$ с помощью операции $V$. Операции $V$ и $P$ считаются неделимыми, т. е. не могут исполняться одновременно.

Приведем пример синхронизации двух процессов:

```
begin  semaphore S;          S:=1;
       process 1:  begin     L1:P(S);
                             Критический участок 1;
                             V(S);
                             Остаток цикла, go to L1
                   end                                   (3.3)
       process 2:  begin     L2:P(S);
                             Критический участок 2;
                             V(S);
                             Остаток цикла, go to L2
                   end
end
```

В (3.3) операторы *parbegin* и *parend* обрамляют блоки *process* 1 и *process* 2, которые могут выполняться параллельно. Процесс может захватить ресурс только тогда, когда $S$:=1. После захвата процесс закрывает семафор операции $P(S)$ и открывает его вновь после прохождения критической секции $V(S)$.

Таким образом, семафор $S$ обеспечивает неделимость процессов *Li* и, значит, их последовательное выполнение. Это и есть решение задачи взаимного исключения для процессов *Li.*

Семафор обычно реализуется в виде ячейки памяти. Процессы *Li,* чтобы захватить ресурс, вынуждены теперь на конкурентной основе обращаться к ячейке памяти $S$, которая таким образом сама стала ресурсом. Следовательно, обращение к $S$ также является критической секцией каждого *Li.* Однако, эта КС значительно короче и ее можно реализовать в виде пары команд, за которыми закрепилось название *TS* (Test and Set — проверить и установить) и *SI* (Store Immediate — записать немедленно). Функции команд *TS* и *SI* представлены на рис. 3.3. На этом рисунке для семафора используется наименование *mutex* (MUTual EXcluding — взаимное исключение). С целью упрощения выражений в дальнейшем тексте в отличие от семафоров Дийкетры закрытым состоянием *mutex* является 1 (true), открытым — 0 (false). Команды *TS* и *SI* технически реализуются неделимыми.

Установка *mutex* в 1 производится сразу же после чтения *mutex* в регистр *TS*. Это делается для того, чтобы сократить задержку на обращение в память другим процессам во время выпол

нения команды *TS* процессом *Li.* Если дальнейший анализ регистра *TS* покажет, что в момент чтения *mutex* находился в состоянии 1, то это означает, что распределяемый ресурс занят другим процессом и *Li* будет повторно запрашивать *mutex*. Если же окажется, что *mutex* находился в состоянии 0, то процесс *Li* может входить в КС, а предварительная установка *mutex* в 1 запретит другим процессам использовать ресурс до момента его освобождения. Это делает команда *SI,* которая обращается в ячейку *mutex,* минуя все очереди на обращение к памяти.

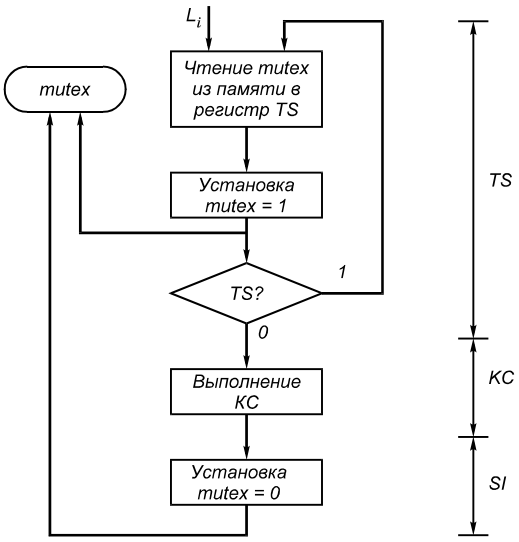



Рис. 3.3. Выполнение команд TS и SI

Взаимодействие двух процессов с целью использования некоторого ресурса можно представить в виде отрезка программы:

*Lj*: **if** *TS*(*mutex*) **then go to** *Lj*;
*Li*: **if** *TS*(*mutex*) **then go to** *Li*; (3.4)
KCi; *SI*;

Процессы *Li* и *Lj* выполняются независимо друг от друга и асинхронно. В (3.4) предполагается, что *Li* первым вошел в свою КС. Процесс *Lj* будет выполнять опрос *mutex,* пока *Li* будет находиться в КС. Это называется "*занятым ожиданием*", во время которого *Lj* непроизводительно затрачивает время процессора и циклы памяти.

Более подробно эта ситуация представлена на рис. 3.4, а для тех же двух процессов, хотя рис. 3.4, как и программа (3.4) могут быть расширены для произвольного числа процессов. На рисунке предполагается, что процессы, выполняясь на разных процессорах, одновременно подошли к своим КС.

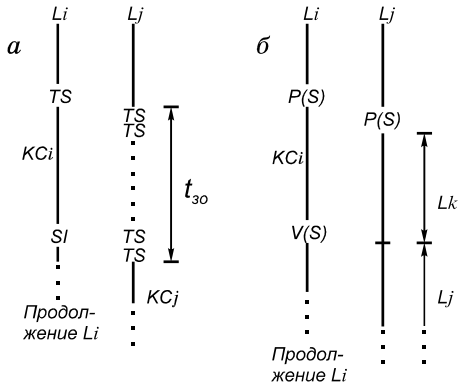



Рис. 3.4. Взаимодействие асинхронных процессов при входе в КС

Поскольку время занятого ожидания $t_{зо}$ может быть произвольно большим, разработаны способы его исключения или уменьшения.

Чтобы устранить занятое ожидание процесса *Lj* или других ожидающих освобождения ресурса *R* процессов, необходимо разработать механизм постановки этих процессов в очередь к ресурсу *R*. Постановка этих процессов в очередь позволит освободить соответствующие процессоры для выполнения других работ до момента освобождения занятого ресурса. С этой целью структура семафора усложняется (рис.3.5).

| mutex | S | Адрес начала очереди процесса |
|---|---|---|

Рис. 3.5. Структура ячейки семафора

В поле *S* хранится семафор *S* ресурса *R*. Над *S* выполняются операции *P*(*S*) и *V*(*S*). Семафор *S* может принимать отрицательные

значения. Абсолютная величина такого значения определяет длину очереди к *R*. Семафор *mutex* обеспечивает неделимость операции постановки процессов в очередь к *R*.

Над семафором *S* возможны следующие операции занятия *P*(*S*) и освобождения *V*(*S*) ресурса *R*:

*P*(*S*): *L*: **if** *TS* (*mutex*) **then go to** *L*; (3.5)
*S*: = *S* - 1
**if** *S*<= - 1 **then begin** поставить *L* в очередь к *R*; *SI*;
**end**;
**else** *SI*.

*V*(*S*): *L*: **if** *TS* (*mutex*) **then go to** *L*; (3.6)
*S*: = *S* + 1
**if** *S* < 0 **then begin** продвинуть очередь к *R*; *SI*; **end**;
**else** *SI*.

На рис. 3.4, б показано поведение двух процессов при одновременном обращении к общему ресурсу *R*. Здесь процесс *Li* захватывает ресурс благодаря тому, что он первым успешно выполняет *TS,* а процесс *Lj* становится в очередь к *R,* при этом выполнявший его процессор переключается на другой процесс *Lk*, устраняя тем самым простой процессора. Таким образом, очередь поглотила “занятое ожидание”.

Особенностью операций (3.5) и (3.6) является то, что они включены во все процессы, использующие ресурс *R,* при этом операции *P*(*S*) и *V*(*S*) выполняются над семафором *S* ресурса, а операции *TS* (*mutex* ) и *Si* (*mutex* ) — над семафором, защищающим операции над *S.* По существу, операции *P*(*S*) и *V*(*S*) сами стали объектом, который обладает всеми признаками ресурса, поскольку эти операции разделяются процессами {*Li*} на основе конкуренции и имеют свой семафор *mutex*, обеспечивающий их неделимость.

На входе операций *P*(*S*) и *V*(*S*) существует занятое ожидание. Если бы эти операции имели большую длительность, имело бы смысл для уменьшения занятого ожидания завести очереди к ним, но обычно обработка общих переменных типа очередей и другого рода таблиц выполняется быстро. Во всяком случае время работы растет не более, чем линейно с ростом объемов таблиц, списков. Поэтому процессу *L* нецелесообразно, обнаружив, что другой процесс захватил их в свое распоряжение, покидать процессор и ста

новиться в очередь к *mutex*. Время переключения процессора на другой процесс может оказаться больше времени исполнения *P*(*S*) или *V*(*S*).

С целью упрощения программирования процессов и уменьшения количества ошибок используются синхропримитивы более высокого уровня, в частности мониторы и почтовые ящики [13]. Монитор представляет собой совокупность управляющих переменных, связанных с некоторыми общими ресурсами, и процедур запросов и освобождения этих ресурсов. Монитор "отторгает" от процессов все их критические секции, связанные с данными ресурсами и превращает их в свои процедуры, которые вызываются указанием имен процедуры и монитора. Они доступны всем процессам в том смысле, что любой из процессов может попытаться вызвать любую процедуру, но только один из процессов может войти в процедуру, остальные должны дожидаться того момента, когда будет завершен предыдущий вызов. Процедуры монитора могут содержать только переменные, локализованные в теле монитора.

Почтовый ящик служит примером синхронизации с помощью сообщений. Если процессу А необходимо взаимодействовать с процессом В, он просит систему образовать почтовый ящик для передачи сообщения. После этого процесс А помещает сообщение в почтовый ящик, откуда оно может быть взято процессом в любое время. С программной точки зрения почтовый ящик — это структура данных, для которой фиксированы правила работы с ней. Она включает заголовок, содержащий описание почтового ящика, и несколько гнезд, в которые помещаются сообщения. Число и размеры гнезд задаются при образовании почтового ящика. Правила работы с ящиком различаются в зависимости от его сложности.

**Управление процессами и ресурсами**. Пусть имеется система типа МКМД с общей памятью. В системе имеется несколько процессоров и несколько классов ресурсов, причем в каждом классе имеется несколько единиц ресурса. К процессорам имеется общая очередь процессов, которой управляет планировщик (диспетчер).

Процессы в этой очереди могут находиться в одном из трех состояний:

- выполняется на некотором процессоре;

• готов к исполнению, но свободный процессор отсутствует;

• блокирован, то есть находится в очереди к одному из ресурсов, а значит, не может выполняться на процессоре.

Рассмотрим, как производится управление процессами и ресурсами в такой системе.

Основной единицей планирования в системе является *задача.* Задача — это независимая единица вычислений, которая обладает собственными ресурсами и представляет пользователя. Описание задачи, содержащее состав требуемых ресурсов, время работы и другие условия выполнения задачи составляются пользователем и входят в состав задачи. Задача состоит из одного или нескольких процессов, находящихся в разной степени подчинения. Каждый процесс характеризуется своим описанием, которое называется *дескриптором.* Дескриптор создается специальной процедурой Create перед выполнением процесса на основе описания задачи и информации, имеющейся в программе пользователя. Дескрипторы хранятся в системной области памяти и используются во время работы процесса. Уничтожение процесса вызывает также и уничтожение дескриптора. Дескриптор содержит следующую информацию:

1. Имя процесса *Li.*

2. Описание ресурсов, занимаемых процессом *Li:* номер процессора, разделы и адреса памяти, информация о других ресурсах.

3. Состояние процесса (выполняется, готов, блокирован).

4. Имя процесса, породившего данный процесс; имена подчиненных процессов, порожденных данным процессом. Процесс *Li* считается выполненным, когда будут выполненными все подчиненные процессы. Для них заводится счетчик, из которого вычитается 1 при выполнении каждого подчиненного процесса. Следовательно, блок 4 дескриптора, называемый *критическим блоком* (не совпадает с критической секцией), выполняет функции операторов FORK и JOIN.

5. Прочие данные: приоритет *Li,* приоритеты подчиненных процессов, предельно допустимое время использования процессора и др.

Дескриптор класса ресурсов создается операционной системой и обычно содержит следующие компоненты:

1. Имя класса ресурса.

2. Состав ресурса, например: число процессоров в системе; число сегментов, страниц основной памяти; число ВнУ одного типа.

3. Таблицу занятости единиц ресурса, которая указывает имена процессов, занимающих каждый ресурс.

4. Описание очереди ждущих процессов, указывает число элементов очереди, адреса начального и конечного элементов очереди.

5. Адрес или имя распределителя, ответственного за порядок занятия единиц ресурса процессами. Распределитель процессора обычно называется планировщиком или диспетчером.

Как уже ранее говорилось, общие переменные также являются разделяемым ресурсом, однако, их описание является ограниченным и состоит из определения семафора типа *mutex*.

Дескрипторы ресурсов хранятся в системной области памяти. Информация, содержащаяся в дескрипторах ресурсов, изменяется при выполнении задачи соответствующими процедурами.

Операции над процессами и ресурсами будем называть *примитивами*. Примитив следует рассматривать как команду, которая выполняется в пределах вызывающего процесса и не является отдельным процессом. Примитивы являются неделимыми, они защищаются семафором типа *mutex*, следовательно в начале и конце такой операции используются команды *TS* и *SI*. Во время выполнения примитива прерывания процессора запрещены, чтобы обеспечить выполнение примитива до освобождения процессора.

При дальнейшем описании примитивов эти защитные операции явно не введены, однако их наличие подразумевается.

Примитивы реализуются программно, аппаратно или смешанным образом. Вызовы примитивов (или команды) размещаются в тексте программы пользователем или включаются в текст программы компьютером на основе информации, размещенной пользователем в исходном тексте задачи.

Основное назначение примитивов — связать задачу с ресурсами системы.

Для управления процессами используются два основных примитива:

1. Create — создать процесс.

2. Destroy — уничтожить один или несколько процессов.

Новый процесс создается задачей или некоторым процессом этой задачи. Создание подчиненного процесса заключается в построении части дескриптора нового процесса. Информация для дескриптора передается от порождающего процесса к порождаемому при выполнении первым примитива Create. Начальные ресурсы памяти $M$ и другие ресурсы $R$, необходимые для начальной стадии функционирования создаваемого процесса $L$, предоставляются из состава ресурсов, принадлежащего порождающему процессу, то есть эти ресурсы являются общими. В дальнейшем порожденный процесс может затребовать собственные ресурсы с помощью операции Request.

Приоритет порождаемого процесса не может превышать приоритет порождающего процесса.

*Procedure Create* ($L$, $E_0$, $M_0$, $R_0$, $P_0$)

**begin**
присвоить процессу имя $Li$
описать начальные ресурсы $R_0$ и требуемую память $M_0$;
задать начальное состояние процесса $E_0$=“готов”; поместить процесс в очередь готовых;
указать имя процесса-отца и процессов-потомков;
задать приоритет процесса $P_0$;
**end**.

Уничтожение процесса $Li$ состоит в возврате всех занятых ресурсов $R$ и областей памяти. При этом изменяются таблицы занятости соответствующих дескрипторов ресурсов. Если процесс находился в состоянии выполнения, он останавливается и вызывается планировщик для реорганизации очереди процессов к процессорам. Наконец, удаляется дескриптор самого уничтожаемого процесса $Li$. Аналогичные действия выполняются для всех подчиненных процессов, порожденных процессом $Li$, если они есть, то есть удаляется все дерево процессов. Операция Destroy ($Li$), где $Li$ имя корневого процесса, выполняется в процессе, породившем $Li$.

Ее можно описать следующей процедурой:

*Procedure Destroy* ($Li$);

**begin**
*sched*: = **false**;
**if** *E* (*Li*) = “выполняется” **then**
**begin** *stop* (*Li*); *sched* = **true**; **end**;
Удалить *Li* из очереди на блокирующий ресурс;
Освободить личные ресусы M и R, включить их в
дескрипторы как свободные;
Удалить из памяти дескриптор процесса *Li*;
**if** *sched* = **true then** вызвать планировщик;
**end**.

Для управления ресурсами используются следующие основные примитивы:

1. Request — затребовать ресурс.
2. Release — освободить ресурс.

Операцию Request можно представить в виде процедуры Request (*R*), где *R* — имя класса требуемых ресурсов.

*Procedure Request* (*R*);

**begin**
Поместить процесс *Li* в очередь к *R*;
Получить результат распределения очереди *R*(*Q*);
**if** $Li \subset Q$ **then** Установить: “*Li* блокирован”
**else** поместить *Li* в очередь готовых;
Выполнить планирование готовых процессов;
**end**.

Согласно процедуре запрос процесса *Li* сначала становится в очередь на ресурс (с предварительным запретом прерываний и выполнением операции *TS*), затем эта очередь обрабатывается распределителем данного ресурса, который вырабатывает список *Q* и количество *k* удовлетворенных запросов. Следовательно, за одно обращение может удовлетворяться более одного запроса. Затем в цикле каждый элемент списка *Li* включается в очередь готовых процессов. Для запрашивающего процесса *M* это не делается, поскольку он уже находится в этой очереди. Если *M* оказался не включенным в список *Li*, он блокируется. Во всех ситуациях включается планировщик для реорганизации очереди.

*Procedure Release* (*R*);

**begin**
Включить освобождаемый ресурс в опись свободных дескриптора ресурса;
Получить результат перераспределения очереди *R*(*K, Q*);
Поместить готовые процессы в очередь готовых;
Выполнить планирование готовых;
**end**.

Чтобы рассмотреть пример функционирования МКМД-системы в общем, введем еще две операции, которые по своим функциям аналогичны действию операторов FORK и JOIN. Сохраним эти названия за соответствующими процедурами. Процедура FORK позволяет запустить одновременно *k* процессов, которые помещены в списке *Q*:

*Procedure Fork* (*k*, *Q*);

**begin**
Создать критический блок в дескрипторе порождающего процесса;
**for** *i* = 1 **to** *k* **do**
**Create** (*Li*, $E_0$, $M_0$, $R_0$, $P_0$);
Установить статус порождающего процесса *E*=“блокирован”;
Произвести планирование очереди процессоров;
**end**.

В процедуре JOIN *Li* — имя процесса-отца, *l* — имя порожденного процесса, а *k* — число подчиненных процесссов. При каждом вызове процедуры выполняются следующие действия:

*Procedure* JOIN (*Li*, *l*, *k*);

**begin**
*k* = *k* - 1;
**if** *k* = 0 **then begin**
Установить *E*(*Li*): = “готов”;

Включить *Li* в очередь готовых; **end**;

Выполнить планирование готовых;

**end**.

На рис. 3.6 приведена диаграмма состояний группы процессов *i*1, *i*2, *i*3, порожденных с помощью примитива FORK основным процессом *i*0, в качестве которого может, например, выступать и задача. Для процессов на рис. 3.6 отмечены 9 точек изменения состояний, а в таблице 3.1 указано состояние процессов и общей очереди к процессорам во всех этих 9 точках. Для рис. 3.6 предполагается, что процессы *i*1 и *i*2 получают ресурсы без блокирования, а процесс *i*3 некоторое время блокирован из-за очереди на ресурсы. Основной процесс *i*0 после выполнения оператора FORK не выполняется до момента, когда сработает критический блок ($k = 0$), но и после этого из-за отсутствия свободного процессора он также блокирован. Одним из первых его действий после получения процессора является уничтожение процессов *i*1, *i*2, *i*3 и возврат ресурсов.

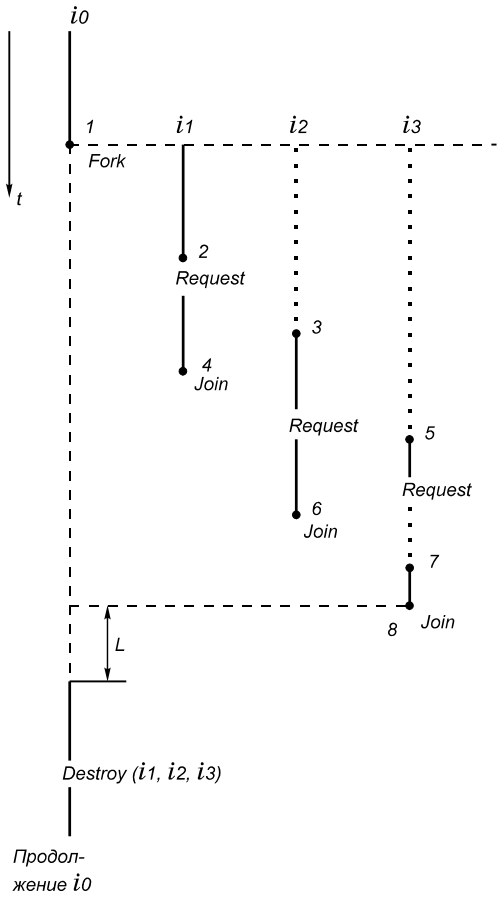



Рис. 3.6. Диаграмма выполнения операций FORK и JOIN

Таблица 3.1.

Состояния процессов и процессоров
при выполнении примера на рис. 3.6

| NN состояния | *P*1 | *P*2 | Очередь к процессорам |
|---|---|---|---|
| 1 | *N* | *i*0 | *i*1, *i*2, *i*3 |
| 2 | *N* | *i*1 | *i*2, *i*3 |
| 3 | *i*2 | *i*1 | *i*3, *K*, *L*, *M* |
| 4 | *i*2 | *i*3 | *N*, *L* |
| 5 | *i*2 | *N* | *L* |
| 6 | *L* | *N* | *M* |
| 7 | *L* | *i*3 | *M* |
| 8 | *L* | *M* | *i*0 |
| 9 | *i*0 | *M* | - |

В большинстве случаев основной функцией порождающего процесса является управление подчиненными процессами.

Отличие указанных выше примитивов Create, Destroy, Request, Release, Fork, Join от операций над семафорами *P*(*S*), *V*(*S*) состоит в том, что в примитивах, привязанных к операционной системе, стандартизован большой объем функций, связанных с управлением процессами и ресурсами, в то время как операции *P*(*S*) и *V*(*S*) являются небольшой частью этих функций, а остальную часть функций должен достраивать пользователь, что приводит к запутанности управления и частым ошибкам.

Примером таких ошибок являются *тупики* (deadlock). Рассмотрим в качестве примера ситуацию, когда два процесса *L*1 и *L*2 обращаются за общим ресурсом, которым является, например, память. Пусть максимальный объем памяти 100 страниц. Тогда представленная ниже ситуация является тупиковой:

| Процесс | Требуемое число страниц | Уже имеется страниц | Дополнительное требование |
|---|---|---|---|
| *L*1 | 80 | 41<br>+ | 39 |

| | | | |
|---|---|---|---|
| *L*2 | 60 | 21 | 39 |

Свободная память 100 - 62 = 38

Таким образом, свободная память 38 страниц меньше запроса от любого из процессов, поэтому ни один из процессов далее не может выполняться. В настоящее время большое внимание уделяется разработке методов формального определения тупиков [14].

Различают *централизованное* и *децентрализованное управление* мультипроцессорной системой. В первом случае вводится специальный управляющий процессор, в котором хранятся все системные таблицы и который выполняет все примитивы и планирование. Другие процессоры обращаются к управляющему с запросами на выполнение системных функций через систему прерываний.

Централизованное управление реализуется достаточно просто, однако при большом потоке запросов от исполнительных процессоров управляющий процессор может не успевать обрабатывать запросы и исполнительные процессоры будут простаивать.

Этого недостатка лишена более сложная система с децентрализованным управлением. При таком управлении каждый исполнительный процессор содержит ядро операционной системы и в состоянии сам выполнять системные примитивы и производить планирование.

Децентрализованное управление обладает еще одним достоинством: при выходе из строя какого-либо из процессоров производится автоматическое перераспределение работ и система продолжает функционирование.

В зависимости от назначения системы возможны различные *алгоритмы планирования*. Различают планирование циклическое, приоритетное, адаптивное. При *циклическом* планировании процесс, использовавший выделенный ему квант времени, помещается в конец очереди на выполнение. Возможно организовать несколько очередей, в каждой из которых величина кванта времени меняется.

При *приоритетном* планировании каждому процессу присваивается приоритет, который определяет его положение в очереди на выполнение. Процесс с высшим приоритетом ставится в на

чало очереди, с низшим — в ее конец. Приоритет может быть статическим и динамическим.

При *адаптивном* планировании предполагается контроль над реальным использованием памяти. Каждому процессу устанавливаются значения объема памяти и виртуального времени процессора, которое обратно пропорционально максимальному объему памяти, необходимому процессу. Процессу выделяется необходимый временной интервал только при наличии достаточного объема свободной памяти. Планирование ориентировано на систему процессов с минимальными запросами, т. е. отдается предпочтение процессам небольшого размера с малым временем выполнения.

Управление процессами и ресурсами для многопроцессорных ЭВМ в значительной степени позаимствовало методы управления из ОС мультипрограммных ЭВМ. Дополнительным свойством МКМД ЭВМ является управление синхронными процессами типа FORK-JOIN.

## § 3.2. Многопроцессорные ЭВМ с индивидуальной памятью

МКМД-ЭВМ с *индивидуальной памятью* получили большое распространение ввиду относительной простоты их архитектуры. В таких ЭВМ межпроцессорный обмен осуществляется с помощью передачи сообщений в отличие от описанных в предыдущем параграфе ЭВМ с общей памятью, где такой обмен отсутствует. Наиболее последовательно идея ЭВМ с индивидуальной памятью отражена в транспьютерах и параллельных системах, построенных на их основе [15, 16].

*Транспьютер* (transputer = transfer (передатчик) + computer (вычислитель)) является элементом построения многопроцессорных систем, выполненном на одном кристалле СБИС (рис. 3.7, а). Он включает средства для выполнения вычислений (центральный процессор, АЛУ для операций с плавающей запятой, внутрикристальную память объемом 2...4 кбайта) и 4 канала для связи с другими транспьютерами и внешними устройствами. Встроенный интерфейс позволяет подключать внешнюю память объемом до 4 Гбайт.

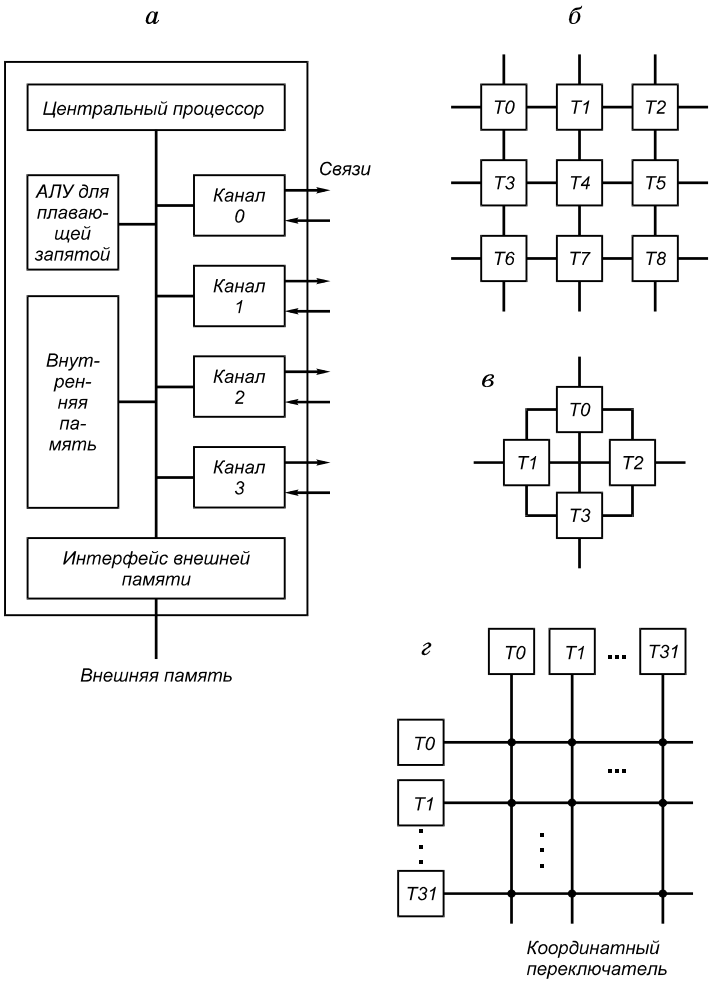



Рис. 3.7.Структура транспьютера и систем на его основе

Для образования транспьютерных систем требуемого размера каналы различных транспьютеров могут соединяться непосредственно (рис. 3.7, б, в) или через коммутаторы типа координатный переключатель на 32 входа и выхода, который обеспечивает одновременно 16 пар связей (рис. 3.7, г). Такие переключатели могут

настраиваться программно или вручную и входят в комплект транспьютерных СБИС.

Размер транспьютерных систем не ограничен, а структура системы может быть сетевой, иерархической или смешанной. Для описания организации и функционирования транспьютера и многопроцессорных систем на его основе необходимо сначала рассмотреть элементы языка Оккам.

**Введение в Оккам.** Организация транспьютеров основана на языке *Оккам* (Occam), разработанном под руководством Хоара (C.A.R.Hoar) в 1984 году. Основой языка являются: средства описания параллелизма выполняемых процессов; средства описания межпроцессорного обмена данными; средства описания размещения процессов по единицам оборудования. Рассмотрим основные конструкции этого языка.

Согласно Хоару [17] Оккам-программа — это процесс. Сложные процессы строятся из процессов-примитивов. Рассмотрим некоторые процессы-композиции в таком порядке: последовательные процессы, параллельные процессы, каналы в Оккаме, циклы, подпрограммы, размещение процессов, примеры вычислительных программ.

Последовательные процессы описываются с помощью конструктора *SEQ*. Чтобы записать последовательный процесс, необходимо записать имя конструктора последовательного процесса *SEQ*, а затем с отступом в две позиции записать его подпроцессы в порядке сверху вниз, например:

```
SEQ
  a: = 1
  b: = a + 7
  c: = b ∗ 12
  d: = ((b + c) ∗ 100)/4
```

Процесс *SEQ* считается законченным, когда последовательно будут выполнены все его подпроцессы.

Последовательный условный конструктор состоит из ключевого слова IF и списка процессов-компонент, у каждого из которых есть свое условие исполнения. Число компонент не ограничено.

Процессы-компоненты записываются под своим условием и представляют собой либо процессы-примитивы, либо процессы-

композиции. При выполнении IF условия просматриваются сверху вниз до тех пор, пока не будет найдено первое условие, имеющее значение “истина”. Затем выполняется процесс, относящийся к этому условию, например:

```
IF
    x < 0
      y: = -1
    x = 0
      y: = 0
    x > 0
      y: = 1
```

Следует отметить, что в конструкторе IF выполняется только один процесс, соответствующий первому истинному условию.

Параллельные процессы формируются с помощью конструктора PAR, например:

```
PAR
    a: = b - 5
    d: = b + 1
```

Здесь порядок выполнения подпроцессов является произвольным. Параллельный процесс считается законченным, когда выполнены все его подпроцессы.

При выполнении параллельных процессов возможны ошибки программирования следующего вида:

```
PAR
    a: = b - 5                                    (3.7)
    b: = a + 2
```

Здесь одновременное выполнение подпроцессов недопустимо, так как они зависят друг от друга (прямая и обратная типы зависимостей, см. § 1.1). Компиляторы должны автоматически выявлять ошибки типа (3.7).

Операции ввода-вывода рассмотрим на следующем примере:

```
SEQ
    c1 ? x
    c2 ! x * x
```

Подпроцессы в этом конструкторе *SEQ* обозначают следующее: сначала через *канал* *c*1 в транспьютер вводится значение, которое будет размещено в ячейке x. Знак “?” обозначает операцию ввода. Второй подпроцесс вычисляет выражение *x***x* и результат выводит из транспьютера через канал *c*2. Знак “!” обозначает операцию вывода.

При программировании ввода-вывода также возможны “тупики”, например:

```
L1              L2
SEQ             SEQ
   c1 ! a          c2 ! x                    (3.7)
   c2 ? b          c1 ? y
```

В этом примере процесс *L*1 не может обрабатываться, пока процесс *L*2 не прочтет из канала *c*1 значение *a*; процесс *L*2 не может этого сделать, пока не выполнит операцию записи значения *y* в канал *c*2. Но прием значения из *c*2 в переменную *b* в процессе *L*1 может произойти только после операции записи *a* в канал *c*1 и т. д.

В языке Оккам имеется два вида циклов: неограниченный цикл последовательного выполнения процессов WHILE и ограниченный цикл или размножитель процессов FORK.

Цикл WHILE заканчивает работу при выполнении некоторого условия, поэтому заранее число итераций этого цикла неизвестно.

Рассмотрим примеры:

```
WHILE TRUE
    SEQ
       c1 ? y                                 (3.8)
       c2 ! x *x
```

Поскольку значение условия не меняется (значение логической переменной постоянно равно TRUE ), то цикл будет бесконечно вырабатывать на выходе значение *x*.

Другой цикл:

```
SEQ
   c1 ? x
   WHILE x > = 0
   SEQ
      c2 ! x*x
```

```
c1 ? x
```

возводит в квадрат только положительные числа, а при получении отрицательного числа процесс завершается.

Цикл WHILE используется и в конструкторе PAR:

```
CHAN OF INT cc:
PAR
  WHILE TRUE
    INT x:
    SEQ
      c1 ? x                                    (3.9)
      cc ! x * x
  WHILE TRUE
    INT y:
    SEQ
      cc ? y
      c2 ! y * y
```

Здесь вычисляется четвертая степень входных целых значений. Для этого два процесса необходимо связать каналом *cc*. Поскольку канал *cc* находится внутри данной параллельной конструкции, поэтому он описывается перед началом конструкции предложением CHAN OF, а INT — описание переменной целого типа.

В этой конструкции два процесса образуют конвейер, обе ступени которого работают параллельно. Оба процесса начинают одновременно и взаимодействуют через канал *cc*. Ввод и вывод происходят синхронно. Ввод не завершится, пока не произойдет вывод по тому же каналу, и наоборот. Если один из процессов достиг состояния обмена, а другой еще нет, то первый приостанавливается и ждет, когда второй процесс также будет готов к обмену.

Когда взаимодействие произойдет, оба процесса продолжат работу независимо. Очередная синхронизация произойдет при передаче следующего промежуточного результата.

*Повторитель* FOR используется в конструкциях SEQ, PAR, IF, например:

```
CHAN OF INT c:
SEQ i = 1 FOR n
```

```
c ! i
```

Этот процесс последовательно выводит в канал с значения от 1 до *n.*

Повторитель в конструкции PAR создает массив параллельных процессов:

```
[n + 1] CHAN OF INT pipe:
PAR i = 0 FOR n
  WHILE TRUE
    INT x:
    SEQ
      pipe [i] ? x
      pipe [i + 1] ! x
```

Здесь образуется буфер на *n* значений, работающий по принципу “первым пришел — первым ушел”.

В Оккаме используются процедуры, содержащие формальные параметры, которые при вызове процедуры заменяются на фактические.

Если процесс (3.8) записать в виде процедуры:

```
PROC squar (CHAN OF INT in, out)
WHILE TRUE
  INT x:
  SEQ
    in ? x
    out ! x * x
```

то параллельный процесс (3.9) записывается в простой форме:

```
CHAN OF INT cc:
PAR
  squar (c1, cc)                                    (3.10)
  squar (cc, c2)
```

Параллельные процессы распределяются между связанными в сеть процессорами конструкцией PLACED PAR. Каждый процессор с его локальной памятью исполняет один из процессов-компонент.

Вернемся к примеру (3.10). Пусть имеется двухпроцессорная система, на которой необходимо выполнить параллельные процессы. У каждого из процессоров (обозначим их номерами 1 и 2) имеются локальная память и по два порта связи — link 1.in (входной) и link 2.out (выходной). Физическая связь между процессорами осуществляется соединением попарно их портов, соответствующих одному каналу. Тогда распределение частей задачи (3.10) по процессорам можно описать следующим образом:

```
CHAN OF INT cc:
PLACED PAR
  PROCESSOR 1
    link 1.in IS c1:
    link 2.out IS cc:                                  (3.11.)
      squar (c1,cc)
  PROCESSOR 2
    link 1.in IS cc:
    link 2.out IS c2:
      squar (cc,c2)
```

На рис. 3.8 показана схема соответствующих этой программе соединений. Естественно, перед выполнением программы (3.11) транспьютеры уже должны быть заранее физически соединены согласно рис. 3.8.

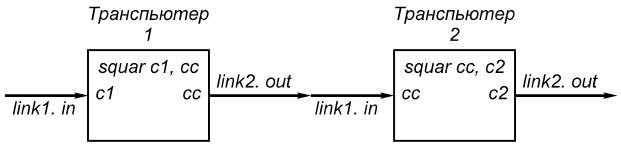



Рис. 3.8 Схема размещения процессов по транспьютерам

Размещение с помощью оператора PLACED PAR может выполняться только для именованных процессов, как это сделано в (3.11) для процедур. Следовательно, конструкция типа:

```
PAR
    a: = b - 5
    d: = b + 1
```

не может быть размещена по двум транспьютерам и может выполняться только в одном транспьютере квазипараллельно (в режиме разделения времени).

Таким образом, вычисления в транспьютерной системе требуют написания двух программ: программы вычислений и программы, закрепляющей ресурсы за процессорами. Более подробно эти вопросы рассмотрены в § 5.4.

Для оценки возможностей транспьютерных систем и языка Оккам рассмотрим пример — выполнение быстрого преобразования Фурье (БПФ), которое является одним из основных алгоритмов цифровой обработки сигналов.

Основной операцией БПФ является операция “бабочка”. Она имеет следующий вид:

$$a' = a + b,$$
$$b' = (a - b) * W^{k}, W = e^{i(2\pi/N)},$$

где все значения комплексные. Эту операцию можно записать в виде одного именованного Оккам-процесса, который по двум входным каналам — ain и bin — получает значения *a* и *b*, а еще по двум — *aout* и *bout* — выдает для последующей обработки *a*' и *b*'.

Программа имеет такой вид:

```
PROC butterfly (CHAN OF REAL32 ain, bin, aout, bout,
VAL REAL32 wreal, wimag)
  REAL32 areal, aimag, breal, bimag:
  WHILE TRUE
    SEQ
    PAR -- одновременное считывание a и b
      SEQ
        ain ? areal
        ain ? aimag
      SEQ
        bin ? breal                                          (3.12)
        bin ? bimag
    PAR -- одновременный вывод a' и b'
      SEQ
        aout ? areal + breal
        aout ? aimag + bimag
      SEQ
        bout ? (wreal * (areal - breal))
         - (wimag * (aimag - bimag))
        bout ? (wreal * (aimag - bimag))
         - (wimag * (areal - breal))
```

Пусть требуется выполнять 8-точечное БПФ. Параллельная программа алгоритма следующая:

```
VAL [12] INT ktable IS [0,1,2,3,0,0,2,2,0,0,0,0]:
  CHAN OF REAL32 c [32]:
    PAR stage = 0 FOR 3
      PAR node = 0 FOR 4                                     (3.13)
        butterfly (c [(stage * 8) + node],
                   c [(stage * 8) + node + 4],
                   c [((stage + 1) * 8)+ node * 2)],
                   c [((stage + 1) * 8) + (node * 2)+ 1],
                   twidle.real [ktable [(stage * 4) + node]],
                   twidle.imag [ktable [(stage * 4) + node]])
```

Здесь в массивах twidle.real и twidle.imag хранятся заранее посчитанные константы $W^k$ twidle[k] = $e^{i(2\pi k/N)}$

Во входные каналы c[0]...c[7] подаются входные данные, из выходных каналов c[24]...c[31] считываются результаты.

Изображая процесс butterfly в виде квадрата, а каналы в виде отрезков с указанием направления передачи данных, получаем сеть, изображенную на рис. 3.9.

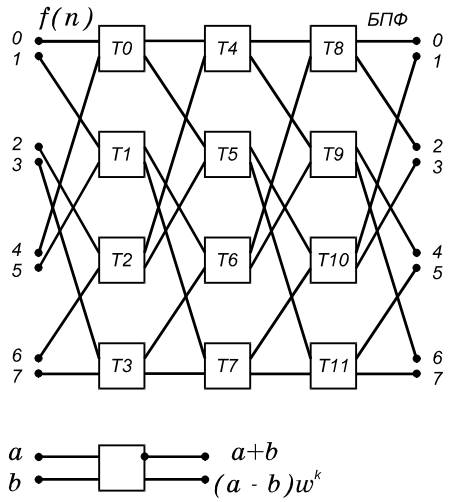



Рис. 3.9. Мультитранспьютерная сеть для БПФ

Если 12 транспьютеров связать между собой в соответствии с этим рисунком и загрузить в каждый программу (3.12) и коэффициенты $W$ в соответствии с программой (3.13), то получим вычислительную систему, которая может выполнять 8-точечное БПФ по программе (3.13).

В этой вычислительной системе транспьютеры образуют конвейер из трех ступеней. Все транспьютеры работают параллельно, позволяя выполнять множественное БПФ в потоковом режиме.

**Организация транспьютеров.** Состав оборудования транспьютера представлен на рис. 3.7. Рассмотрим особенности его структуры.

Каждый канал транспьютера физически состоит из двух одноразрядных каналов, один для работы в прямом, другой — для ра

боты в обратном направлении, обозначаемые как link.in и link.out. Таким образом, один канал транспьютера соответствует двум каналам языка Оккам. Поскольку каждый канал транспьютера имеет автономное управление, то все каналы могут работать независимо друг от друга и от процессоров транспьютера.

АЛУ транспьютера, а значит, и система команд строятся по стековому принципу. На рис. 3.10 представлена регистровая структура центрального процессора.

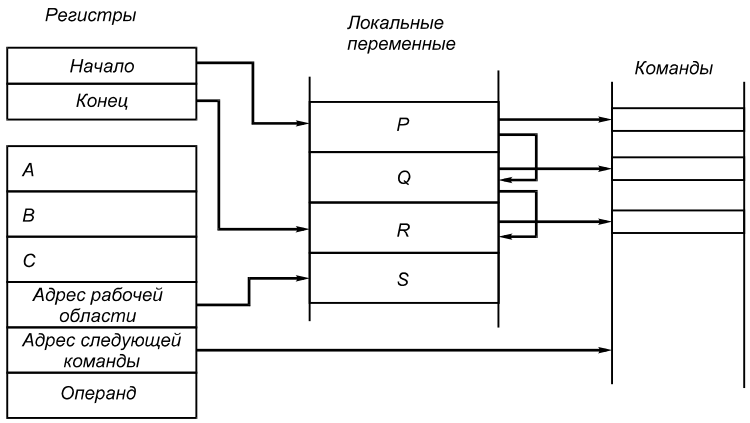



Рис. 3.10. Обработка параллельных процессов

В ЦП используются 6 регистров по 32 разряда каждый:

- указатель рабочей области для локальных переменных программы;
- указатель следующей команды;
- регистр операндов, в котором формируются операнды и команды;
- *A*, *B* и *C* — регистры, образующие вычислительный стек.

В вычислительном стеке выполняются не только арифметические и логические операции, но и команды планирования параллельных процессов и коммуникаций, в него записываются параметры при вызове процедур и др.

Наличие вычислительного стека устраняет необходимость задания в командах явного указания регистра. Например, команда *ADD* складывает значения из регистров *A* и *B*, помещает результат

в *A* и копирует *C* в *B*. Поэтому большинство команд транспьютера (70-80%) являются однобайтовыми.

В транспьютере, кроме вычислительного стека ЦП для целочисленной арифметики, имеется стек для работы над данными с плавающей запятой с регистрами *AF*, *BF*, *CF*.

Список команд транспьютера включает 110 команд. Они делятся на две группы: с прямой адресацией (один байт) и с косвенной адресацией (два или более байтов).

Транспьютер может одновременно обрабатывать любое число параллельных процессов. Он имеет специальный планировщик, который производит распределение времени между ними. В любой момент времени параллельные процессы делятся на два класса: активные процессы (выполняются или готовы к выполнению) и неактивные процессы (ожидают ввода-вывода или определенного времени).

Активные процессы, ожидающие выполнения, помещаются в планировочный список. Планировочный список является связным списком рабочих областей этих активных процессов в памяти и задается значениями двух регистров, один из которых указывает на первый процесс в списке, а другой — на последний. Состояние процесса, готового к выполнению, сохраняется в его рабочей области. Состояние определяется двумя словами — текущим значением указателя инструкций и указателем на рабочую область следующего процесса в планировочном списке. В ситуации, изображенной на рис. 3.10, имеется четыре активных процесса, причем процесс *S* выполняется, а процессы *P*, *Q* и *R* ожидают выполнения в планировочном списке.

Команда транспьютера *start process* создает новый активный процесс, добавляя его в конец планировочного списка. Перед выполнением этой команды в *A*-регистр вычислительного стека должен быть загружен указатель инструкций этого процесса, а в *B*-регистр — указатель его рабочей области. Команда *start process* позволяет новому параллельному процессу выполняться вместе с другими процессами, которые транспьютер обрабатывает в данное время.

Команда *end process* завершает текущий процесс, убирая его из планировочного списка. В Оккаме конструкция *PAR* может закончиться только тогда, когда завершатся все ее компоненты па

раллельного процесса. Каждая команда *start process* увеличивает их число, а *end process* уменьшает. В транспьютере предусмотрен специальный механизм учета числа незавершившихся компонент данной параллельной конструкции (необходимо учитывать как активные, так и неактивные процессы).

При обработке параллельных процессов на самом деле присутствует не один, а два планировочных списка — список высокого приоритета и список низкого приоритета. Процессы с низким приоритетом могут выполняться только тогда, когда список высокого приоритета пуст, что должно быть его нормальным состоянием. Процессы с высоким приоритетом обычно вводятся для обеспечения отклика системы на них в реальном времени.

Коммуникации между параллельными процессами осуществляются через каналы. Для организации этого объема используются команды транспьютера "*input message*" и "*output message*". Эти команды используют адрес канала, чтобы установить, является он внутренним (в том же транспьютере) или внешним. Внутренний канал реализуется одним словом памяти, а обмен осуществляется путем пересылок между рабочими областями этих процессов в памяти транспьютера.

Несмотря на принципиальные различия в реализации внутренних и внешних каналов команды обмена одинаковы и различаются только адресами. Это позволяет осуществить компиляцию процедур безотносительно к способу реализации каналов, а следовательно, и к конфигурации вычислительной системы.

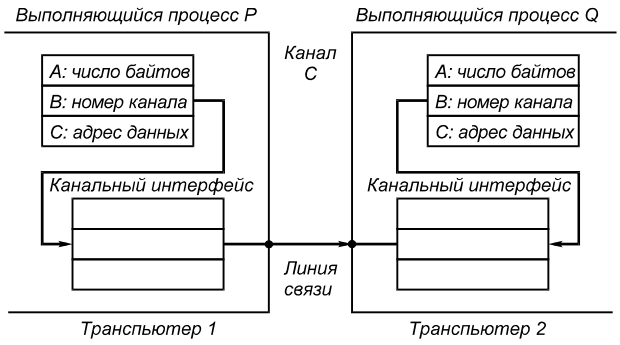
Рассмотрим пересылку данных по внешнему каналу. Команда пересылки направляет автономному канальному интерфейсу задание на передачу данных и приостанавливает выполнение процесса.

После окончания передачи данных канальный интерфейс помещает этот процесс в планировочный список. Во время обмена по внешнему каналу оба обменивающихся процесса становятся неактивными. Это позволяет транспьютеру продолжать обработку других процессов во время пересылки через более медленные внешние каналы.

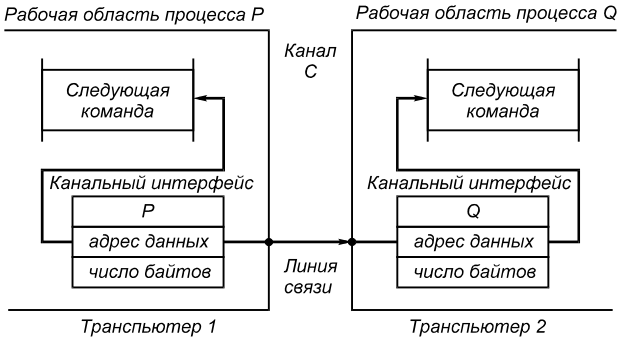
Каждый канальный интерфейс использует три своих регистра, в которые загружаются указатель на рабочую область процесса, адрес пересылаемых данных и количество пересылаемых байтов.

В следующем примере процессы *P* и *Q*, которые выполняются на различных транспьютерах, обмениваются данными по внешнему каналу *C*, реализованному в виде линии связи, соединяющей эти два транспьютера (рис. 3.11).

*а*




*б*




*в*

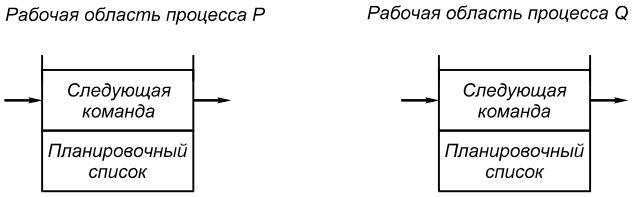



Рис. 3.11. Пересылка данных по внешнему каналу

Пусть *P* передает данные, а Q принимает (рис. 3.11, а). Когда процесс *P* выполняет команду *output message*, регистры канального интерфейса транспьютера, на котором выполняется *P*, инициализируются, а процесс *P* прерывается и становится неактивным. Аналогичные действия происходят в другом транспьютере при выполнении процессом *Q* команды *input message* (рис. 3.11, б).

Когда оба канальных интерфейса инициализированы, происходит копирование данных по межтранспьютерной линии связи. После этого процессы *P* и *Q* становятся активными и возвращаются в свои планировочные списки (рис. 3.11, в).

Для пересылки данных используется простой протокол, не зависящий от разрядности транспьютера, что позволяет соединять транспьютеры разных типов.

Сообщения передаются в виде отдельных пакетов, каждый из которых содержит один байт данных, поэтому наличие буфера в один байт в принимающем транспьютере является достаточным для исключения потерь при пересылках.

После отправления пакета данных транспьютер ожидает получения пакета подтверждения от принимающего транспьютера. Пакет подтверждения показывает, что процесс-получатель готов принять этот байт и что канал принимающего транспьютера может начать прием следующего байта. Форматы пакетов данных и пакетов подтверждения показаны на рис. 3.12.

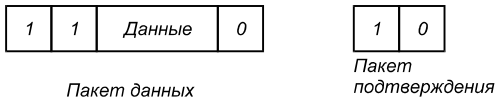



Рис. 3.12. Форматы пакетов данных и пакетов подтверждения

Протокол передачи пакета данных позволяет отсылать пакет подтверждения, как только транспьютер идентифицировал начало пакета данных. Подтверждение может быть получено передающим транспьютером еще до завершения передачи всего пакета данных, и поэтому пересылка данных может быть непрерывной, то есть без пауз между пакетами.

В последних модификациях транспьютеров для упрощения программирования и увеличения пропускной способности физиче

ских каналов связи используется *процессор виртуальных каналов* VCP (Virtual Chanal Processor). VCP позволяет использовать на этапе программирования неограниченное число виртуальных каналов.

## § 3.3. ЭВМ с управлением от потока данных

Как отмечалось в § 1.2, кроме МКМД-ЭВМ типа IF с управлением от потока команд существуют МКМД-ЭВМ типа DF с управлением от потока данных.

Рассмотрим принципы построения DF-ЭВМ на примере машины Денниса [1]. В таких машинах отсутствуют понятия “память данных”, “переменная”, “счетчик команд”. Вместо этого команды (операторы) управляются данными. Считается, что команда готова к выполнению, если данные присутствуют на каждом из ее входных портов и отсутствуют в выходном порту. Программа представляет собой направленный граф, образованный соединенными между собой командами: выходной порт одной команды соединен с входным портом другой команды. Следовательно, порядок выполнения команд определяется не счетчиком команд, а движением потока данных в командах.

Принцип обслуживания потоков данных рассмотрим на уровне языка программирования, а структуру и функционирование DF-ЭВМ — на уровне исполнения конкретной программы на машинном языке.

Хотя для потоковых ЭВМ разрабатываются языки программирования разного типа, для ясности изложения принципов функционирования далее будет использован двумерный графический язык. В состав языка входят следующие понятия: “исполнительный элемент”, “информация”, “связи”.

Основные типы исполнительных элементов приведены на рис. 3.13. Информация представляется в виде *токенов* (зачерненные точки), которые передаются по линиям связи и поглощаются исполнительными устройствами. Имеется два вида информации: числовая и управляющая.

Линии связи — это аналог понятий “переменная”, “память”. Здесь и далее на рисунках используются два типа линий связи: данных (сплошные линии) и управления (штриховые линии).

Операция “размножения” выполняется, если есть токен на входе и выходные линии пусты. Блок выполнения операций работает при наличии двух токенов на входе и свободном выходе.

Блок принятия решения выполняет операцию отношения типа $a > b$, $a = b$; результат — $T$ (истина) или ложь $F$ (ложь). Вентили $T$ и $F$ пропускают токен данных на выход в том случае, если они находятся в состоянии $T$ или $F$. В зависимости от управляющего сигнала на выход передается входная информация $T$ или $F$. Рассмотрим пример выполнения программы, записанной на языке PL-1:

```
Z: PROCEDURE (X, Y);
     DO I = 1 TO X;
          IF Y > 1 THEN Y=Y*Y;

ELSE Y=Y+Y+2;
```

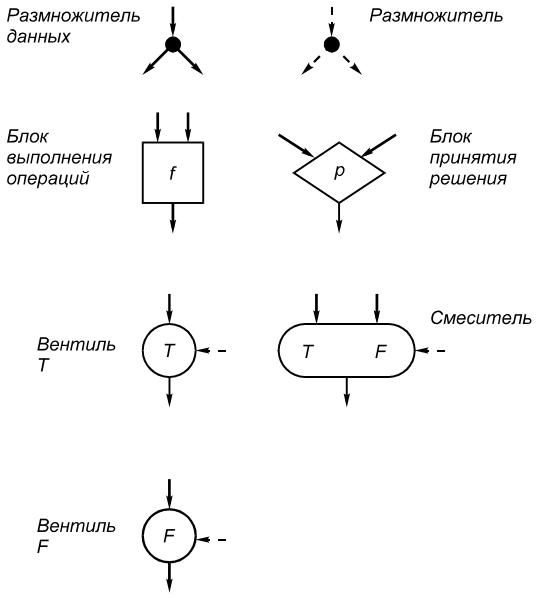



Рис. 3.13.Исполнительные элементы двумерного языка

```
    END;
END;
```

Программа приведена на рис. 3.14. Там же указано начальное состояние программы при значениях входных переменных X = 2 и Y = 0. Слева на рисунке — тело цикла, а справа — управление циклом DO. Отдельные блоки имеют на входах константы. Это означает, что как только блок выполняет предписанную ему работу, константа вновь автоматически восстанавливается на входе.

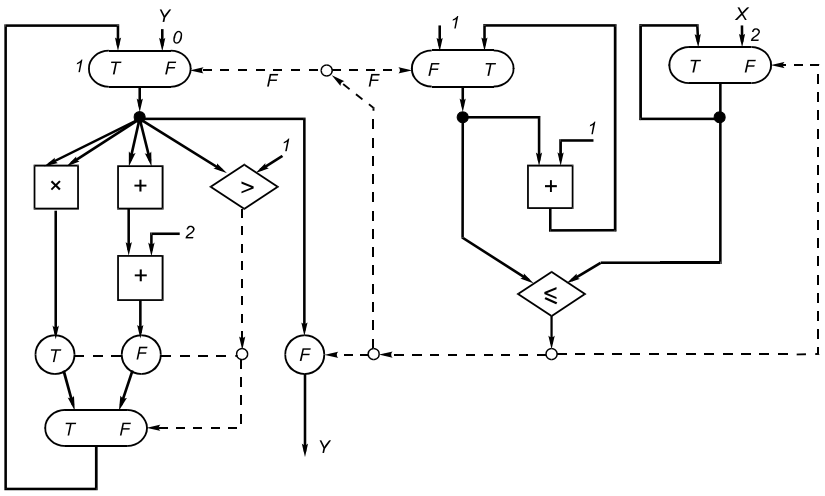



Рис. 3.14. Графическое представление программы на языке потоков данных

Фазы выполнения программы изображены на рис. 3.15. Дадим некоторые пояснения. В соответствии с начальным состоянием три блока готовы к работе. Один смеситель (вверху слева) находится в состоянии готовности, поскольку к нему приложен управляющий токен FALSE, а на его входной линии *F* имеется токен данных. Два других смесителя также готовы к работе. Таким образом, одновременно выполняются три операции (см. рис. 3.15, а).

Первый смеситель передает величину 0 через размножитель на шесть входов других блоков. Через третий смеситель (вверху

справа) и размножитель величина 2 поступает на вход *T* этого же смесителя и на блок принятия решения. Работа второго смесителя (вверху по центру) завершается передачей величины 1 на исполнительный блок и блок принятия решения. Теперь пять блоков готовы к работе в параллельном режиме. Блок принятия решения по критерию “первый операнд отношения больше второго” формирует управляющий токен FALSE, который посылается в три блока: вентили *T*, *F* и нижний смеситель. В результате блок принятия решения по критерию “первый операнд отношения меньше или равен второму” формирует управляющий токен TPUE, направляемый четырем адресатам. Блок сложения пошлет величину 2 обратно на вход второго смесителя.

Момент, когда к работе готовы три вентиля, соответствует изображению на рис. 3.15, б. Поскольку вентиль *F* имеет управляющий токен TRUE, величина 0 будет просто “поглощена” этим вентилем (предотвращается вывод из программы ошибочного результата). То же самое произойдет и в вентиле *T*, но второй вентиль *F* формирует величину 2, которая переведет в состояние готовности расположенный под этим вентилем смеситель, пересылающий величину 2 на другой смеситель. При графическом изображении программы удается отразить тот факт, что оба арифметических выражения, определенные в операторе IF, могут вычисляться параллельно с вычислением отношения оператора IF (*Y* > 1). Описываемые действия завершаются работой вентилей *T* и *F*, осуществляющих выбор нужного выражения с целью его передачи для нового шага итерации.

Одно из следующих состояний программы отражено на рис. 3.15, в. Величина 2 будет возвращена на вход программы в результате нового повторения циклического процесса в левой части графа, т. е. действие программы аналогично предыдущему. На выходные линии никакие данные помещены не будут, но при этом блок принятия решения по критерию “первый операнд отношения больше второго” формирует на выходе значение TRUE, что дает возможность входным данным вентиля *T* пройти на его выход и далее, в верхнюю часть графа.

Очередное состояние программы иллюстрирует рис. 3.15, г.

Блок принятия решения по критерию “первый операнд отношения меньше или равен второму” формирует управляющий токен FALSE, рассылаемый четырем адресатам программы.

На рис. 3.15, д изображено следующее состояние программы: на выходе вентиля *F*, расположенного в середине графа,

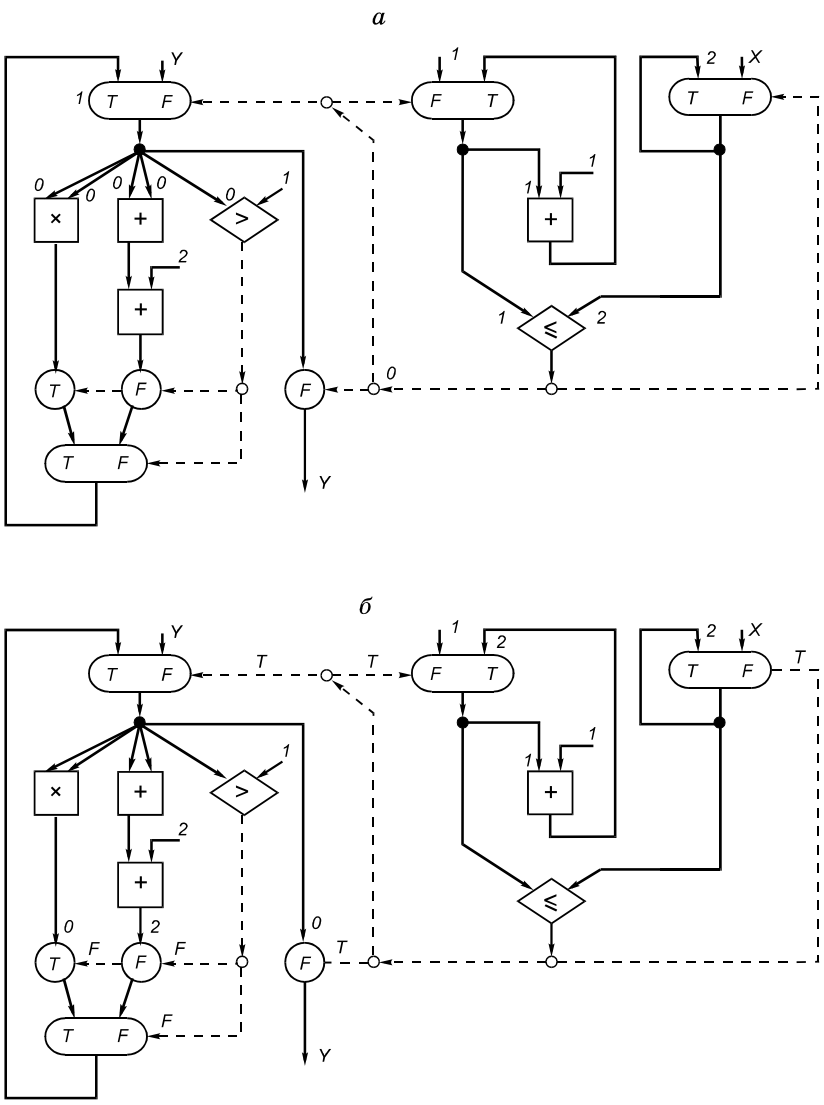



Рис. 3.15. Процесс выполнения потоковой программы

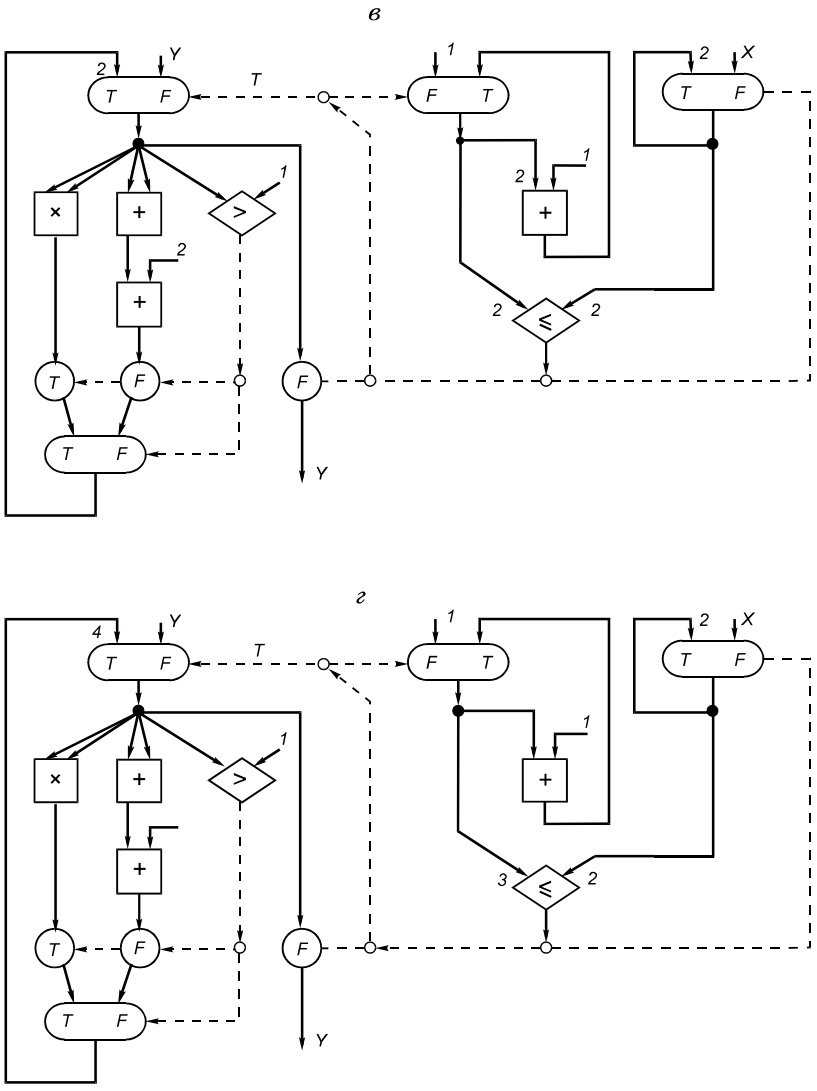



Рис. 3.15. Процесс выполнения потоковой программы (продолжение)

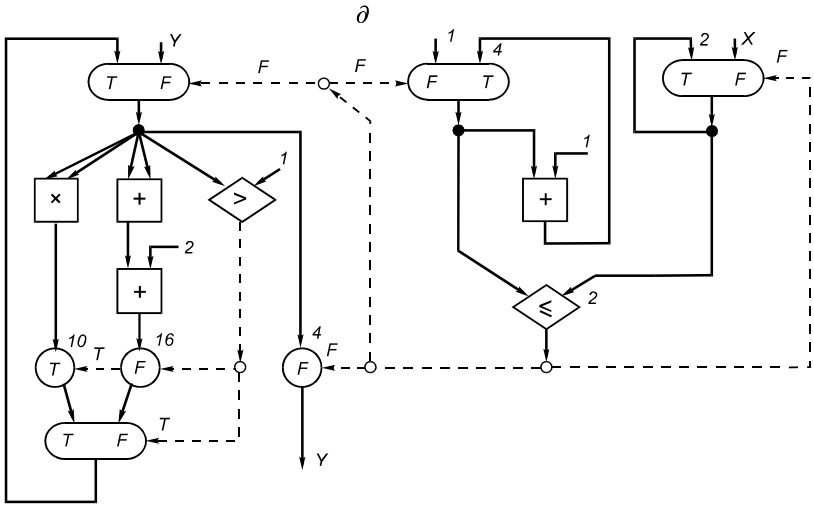



Рис. 3.15. Процесс выполнения потоковой программы (продолжение)

имеются управляющий токен FALSE и токен данных, значение которого равно 4 (поэтому на выходной линии программы появится величина 4). Хотя результат получен, программа все еще продолжает работать, поэтому важно проследить ее работу до момента остановки. Вентиль *F*, расположенный в левой нижней части, уничтожит оба своих входных сигнала. Вентиль *F* пропустит величину 16, которая пройдет через смеситель в верхнюю часть графа.

Здесь дальнейшее движение этой величины будет заблокировано, поскольку на вход соответствующего смесителя подается управляющий токен FALSE. Таким же образом окажутся заблокированными величины 2 и 4, поступающие соответственно на правый и средний смесители.

Описанный способ представления программы позволяет достичь максимального параллелизма. В частности, в этой простой программе он составил по тактам: 5 (см. рис. 3.15, а), 3 (см. рис. 3.15, б - г), 2 (см. рис. 3.15, д).

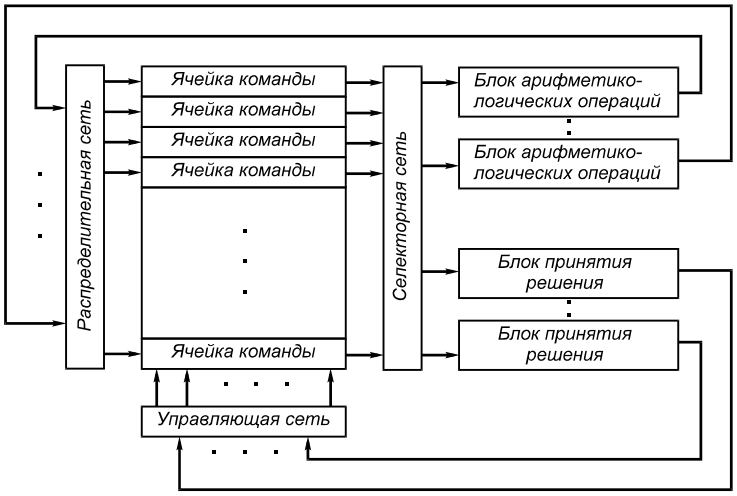



Рис. 3.16. Структура машины Дениса

В качестве примера машины, реализующей принципы DF-ЭВМ, опишем машину Денниса (рис. 3.16). Она состоит из трех секций. К первой относится память с ячейками команд. Эти ячейки являются пассивной памятью, но в них содержатся и некоторые логические схемы. Блоки арифметико-логических операций вырабатывают данные, а блоки принятия решений — управляющую информацию. Машина Денниса может содержать произвольное число блоков обоих типов, числом которых и определяется ее реальный параллелизм. Эти блоки составляют вторую секцию. Третья секция состоит из трех переключающих сетей. Селекторная сеть извлекает готовую к выполнению команду и направляет ее в соответствующий блок операций или блок принятия решения. Распределительная сеть принимает пакет результата (данные и адрес назначения) и направляет его в указанную ячейку команды. Точно так же управляющая сеть передает управляющий пакет (результат принятия решения и адрес назначения) в указанную ячейку команды.

Вопрос представления в DF-ЭВМ структур данных (массивы, графы) достаточно сложен. Один из вариантов его решения предусматривает введение дополнительной памяти для хранения структур данных, а также отдельных блоков для их обработки. Запуск

на обработку структуры данных выполняется командой потокового типа, а обработка больших массивов производится методами, близкими к традиционным.

Команда такой машины соответствует одному из элементов, представленных на рис. 3.15 и содержит код операции; блок состояний, описывающий наличие операндов; сами операнды; адреса, по которым нужно разослать результат.

Этот язык имеет достаточно низкий уровень и не пригоден для написания больших программ, поэтому разрабатывается ряд языков с традиционной формой представления операторов. Символьный ЯВУ можно получить и на основе языка Денниса, для этого необходимо исполнительные элементы (см. рис. 3.13) представить в виде операторов ЯВУ. Блок выполнения операций при этом отображается в обычный оператор присваивания; блок принятия решения совместно со смесителем преобразуются в подобие оператора: IF (условие) THEN $X$: = $X$1 ELSE $X$: = $X$2. Несколько сложнее с размножителями данных, однако эти и другие элементы графического языка все же могут быть заменены обычными или специально введенными операторами символьного типа.

На таком символьном ЯВУ программа (см. рис. 3.14) будет изображаться в виде списка операторов, причем порядок следования операторов в списке может быть произвольным и ни в коей мере не определяет порядок вычислений.

Чтобы повысить уровень программирования в символьных ЯВУ для потоковых машин, некоторые устойчивые совокупности операторов, используемые, в частности, для организации циклов, заменяются одним оператором, которым и пользуется программист.

Порядок вычислений в потоковой символьной программе определяется связями между операторами и готовностью данных на линиях этих связей. Чтобы правильно преобразовать графическую программу (см. рис. 3.14) в символьную, необходимо всем связям на рис. 3.14 присвоить уникальные, неповторяющиеся имена и эти имена перенести в операторы. Тогда символьная программа по связям будет строго соответствовать графической. Присвоение уникального имени называется в потоковых машинах *принципом однократного присваивания* и используется практически во всех графических и символьных языках.

## § 3.4. Надежность и отказоустойчивость параллельных ЭВМ

Параллельные ЭВМ, в частности, многопроцессорные, наряду с высоким быстродействием естественным образом обеспечивают высокую надежность и отказоустойчивость вычислений. Это достигается за счет многоэлементности (многопроцессорности) и способности к реконфигурации этих ЭВМ. В этом параграфе надежность и отказоустойчивость рассматриваются преимущественно в отношении оборудования, а не программного обеспечения.

Приведем некоторые сведения из теории надежности, необходимые для дальнейшего рассмотрения [18].

***Надежность*** определяется как свойство технического изделия выполнять заданные функции в течение требуемого времени.

Основным понятием теории надежности является **отказ**, под которым понимают случайное событие, нарушающее работоспособность изделия. Применительно к вычислительной технике различают два вида отказов: устойчивые (собственно отказы) и самоустраняющиеся (сбой, перемежающиеся отказы). Сбой возникает вследствие одновременного неблагоприятного изменения нескольких параметров и существует кратковременно. Перемежающиеся отказы могут возникать, например, при плохом контакте в соединителе. Сбои встречаются в ЭВМ наиболее часто.

Основными параметрами надежности для невосстанавливаемых изделий являются ***интенсивность отказов*** $\lambda$, вероятность безотказной работы $P(t)$, среднее время безотказной работы $T$:

$$\lambda = \frac{m}{N \cdot t}$$

$$P(t) = e^{-\lambda t}$$

$$T = \int_0^\infty P(t)dt = \frac{1}{\lambda},$$

Здесь $m$ - число изделий, отказавших за время $t$, а $N$ — число исправных элементов на начало промежутка времени. Следова

тельно, $\lambda$ определяет долю (а не количество) изделий, отказавших в единицу времени, в качестве которой обычно принимают один час. Интенсивность принимается постоянной на этапе нормальной эксплуатации изделий.

Для микросхем обычно $\lambda=10^{-6}...10^{-8}$ 1/час, поэтому среднее время безотказной работы большое: $T=10^{6}...10^{8}$ часов. Но в ЭВМ обычно входят тысячи микросхем. Поскольку

$$\lambda_{\text{ЭВМ}} = \sum_{i=1}^{N} \lambda_i ,$$

где $\lambda_{\text{ЭВМ}}$, $\lambda_i$ — интенсивность отказов ЭВМ и микросхемы соответственно, а $N$ — число микросхем в составе ЭВМ, то время безотказной работы ЭВМ может составлять всего десятки или сотни часов.

Для восстанавливаемых систем наиболее важным параметром является наработка на отказ $t_{cp}$. Если восстановление является полным, то время наработки на отказ $t_{cp}$ восстанавливаемой системы соответствует времени безотказной работы $T$ невосстанавливаемой системы.

По назначению вычислительные системы, к которым относятся и многопроцессорные ЭВМ, можно разделить на две группы: системы для управления в реальном масштабе времени и информационно-вычислительные системы. Для первых отказ может вызвать тяжелые последствия, поэтому одним из основных показателей эффективности таких систем является надежность. Вторые не критичны к отказу, основным показателем их эффективности является производительность.

Рассмотрим показатели надежности ЭВМ, применяемых для систем управления в реальном времени. Как говорилось выше, наработка на отказ ЭВМ невысока. Основной способ повысить ее — ***резервирование***. Различают два вида резервирования: общее и поэлементное. При общем резервировании резервируется вся ЭВМ, т. е. в случае выхода из строя она заменяется такой же. При поэлементном резервировании резервируются отдельные части ЭВМ (процессоры, каналы) и в случае отказов они заменяются идентичными. Наиболее употребительным является постоянное (горячее) резервирование, при котором резервное устройство выполняет ту же нагрузку, что и основное, и при отказе последнего

резервное устройство без задержки замещает основное. Число резервных устройств может быть более одного.

Общее резервирование значительно повышает надежность системы, в частности для системы без восстановления вероятность безотказной работы $P_{рез}(t)$ равна:

$$P_{рез}(t) = 1 - \left[\left(1 - P(t)\right)\right]^m,$$

где $P(t)$ — вероятность безотказной работы одной ЭВМ, а $m$ — число ЭВМ в системе. Так, если $P(t)$ = 0,9, а m = 2 (используется дублирование), то $P_{рез}(t)$ = 0,99. Для восстанавливаемых систем и при m = 3...4 резервирование позволяет достичь характеристик, близких к идеальным.

Однако, резервирование неэкономично, так как объем оборудования возрастает в $m$ раз, а производительность остается на уровне одной ЭВМ. Такую систему нельзя назвать и параллельной, поскольку в последней каждая ЭВМ выполняет независимую работу. Значительно эффективнее поэлементное резервирование, которое прямолинейно реализуется структурой многопроцессорной ЭВМ.

Пусть многопроцессорная ЭВМ содержит $m$ одинаковых процессоров, $l$ из которых являются избыточными. Избыточный процессор полноценно заменяет любой из основных в случае отказа последнего, то есть реализуется плавающее резервирование. Каждый из основных процессоров выполняет независимую часть работы.

Пусть процессор имеет интенсивность отказов $\lambda$ и интенсивность восстановления $\mu$ ($\mu = 1/T_\mu$, где $T_\mu$ — среднее время восстановления процессора). В этом случае средняя наработка на отказ многопроцессорной системы с $l$ резервными элементами:

$$t_{ср,рез} = \frac{1}{(m-e)\cdot\lambda} \sum_{i=0}^{l} \frac{(m-l)!}{(m-l+i)!} \left(\frac{\mu}{\lambda}\right)^i$$

Так как $1/(m\lambda)$ — наработка на отказ неизбыточной ЭМВ, то при малых $l$ выигрыш надежности избыточной ЭВМ будет:

$$G = \frac{t_{ср,рез}}{t_{ср}} = \sum_{i=0}^{l} \frac{(m-l)!}{(m-l+i)!} \left(\frac{\mu}{\lambda}\right)^i$$

В практически важных случаях $\mu/\lambda >> 1$, тогда $G \approx [\mu/(m\lambda)]^l$. Для ЭВМ обычно $\mu/(m\lambda) \geq 100$, то уже при $l=2$ или 3 наработка на отказ избыточной ЭВМ будет приближаться к границе долговечности.

Система с плавающим резервированием должна обладать способностью к *реконфигурированию*.

Все вышеприведенное относилось к системам управления реального времени, которые можно определить как двухпозиционные системы: работает — не работает. Именно для таких систем основной характеристикой функционирования является надежность.

Информационно-вычислительные системы являются многопозиционными. Функции, выполняемые этими системами, можно разделить на основные и второстепенные. Если отказ одного из элементов делает невозможным выполнение одной из второстепенных функций, это не препятствует дальнейшему функционированию системы. Если же отказ затрагивает исполнение основной функции, то в результате автоматической реконфигурации выполнение этой основной функции передается на оставшиеся работоспособные элементы (возможно, с вытеснением второстепенных функций). В таких системах возможен отказ более чем одного элемента при сохранении работоспособности системы. После восстановления отказавших элементов система будет выполнять функции в полном объеме. Восстановление производится в процессе функционирования системы.

Системы, обладающие вышеуказанными свойствами, называют отказоустойчивыми (толерантными, “живучими”).

Очевидно, для отказоустойчивых систем не подходит понятие отказа в вышеописанном случае, а надежность не является основным показателем эффективности. В таких системах отказом следует считать такой отказ элементов системы (в общем случае многократный), который вызывает поражение одной из основных функций. Что касается эффективности, то для информационно-вычислительных систем одним из основных ее показателей явля

ется производительность, которую можно определить выражением:

$$V_c = V \cdot \sum_{i=1}^{m} i \cdot p_i ,$$

где $V_c$ — средняя производительность системы, $V$ — производительность одного элемента системы (процессора), $m$ — максимальное число элементов в системе, $i$ — число элементов в конфигурации, $p_i$ — вероятность этой конфигурации.

Системы с восстановлением имеют существенно лучшее распределение вероятностей $p_i$, чем системы без восстановления.

Частным случаем отказоустойчивых систем являются системы с постепенной деградацией, в которых ремонт и восстановление невозможны или по каким-либо причинам нецелесообразны. Такие системы продолжают функционировать до тех пор, пока число работоспособных элементов не достигнет минимально допустимого уровня.

Центральным качеством надежных и отказоустойчивых систем является автоматическая реконфигурация. Реконфигурация возможна, если в многопроцессорной ЭВМ имеются аппаратные и программные средства *контроля* и диагностики, реконфигурации и повторного запуска системы после отказа (рестарта).

Средства контроля могут быть реализованы аппаратным, программным или смешанным способом. Аппаратные средства контроля предназначены для контроля передачи информации и контроля правильности выполнения арифметико-логических операций.

Контроль путей пересылки информации (память — АЛУ, память — ВнУ и др.) производится на основе избыточного кодирования (коды Хэмминга, Грея и др.). Особенное распространение имеет простой код с проверкой четности, требующий только одного дополнительного разряда на блок двоичных разрядов (обычно восьми). Эти же методы и дублирование блоков используется для контроля операций в АЛУ. Аппаратный контроль выполняется в темпе основных вычислений, что и является основным его достоинством.

Программный контроль не требует дополнительного оборудования, однако он выполняется с задержкой во времени. К программным методам контроля относятся тестовый и программно-логический контроль. Тестовый контроль выполняется периодически, либо при наличии свободного времени в ЭВМ, либо после обнаружения отказа другими средствами. В этом случае тестовый контроль частично выполняет функции диагностики. Примерами программно-логического контроля являются: двойной просчет, решение задачи по основному и упрощенному алгоритму, проверка предельных значений переменных и др.

Аппаратной основой реконфигурации является наличие развитой системы коммутации, позволяющей исключать из структуры многопроцессорной ЭВМ неисправные элементы в случае отказов и устанавливать новые связи между исправными элементами. Разработано большое количество типов коммутаторов. Наиболее перспективными являются коммутаторы типа многомерный куб. Следует отметить, что коммутаторы являются неотъемлемыми элементами как параллельных, так и отказоустойчивых ЭВМ. Более подробно коммутаторы описаны в § 2.3.

К системному и прикладному программному обеспечению отказоустойчивой ЭВМ предъявляются следующие требования:

1. Вся адресация памяти, ВнУ, каналов должна выполняться на логическом уровне. В этом случае отказ элемента приводит только к изменению таблиц связи логических и физических адресов.

2. Операционная система должна носить распределенный характер, то есть в процессорах должны находиться копии ОС или ее частей.

В наибольшей степени этим условиям удовлетворяют параллельные ЭВМ типа МКМД с децентрализованным управлением.

Для обеспечения рестарта в процессе вычислений необходимо создавать контрольные точки, то есть запоминать результаты вычислений. Рестарт осуществляется с ближайшей контрольной точки. Чем чаще устанавливаются контрольные точки, тем меньше времени будет потрачено на повторение вычислений в процессе рестарта, однако создание контрольных точек также требует дополнительного времени.

## Контрольные вопросы

1. Чем отличаются МКМД ЭВМ с общей памятью от МКМД ЭВМ с локальной памятью?
2. Определите понятия процесс, ресурс, синхронизация, назовите виды процессов.
3. Что такое критическая секция, семафор, занятое ожидание? Опишите операции над семафором.
4. Что такое дескриминаторы процессов и ресурсов? Каков их состав?
5. Охарактеризуйте основные операции над процессами.
6. Охарактеризуйте основные операции над ресурсами.
7. В чем состоит централизованное и децентрализованное управление в многопроцессорной ЭВМ?
8. Что такое транспьютер, какова его структура?
9. Назовите основные конструкции языка Оккам.
10. Охарактеризуйте средства описания параллелизма в языке Оккам.
11. Опишите структуру центрального процессора транспьютера и особенности его системы команд.
12. Как осуществляется обмен данными в системе с несколькими транспьютерами.
13. Определите основные понятия двухмерного графического языка для системы с управлением от потока данных.
14. Приведите пример программы на двухмерном графическом языке, опишите ее функционирование.
15. Объясните, что такое надежность и отказоустойчивость многопроцессорных ЭВМ.