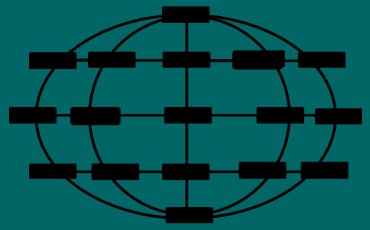
Основы Параллельного программирования



ЛЕКЦИЯ 4

Программирование взаимодействующих процессов

В.Э.Малышкин

Новосибирский Национальный исследовательский университет

Новосибирский технический госуниверситет

Новосибирск 2010

http://ssd.sscc.ru

IV. Программирование взаимодействующих процессов

- Ниже приведены несколько примеров программирования взаимодействия параллельно исполняющихся процессов. Все они определяют асинхронное взаимодействие процессов.

Асинхронное программирование

- Асинхронная модель вычислений наиболее пригодна для описания вычислений на мультипроцессоре и/или мультикомпьютере. Асинхронная программа определяет систему независимо исполняющихся и взаимодействующих процессов. Существует много различных воплощений этой модели вычислений в различных языках программирования. Здесь эта модель рассматривается в самом общем виде.

Понятие асинхронной программы

Асинхронная программа (А-программа) - это конечное множество А-блоков $\{A_k \mid k \in \{1,2,...,m\}\}$ определенных над информационной IM и управляющей CM памятями.

- Каждый А-блок $A_k = <tr(ak), ak, c(ak)>$ состоит из *спусковой* функции *tr(ak) (trigger function),* операции *ak, ak* $\in$ F и *управляющего* оператора *c(ak).*
- Памятью называется конечное множество ячеек памяти, способных хранить значения переменных.
- Спусковая функция *tr(ak)* - это предикат, в программе обычно задается условным выражением либо логической функцией. Как обычно здесь, операция *ak* реализуется в программе одноименной процедурой, вычисляющей функцию f_a.

- Управляющий оператор *c(ak)* - оператор присваивания или процедура, меняющая значения ячеек управляющей памяти CM (например, чтобы разрешить или запретить исполнение какого-нибудь A-блока).
- Каждой переменной из in(*ak*) ∪ out(*ak*) в памяти IM соответствует ячейка, в которой хранится ее значение.

Выполнение A-программы состоит:

- в вычислении значения спусковой функции *tr(ak)* для всех, части A-блоков или ни одного,
- в выполнении ни одного, части или всех A-блоков A_k таких, что *tr(ak)*=true при текущем состоянии памяти IM ∪ CM.

- Таким образом, исполняющая система в зависимости от имеющихся в наличии свободных на этот момент ресурсов имеет право инициировать любое подмножество А-блоков, готовых к исполнению (с истинной *tr(ak)*). Понятно, что при разных исполнениях асинхронной программы на каждом этапе исполнения будут инициированы, вообще говоря, разные подмножества А-блоков. Это обстоятельство (недетерминизм исполнения А-программы) сильно усложняет отладку асинхронной программы.

- Выполнение А-блока *Ak* состоит в исполнении:
- процедуры *ak* с теми значениями, которые имеют переменные из in(*ak*) в текущий момент, при этом вычисляются значения переменных из out(*ak*),
- управляющего оператора *c(ak)*.
- Следовательно, спусковые функции и управляющие операторы - это те средства, с помощью которых задается управление в асинхронных программах .
- А-программа считается завершенной, когда ни один блок не выполняется и ни один А-блок не может быть инициирован, так как для всех $к=1,...,m$ значение *tr(ak)*=false .
-

- **П1**

integer ***x, y, z*;**

input (x,y,z)

asynchronous_do

tr(ak)	***ak***
x<y* V *x<z	***x*:=*x* + 1;**
y<z* V *y<x	***y* :=*y* + 1;**
z<x* V *z<y	***z* :=*z* + 1;**

end_do

- Программа примера П1 позволяет в некотором порядке уравнять значения переменных *x, y, z*, доведя их с помощью прибавления единицы до значения наибольшей их них. Это программа максимальной непроцедурности, в ней нет ограничений, которые бы не были обусловлены наличием информационной зависимости между операторами.

П2.Большая непроцедурность часто мешает эффективно выполнить программу и в таких случаях требуется уменьшать непроцедурность программы (представления алгоритма), сделать управление более жестким, более определенным, не оставляющим места на размышления в ходе вычислений. Для программы примера П1 это можно сделать, убрав дизъюнктивные члены из спусковых функций, что усилит управление (увеличит множество ограничений в нем).

integer *x, y, z*;
input (x,y,z)
asynchronous_do

tr(ak)	*ak*
x<y	*x:=x + 1;*
y<z	*y :=y + 1;*
z<x	*z :=z + 1;*

end_do

- П3.Программа нижеследующего примера отличается от программы примера П2 тем, что в спусковые функции добавлены новые конъюнктивные и дизъюнктивные члены. Они добавляют новые ограничения в управление и в результате получается почти последовательная программа. Параллельное выполнение двух А-блоков возможно лишь при условии равенства значений двух переменных (при равенстве значений всех трех переменных программа завершается).
- *input (x,y,z)*
- asynchronous_do
  - tr(ak) ak
  - $x<z \;\&\; x<y$ $V(x=z\&x<y)V(x=y\&x<z)$ $x:=x+1;$
  - $y<x \;\&\; y<z$ $V(y=x\&y<z)V(y=z\&y<x)$ $y:=y+1;$
  - $z<y \;\&\; z<x$ $V(z=y\&z<x)V(z=x\&z<y)$ $z:=z+1$
- end_do

П4.В примерах П1 - П3 А-блоки не имели управляющего оператора (более точно, был пустой управляющий оператор). Рассмотрим теперь организацию конвейерного исполнения А-блоков. Схема программы показана на рис. 4.2.

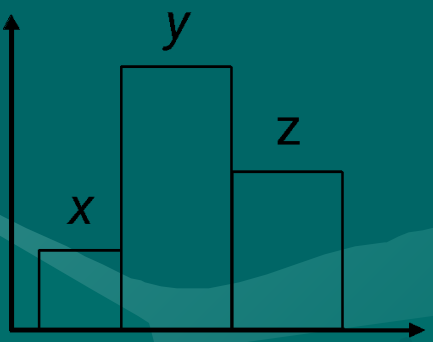


Рис. 4.1.

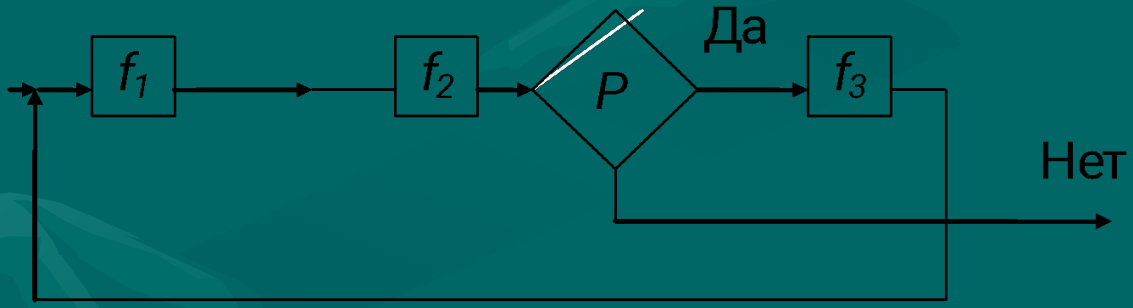


Рис. 4.2.

Асинхронная программа, реализующая этот конвейер (в программе не показаны переменные, обрабатываемые этой программой, демонстрируется только конструирование управления), может иметь вид.

integer ***x, y, z***;

asynchronous_do

***x*:=1; *y*:=*z*:=0;**

tr(ak)	***ak***	***c*(a_k)**
***x*=1**	***f1***	**[*y*:=1; *x*:=0];**
***y*=1**	***f2***	**[*z*:=1, *y*:=0];**
***P* & *z*=1**	***f3***	**[*x*:=1; z:=0];**

end_do

- В этой программе операторы присваивания $x:=1$; $y:=z:=0$; значений управляющим переменным разрешают инициировать операцию *f1* и запрещают инициирование операций *f2* и *f3*. Таким образом, в начальный момент времени сможет быть инициирован только А-блок *f1*. После выполнения операции *f1* управляющий оператор *c1* (здесь это оператор [$y:=1$; $x:=0$]) запретит исполнение А-блока *f1* и разрешит исполнение А-блока *f2* и т.д. Цикл конвейера завершится, когда предикат *P* станет ложным и не позволит инициировать А-блок *f3*.
- Асинхронное программирование оказалось весьма трудоемким делом, особенно отладка программы, в силу обилия возможностей совершить ошибки в синхронизации процессов. Рассмотрим некоторые проблемы асинхронного программирования.

Некорректное вычисление данных

- Возникновение некорректных данных иллюстрирует следующий простой пример. Пусть необходимо начислить зарплату и вычислить сумму денег, подлежащих выдаче на руки. Оставляя в стороне излишние здесь детали, будем предполагать, что зарплату составляет некоторая базисная зарплата N_0 плюс надбавки $N_1, N_2, \ldots, N_n$ (выражаются в процентах к базисной зарплате) минус налог N_{n+1} (выражаются в процентах к начисленной сумме).

Пусть процессы $P_0, P_1, P_2, \dots, P_n, P_{n+1}$ выполняют эти операции. Для вычисления корректного результата процесс P_{n+1}, вычисляющий сумму налога, должен выполняться последним, процесс P_0 - первым, а процессы $P_1, P_2, \dots, P_n$ могут исполняться в любом порядке, хотя само суммирование должно, конечно, выполняться последовательно. Все другие варианты выполнения процессов приведут к некорректным результатам.

Такого сорта ошибки асинхронных программ крайне неприятны, трудно локализуемы и неповторимы. Если позволить процессам $P_1, P_2, \dots, P_n, P_{n+1}$ выполняться асинхронно, то при разных выполнениях асинхронной программы могут получаться разные результаты и повторить предшествующее тестирование удастся только случайным образом. Понятно, что $(P_0 + P_1 + P_{n+1} + P_2 + \dots + P_n) \neq (P_0 + P_{n+1} + P_1 + P_2 + \dots + P_n)$, здесь имеется ввиду, что процессы выполняются в написанном порядке.

Некорректное считывание данных

Следующий пример иллюстрирует этот тип ошибки. Пусть в банке *A* есть счет *acc1*, на котором находится 500 тыс. руб., а в банке *B* - счет *acc2*, на котором находится 300 тыс.руб, и необходимо переслать 100 тыс.руб. со счета *acc1* на счет *acc2*. Сумма денег на обоих счетах неизменна до и после выполнения пересылки и равна 800 тыс. руб. Пусть процесс *P1* посылает деньги из банка *A* в банк *B*, а процесс *P2* принимает посланные деньги в банке *B*. Процессы схематически могут быть описаны так:

- Первоначально
- *A.acc1*=500 тыс.руб.
- *B.acc2*=300 тыс.руб.

- Процесс P_1 Процесс P_2
- *A.acc1*:=*A.acc1* - 100; receive (*x*,*A*,*y*);
- *x* := 100
- send (*x*,*B*,*y*); *B.acc2* := *B.acc2* + y;
- Результат
- *A.acc1*=400 тыс.руб.
- *B.acc2*=400 тыс.руб.

В процессе P_1 счет *A.acc1* вначале уменьшается на 100 тыс. руб., а затем 100 тыс. руб. посылаются в банк *B*. В процессе P_2 сначала принимаются 100 тыс. руб. из банка *A*, а затем увеличивается сумма на счету *B.acc2*.

Однако процесс *P*, контролирующий перевод денег и проверяющий сохранение значения суммы *A.acc1+B.acc2*, может считывать ошибочные данные. Понятно, что если *P* будет считывать данные строго до или после выполнения операции перевода денег, то результат суммирования будет одинаков. Однако в ходе выполнения процессов P_1 и P_2 при разных вариантах считывания значений *A.acc1* и *B.acc2* сумма *A.acc1+B.acc2* может равняться 700 тыс.руб. (если значение *A.acc1* было считано после его изменения, а значение *B.acc2* - до изменения) , 800 тыс.руб. и 900 тыс.руб. при других возможных вариантах считывания данных.

Message passing interface

Широко известным примером реализации модели асинхронного программирования является message passing interface (MPI). Этот метод программирования определяет программу как систему взаимодействующих процессов и стандартизует обмен данными. Обмен всегда происходит через каналы, связывающие два процесса. Такой канал реализуется очередью сообщений, каждый процесс сам определяет свою готовность начать исполнение или завершить его.

В разных формах MPI изучался в асинхронных вычислениях с конца 60-х годов. В настоящее время MPI используется практически во всех коммерческих мультикомпьютерах в качестве основного средства параллельного программирования. Идеи программирования с MPI рассматриваются в самой общей форме.

Определение MPI

Предполагается, что программа определяется как система независимых, параллельно протекающих взаимодействующих процессов. Как обычно, здесь *процесс* это исполняющаяся программа со всеми обрабатываемыми данными. Исполнение этой же программы с другими данными определяет другой процесс.

Взаимодействие определяется как посылка сообщения из одного процесса в другой и прием сообщения. *Сообщение* - любая последовательность битов, которая чаще структурируется и определяется как значение набора (кортежа, структуры) переменных. Сообщение посылается в *канал* в одном процессе и выбирается из одноименного канала в другом процессе.

Канал может рассматриваться как очередь. Оператор send записывает сообщение в канал, в канале может накопиться очередь не выбранных сообщений. Оператор receive выбирает сообщение из канала, если оно есть в канале.

- В программе для использования MPI прежде всего следует описать каналы, например:
- ch *queue* (<*описание_переменной*>)
- Описание определяет канал *queue*, который в состоянии хранить значения переменной канала описанного типа. Переменная канала может быть простой или структурированной. Возможны описания вида:
- ch *symbol* (char);
- ch *data* (*day, month, year* : integer);
- ch *personal_date* (*name* : string, *age* : integer);
- Запись сообщения в канал производится оператором send.
- send <*имя_канала*> (<*выражение*>);

- *Выражение* должно вырабатывать значение переменной, описанной в определении канала. Например:
- send *data* (*d+1, m, y*);
- Переменные *d, m* и *y* определяются в процессе-отправителе и должны содержать целые значения, как это и определено в описании канала *data*.
- Данные выбираются из канала оператором receive.
- receive *personal_date* (*n, a*);
- Переменные *n* и *a* определяются в процессе-получателе и должны быть описаны так же, как переменная канала. После выполнения оператора receive переменная *n* может содержать значение “Иванов И.И.”, а переменная *a* - значение “30”.

- Если канал предполагается неограниченным (асинхронный MPI), то оператор send может быть выполнен неограниченно много раз (*не блокируемый оператор*).
- Оператор receive получает значение из канала, если он не пуст. Если в канале нет значения, выполнение оператора receive задерживается (*блокируемый оператор*) до появления значения в канале.
- Каналы описываются как глобальные объекты, поскольку они разделяются процессами программы. Можно определить массив каналов, например:
- ch *data* [1:k](*day, month, year* : integer);
- Допускается, чтобы несколько процессов посылали сообщения в один и тот же канал. Аналогично, несколько процессов могут получать данные из одного канала. Описанный канал в начальный момент пуст.

- П1.Одно из типичных межпроцессных взаимодействий определяет взаимодействия клиента и производителя. Производитель создает продукт и продает его многим клиентам. Программа их взаимодействий может иметь вид.
- ch *request* (integer, <описание_запроса_клиента>);
- ch *reply* (integer, <описание_результата>);
- % <описание_запроса_клиента> - это описание типов переменных, значения которых специфицируют запрос клиента к производителю. Значение переменной, описанной как integer, определяет уникальный номер клиента.
- <описание_результата> - описание переменных, специфицирующих ответ производителя на запрос клиента %

- ***Producer*** **::** var ***index*****, ...;**
- **do *true*** → receive ***request*** **(*index*, ...); % ожидание запроса клиента%**
- **..... % обработка запроса клиента и формирование ответа%**
- send ***reply*** **[*index*] (...); % ответ клиенту %**
- **od;**
- ***Client*** **[*i* : 1...*n*] ::** send ***request*** **(*i*, ...);**
  % обращение к *Producer* %
- receive ***reply*** **[*i*] (...); % ожидание ответа *Producer* %**
-

- В программе определены *n*+1 независимых параллельно протекающих процессов: один процесс-производитель *Producer* и *n* процессов-клиентов *Client*. Именно так могут быть описаны взаимодействия файл-сервера и программ, запрашивающих (открывающих) нужные им файлы. При этом файл-сервер ответственен за то, чтобы только одна программа получала доступ к файлу по записи. Это же управление годится для программирования удаленного доступа к дисковому файлу нескольких программ в сети. Много подобных задач в разных вариантах возникает при организации обработки данных в сети.

Реализация управления взаимодействующими процессами

Сети Петри описывали поведение процессов, но не определяли, как это поведение реализовать (запрограммировать). В языках программирования для этого разработаны многочисленные средства, в основе которых лежат операции над семафорами.

Семафоры

Семафоры являются тем базовым средством, с помощью которого можно реализовать в программе управление параллельно протекающими процессами, их синхронизацию и взаимодействия.

Семафор S есть переменная типа **integer**, которая после инициализации начального значения доступна только посредством семафорных операций *P* (от голландского слова *proberen* - проверить) и *V* (от голландского слова *verhogen* - увеличить, прирастить). Никаким другим способом значение семафорной переменной не может быть ни проверено, ни изменено.

Операции *P* и *V* определяются следующим образом:

P(*S*): **while** $S \leq 0$ **do** *skip;* *S*:=*S*-*1;*

V(*S*): *S*:=*S*+*1*;

- Каждая из операций *P* и *V* является неделимой, т.е. если семафорная переменная изменяется одной из операций, то в это время к ней нет доступа ни для какого процесса. Таким образом, изменение значения семафорной переменной (*S:=S-1* или *S:=S+1*) может делаться только одной операцией (одним процессом). В определении семафорной операции *P* оператор цикла
- **while** $S \leq 0$ **do** *skip;*
- описывает алгоритм выполнения операции *P(S)*. В реализации операции *P* нет, конечно, необходимости циклить на этом операторе. Обычно операция *P(S)* реализуется таким образом, что если в процессе при выполнении операции *P(S)* удовлетворяется условие $S \leq 0$, то процесс переводится в состояние ожидания и вновь инициируется только по выполнению операции *V(S)* в любом из процессов.

- Понятно, что при доступе к семафору тоже возможна ситуация вечного ожидания - один из процессов постоянно не получает разрешения завершить операцию *P(S)*.
- Неделимое исполнение семафорных операций в мультипроцессорах с разделяемой памятью (все процессоры работают над общей памятью) обеспечивается специальной машинной инструкцией “Проверить и изменить”. Оборудование допускает исполнение этой инструкции только одним процессором (и, следовательно, только в одном процессе). Все остальные процессоры задерживаются на время ее исполнения.

В мультикомпьютерах семафорные операции реализуются программным обеспечением.

Задача взаимного исключения

- Взаимное исключение может быть реализовано с помощью семафоров следующим образом.
- Пусть для *n* процессов *Proc(i), i=1,2, ... ,n,* должно быть обеспечено взаимное исключение при доступе к некоторому ресурсу (все равно какому). Для программирования взаимного исключения используется семафорная переменная *mutex* (mutual exclusion). Тогда структура программы *i*-го процесса такова:

- **var** *mutex*=1: **semaphore;**
- /* семафорная переменная *mutex* инициируется со значением 1 */
- *Proc(i)* :
- **repeat**
- . . . /* начальная часть программы процесса*/
- *P(mutex)*
- . . . /*критический интервал*/
- *V(mutex)*
- . . . /* заключительная часть программы процесса */
- **until** *false*

Программу процесса составляют операторы языка программирования, заключенные между операторами **repeat** и **until**. Логически программа процесса делится на три части. Участок программы процесса между операторами *P(mutex)* и *V(mutex)* называется ***критическим интервалом***. Здесь выполняются те вычисления, ради которых затевалось взаимное исключение. Критический интервал “охраняется” семафором *mutex* от влияния других процессов. Действительно, если инициализировать семафорную переменную *mutex* значением 1, то только один процесс будет в состоянии выполнить операцию *P* и войти в свой критический интервал. Все остальные процессы будут ожидать на операторе **while** *mutex* ≤ 0 **do** *skip;*

- Завершив выполнение критического интервала, процесс выполнит оператор *V(mutex)* и увеличит значение семафорной переменной *mutex* на единицу. Следовательно, один из ожидающих процессов (неизвестно какой) получит право войти в свой критический интервал.
- Если инициировать семафорную переменную со значением, например, 3, то тогда 3 процесса получат право одновременно выполнять свои критические интервалы.

Задача производитель/потребитель с ограниченным буфером

Накопительный буфер имеет два пограничных состояния, ограничивающих активность процессов-потребителей и процессов-производителей:

- *буфер пуст* - процессы-потребители должны ждать
- и
- *буфер полон* - процессы-производители должны ждать.

Заведем соответственно два семафора для описания этих состояний:

b_empty - содержит количество свободных позиций для размещения новых элементов данных в буфере, инициализируется значением *n*, и

b_full - содержит количество элементов данных в буфере, инициализируется значением 0.

Еще один семафор - *mutex* - обеспечивает взаимное исключение процессов.

var *b_empty=n, b_full=0, mutex=1*: **semaphore; . . .**

var *full=0, empty=n; /*счетчики элементов и свободных мест . Producer||Consumer;*

–/* оператор || разрешает параллельное исполнение процессов *Producer* и *Consumer**/

- *Producer*:
- {**repeat**
- ...
- производится очередной элемент данных
- ...
- *P(b_empty); /*число работающих процессов-производителей не должно превышать числа свободных мест в буфере* /*
- *P(mutex);*
- ...
- добавляется вновь произведенный элемент данных в буфер
- ...
- *V(mutex);*
- *V(b_full);*
- **until** false; };

- *Consumer:*
- { **repeat**
- *P(b_full); /*число работающих процессов-потребителей не должно превышать числа произведенных элементов в буфере*/*
- *P(mutex);*
- ...
- элемент данных забирается из буфера
- ...
- *V(mutex);*
- *V(b_empty);*
- ...
- **until** *false;* }

Задача читатели-писатели

Рассмотрим более сложный пример решения задачи программирования взаимодействия множества процессов с использованием семафоров. Пусть выполняются *n* процессов, которые разделяют некоторую переменную программы. Часть процессов - *писатели* - только модифицируют разделяемый объект, другие - *читатели* - только считывают значение. Необходимо организовать корректное выполнение этой системы взаимодействующих процессов.

Прежде всего отметим, что процессы-читатели могут иметь одновременный доступ к разделяемому объекту, т.к. чтение не меняет значение объекта и, следовательно, процессы-читатели не могут влиять друг на друга. Для процессов-писателей следует организовать взаимное исключение при доступе к разделяемому объекту.

Далее, процессы-писатели должны иметь преимущество при доступе к разделяемому объекту, поскольку они готовы записать новые данные. Следовательно, процессы-читатели должны получать доступ к разделяемому объекту лишь в том случае, если нет процессов-писателей, желающих получить доступ к этому разделяемому объекту. А потому, если процесс-писатель изъявил желание получить доступ к разделяемому объекту, то все процессы-читатели, которые в этот момент времени тоже желают получить доступ к разделяемому объекту, должны быть задержаны. С другой стороны, процессы-читатели нехорошо совершенно дискриминировать и потому процессы-читатели, получившие доступ к разделяемому объекту до того, как появился процесс-писатель, должны иметь возможность завершить чтение.

Эта модельная задача может реально интерпретироваться. Такая схема взаимодействий должна быть реализована, если нужно, к примеру, сделать Интернет-газету, которую любой читатель может читать, а репортеры (процессы-писатели) могут в любой момент, по мере поступления новой информации, её обновлять

Для программирования такого взаимодействия будут использованы 4 семафорные переменные *Mutex_n_writers, Mutex_n_readers, NoWriter* и No*Reader* и счетчики процессов-читателей *n_readers* и процессов-писателей n_writers.

/* Вначале описываются общие (глобальные) переменные множества процессов*/
var *Mutex_n_writers:=1, Mutex_n_readers:=1, NoWriters:=1, NoReaders:=1*
semaphore;
var *n_readers*:=0, n_writers:=0 **integer**;

Writer: {	*Reader*: {
/*Если стартовал хотя бы один процесс-писатель, то первым делом следует позаботиться о блокировании всех процессов-читателей, которые стартуют после него */	Если стартовал читатель и есть писатель, то читатель должен ждать. - Если нет писателя, то читатель продолжает и сначала должен остановить писателей, пришедших позже.

P(Mutex_n_writers);
n_writers++;
/*стартовал очередной процесс-писатель*/
V(Mutex_n_writers);
/*произвольные действия, не затрагивающие разделяемые переменные*/
	var *wait* **bool;**
	do { *P(NoWriters)*;
	wait = n_writers!=0;
	if *(wait) V(NoWriters)*;}
	while (wait);
	P(Mutex_n_readers);
	n_readers++;
	if *(n_readers==1) P(NoReaders);*
	V(Mutex_n_readers);
	V(NoWriters);

/*Если нет работающих процессов-писателей, то все вновь стартовавшие процессы-читатели получают беспрепятственный доступ к разделяемому ресурсу. А все позднее стартовавшие процессы-писатели должны подождать завершения уже работающих процессов-читателей

/*Теперь процессы могут работать по своим программам и в некоторый момент придут в точку доступа к разделяемому ресурсу*/

P(NoWriters); … … *P(NoReaders);* /* осуществляется запись */ *V(NoReaders)*; *V(NoWriters)*;	. . . /* осуществляется чтение */

/* Далее может идти некоторый другой фрагмент программы процесса и наконец его завершающая часть */

```
P(Mutex_n_writers);
n_writers--;
/*процесс-писатель завершился*/
V(Mutex_n_writers);
}
```

```
P(Mutex_n_readers);
n_readers--;
if (n_readers==0) V(NoReaders);
V(Mutex_n_readers);
/*если завершился последний процесс-читатель, то надо разрешить работать процессам-писателям*/
}
```

- В программе не рассматриваются случаи аварийного завершения. Например, если последний процесс-читатель или последний процесс-писатель завершится аварийно и не откроет барьерный семафор (*V*(*Mutex_n_writers*) *и/или* *V*(*Mutex_n_readers*)), тогда система процессов, конечно, «зависнет».

Критические интервалы

Программируя межпроцессные взаимодействия с использованием семафоров легко допустить разнообразные неочевидные ошибки, в первую очередь ошибки временные, связанные с доступом к разделяемым данным.

В языках программирования вместо семафоров используются другие конструкции более высокого уровня. Одна из них - критические интервалы. Ее идея такова.

Вводится понятие *разделяемой* переменной, которая доступна из нескольких процессов.

Например: **var** x : **shared real;**

Разделяемые переменные доступны только внутри оператора **region** вида **region** x **do** S;

Только один процесс может исполнять оператор **region** с переменной *x* в качестве параметра. Таким образом, пока исполняется оператор *S* никакой другой процесс не может начать исполнение оператора **region** *x*. Понятно, что оператор **region** *x* реализуется программой

var *x* : **semaphore**;

P(*x*) *S*; … *V*(*x*)

Программировать с использованием оператора критического интервала несколько легче, однако в дедлок тоже легко попасть, например:

*P*1*:* **region** *x* **do** (**region** *y* **do** *S*1);

*P*2*:* **region** *y* **do** (**region** *x* **do** *S*2);

Очевидна возможность дедлока как следствие дозахвата ресурса, когда два процесса *P*1 и *P*2 одновременно начнут выполнять свои вложенные операторы **region**.

Параллельная программа разделения множеств

Определения MPI очевидны и просты. Так же просты и очевидны средства программирования в MPI. К сожалению, их простота и очевидность кажущиеся.

- В качестве примера рассмотрим программу разделения множеств, разработанную Э.Дейкстрой. Она много обсуждалась в публикациях, ее частичная корректность была доказана разными авторами, однако лишь недавно было формально доказано отсутствие свойства тотальной корректности программы.
- Программа называется *корректной*, если при остановке она вырабатывает правильный результат. Программа называется *тотально корректной,* если она всегда останавливается и всегда вырабатывает правильный результат. Корректные, но не тотально корректные программы исключительно опасны. Они создают видимость правильной работы и, как правило, отказываются работать в самый нужный момент.

- И это при том, что программа совершенно тривиальна, в ней попросту не на что смотреть! Это хороший пример, показывающий, что простое тестирование, без формального обоснования правильности программы, не в состоянии обеспечить правильность программы.
- Но и применение одних лишь формальных методов не дает хороших результатов по той причине, что правильно применить формальный метод почти столь же трудно, как и разработать сам формальный метод.
-

- На практике следует комбинировать и тестирование и формальное доказательство правильности. Следует также обратить внимание на опасную кажущуюся правильность корректных, но не тотально корректных программ. Нередко такая программа долгое время нормально работает и обнаруживает ошибку в самый неподходящий момент. И большие сложные, особенно управляющие, программы отлаживаются иной раз десятилетиями и при этом успешно эксплуатируются.
- Пусть заданы два множества натуральных чисел S и T. Сохраняя мощность множеств S и T необходимо собрать в S наименьшие элементы множества $S \cup T$, а в T - наибольшие.
- Последовательный алгоритм и программа очевидны: множества S и T сливаются, затем слитое множество упорядочивается и вновь разделяются на на множества S’ и T’, удовлетворяющие условиям задачи.

Для параллельного асинхронного решения задачи используется следующий алгоритм.

-1.Определяются два параллельно протекающих процесса *Small* и *Large*.

-2.Процесс *Small* выбирает максимальный элемент в множестве S, а процесс *Large* параллельно (в то же самое время) находит минимальный элемент во множестве T.

-3.Процессы *Small* и *Large* синхронизуются и обмениваются данными: наибольшее значения множества S пересылаются процессом *Small* процессу *Large* для включения в множество T, а наименьшее значения множества T пересылаются процессом *Large* процессу *Small* для включения в множество S.

-4.Далее циклически повторяются шаги 3 и 4.

-5.Программа останавливается, когда наибольший элемент в множестве S окажется меньше либо равен наименьшего элемента в множестве T.

- По завершении программы каждый элемент множества S должен оказаться не больше любого элемента множества T, а мощности этих множеств не изменяются.

- **Программа состоит из двух параллельных процессов,**
- ***P=[Small||Large].***

Small::

mx:=max(S); α! mx;

S:=S-{mx}; β? x; S:=S∪{x};
mx:=max(S);
***[mx > x →**
α! mx; S:=S-{mx};
β? x; S:=S∪{x};
mx:=max(S);
]

stop

Large::

α?y; T:=T∪{y};
mn:=min(T); β! mn;
T:=T-{mn}; mn:=min(T);
***[mn < y →**
α? y; T:=T∪{y};mn:=min(T);
β! mn; T:=T-{mn};
mn:=min(T)
]

stop

- Программу можно прокомментировать следующим образом. Определены два процесса *Small* и *Large*. Символ || разрешает параллельное исполнение процессов *Small* и *Large*, оператор * задает циклическое исполнение (итерацию), пока истинно условие циклов. Процессы связаны однонаправленными каналами α и β. По каналу α процесс *Small* передает данные в процесс *Large*, а данные из процесса *Large* передаются в процесс *Small* по канале β.

- Оператор ! задает передачу данных (аналог оператора send), а оператор ? - их прием (аналог оператора receive). В частности, оператор α! mx в процессе *Small* задает передачу значения переменной *mx* в канал α, а оператор α?y в процессе *Large* определяет прием значения из канала α и присваивание этого значения переменной y.
- В программе *[Small||Large]* одновременное выполнение оператора α!mx в процессе Small и оператора α?y в процессе Large (их выполнение синхронизуется, т.е., выполнение одного из них в одном из процессов задерживается до тех пор, пока другой оператор не начнет выполняться в другом процессе) имеет семантику “удаленного присваивания” y:=mx. Аналогична семантика взаимодействия по каналу β.

- Обозначим S^0 и T^0 – начальные множества, а S^{Term} и T^{Term} – заключительные их значения. При правильном завершении программы ожидается выполнение соотношений (в соответствии с начальными условиями задачи):
- (C1). Объединение множеств не изменилось:
- $S^{Term} \cup T^{Term} = S^0 \cup T^0$;
- (C2). Мощности множеств сохранились:
- $|S^{Term}| = |S^0|, \ |T^{Term}| = |T^0|$;
- (C3). Каждый элемент S^{Term} не больше любого элемента T^{Term} :
- $\max(S^{Term}) \leq \min(T^{Term})$.

- Частичная корректность этой программы состоит в том, что если множества S^0 и T^0 конечны и непусты, то после нормального *завершения* программы (т.е. когда каждый из процессов выходит на свой stop) свойства (C1), (C2) и (C3) выполняются. Тотальная корректность ее состоит в том, что если множества S^0 и T^0 конечны и непусты, то программа завершается правильно и свойства (C1), (C2) и (C3) непременно выполняются после этого завершения.
- Оставляя в стороне формальные детали, весьма поучительно рассмотреть технологические приемы, приводящие к пониманию того, что программа не является тотально корректной.

4.2.3.Коммуникационно-замкнутые слои параллельной программы

Это понятие вводится для упрощения *верификации* (доказательства правильности) параллельных программ. Основная идея здесь - это разбиение каждого процесса P_i параллельной программы $P::=[P_1||\ ...||P_n]$ на последовательность его операторов: $P_i=Q_{i,1};Q_{i,2};...Q_{i,k}$ (k может быть выбрано одним и тем же для всех процессов, если допустить возможность использовать в качестве $Q_{i,r}$ пустой оператор). Таким образом, параллельная программа P может быть представлена как:

- $P=[(Q_{1,1};\ Q_{1,2};...;\ Q_{1,k})||...||(Q_{i,1};Q_{i,2};...;Q_{i,k})||\ ...\ ||(Q_{n,1};Q_{n,2};...;Q_{n,k})]$.
- Эту программу можно изобразить матрицей **(рис. 4.3)**.

P_1	$Q_{1,1}$	$Q_{1,2}$	...	$Q_{1,j}$	...	$Q_{1,k}$
	...					
P_i	$Q_{i,1}$	$Q_{i,2}$	...	$Q_{i,j}$	...	$Q_{i,k}$
	...					
P_n	$Q_{n,1}$	$Q_{n,2}$	...	$Q_{n,j}$	...	$Q_{n,k}$

- Каждая *i*-я строка изображает процесс P_i как последовательность операторов: $P_i = Q_{i,1}; Q_{i,2}; \ldots Q_{i,k}$.
- Параллельная программа $L_j = [Q_{1,j} || Q_{2,j} || \ldots || Q_{n,j}]$ называется *j*-м слоем параллельной программы *P* (*j*-й столбец матрицы).

- Слой L_j называется *коммуникационно-замкнутым*, если при всех вычислениях P взаимодействие процессов $P_1 || \ldots || P_n$ происходит только внутри этого слоя, или, иными словами, ни одна команда взаимодействия среди операторов $Q_{r,j}$ при всех выполнениях P не будет синхронизироваться (*сочетаться*) с командами взаимодействия из операторов $Q_{t,i}$ при $i \neq j$.
- Тогда последовательность слоев $L_1;\ldots;L_k$ представляет собой параллельную программу:
- $P^* = L_1;\ldots;L_k = [Q_{1,1}||Q_{2,1}||\ldots||Q_{n,1}];\ldots ;[Q_{1,k}||Q_{2,k}||\ldots||Q_{n,k}]$,
- В программе P^* все L_j исполняются последовательно в порядке перечисления, а операторы каждой L_j исполняются параллельно. Программа P^* называется *безопасной*, если и только если все ее слои коммуникационно-замкнуты.

Если программа P^* безопасна, то вместо верификацию всей параллельной программы P можно проводить ее послойную верификацию, т.е., доказывать утверждение $\{p_0\}L_1\{p_1\},\ldots,\{p_{k-1}\}L_k\{p_k\}$ вместо утверждения $\{p_0\}P\{p_k\}$. Здесь, как обычно, $\{s\}P\{q\}$ обозначает утверждение, что программа P частично корректна по отношению к предусловию s и постусловию q (вход-выходные соотношения), при этом, если до начала исполнения программы P предикат s истинен, то после исполнения P предикат q тоже истинен.

Таким образом, если параллельную программу P удается разбить на последовательность коммуникационно-замкнутых слоев, то доказательство ее (частичной) корректности сводится к последовательному доказательству вход-выходных соотношений для каждого слоя. Это существенно упрощает анализ корректности параллельной программы.

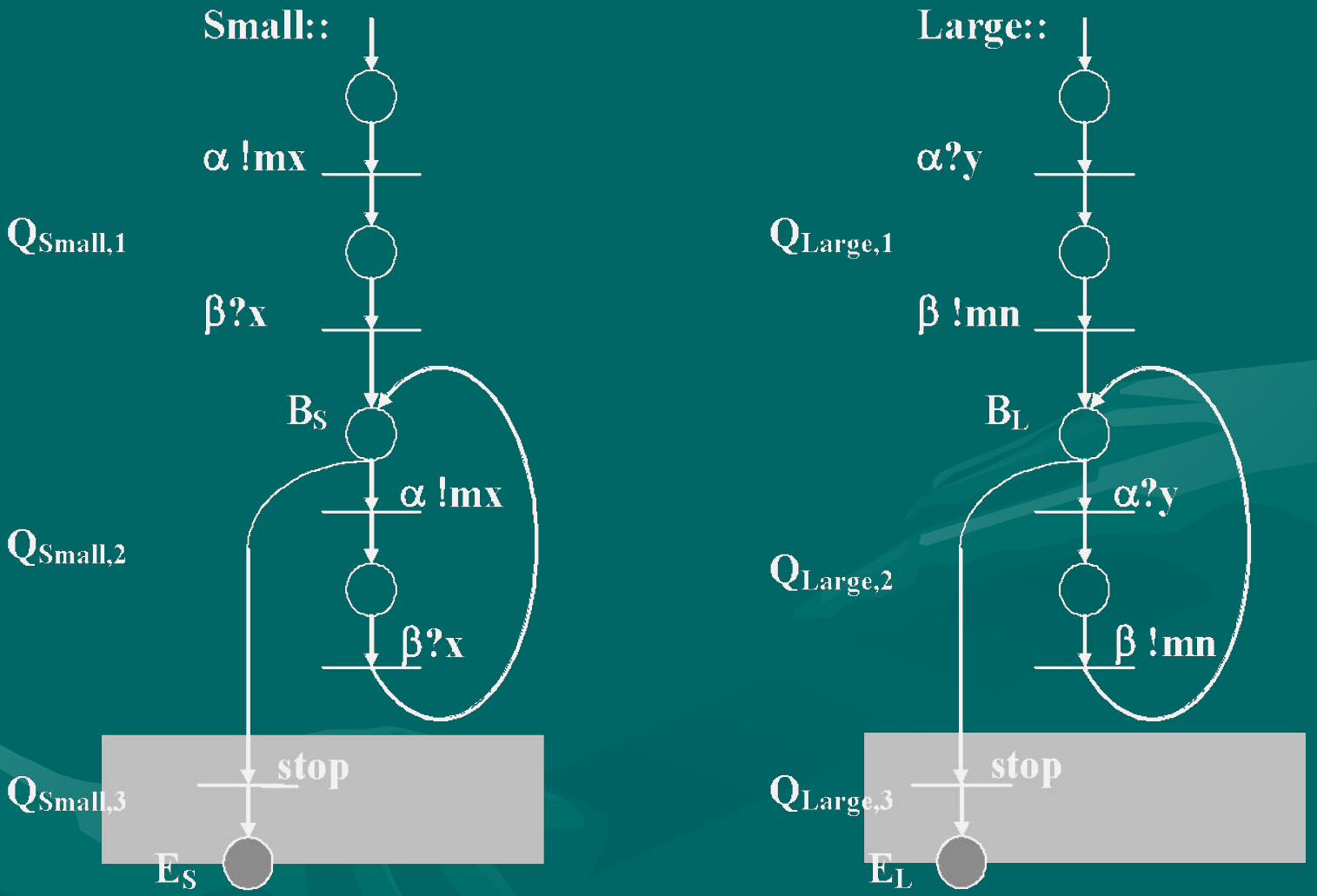

Small::
α !mx
$Q_{Small,1}$
β?x
B_S
α !mx
$Q_{Small,2}$
β?x
stop
$Q_{Small,3}$
E_S
Large::
α?y
$Q_{Large,1}$
β !mn
B_L
α?y
$Q_{Large,2}$
β !mn
stop
$Q_{Large,3}$
E_L

Рис. 4.4.

- Процессы *Small* и *Large* разбиваются на коммуникационно-замкнутые слои совершенно естественно. Разбиение приведено на рис. 4.4.
- На рисунке показаны только “синхронизационные скелеты” параллельных процессов. В первый слой входят операторы до цикла, во второй слой - операторы внутри цикла, третий слой составляет оператор stop после выхода из цикла. Процесс правильно завершается, если он заканчивает вычисления в заключительном состоянии: процесс *Small* в состоянии E_S , а процесс *Large* в состоянии E_L. В общем случае процессы могут:
- а) оба завершиться нормально,
- б) оба не завершатся,
- в)один завершится, а другой нет, например, при невозможности выполнить операцию синхронного взаимодействия.
- Условия *BS* и *BL* определяют условия окончания циклов в процессах; они равны соответственно $mx \leq x$ и $mn \geq y$.

- 4.2.4.Когерентность параллельных программ
- Для упрощения верификации параллельной программы уже при ее проектировании на нее можно наложить дополнительные требования, облегчающие ее понимание и анализ, и заранее устраняющие некоторые типы возможных ошибок. Одним из таких требований является *когерентность* параллельной программы.
- Неформально, требование когерентности означает:
- -каждый раз, когда процесс хочет послать сообщение другому (в динамике), его партнер готов принять это сообщение, т.е., обязательно выходит на оператор приема сообщения;
- -каждый раз тогда, когда процесс хочет получить сообщение некоторого типа от другого процесса, его партнер посылает ему это сообщение.

В когерентной программе невозможна ситуация, когда в одном процессе выполняется оператор α! mx, а процесс-партнер не может выйти на исполнение оператора α?y.

Если параллельная программа $P::=[P_1||...||P_n]$ разбита на коммуникационно-замкнутые слои $P_i = Q_{i,1};Q_{i,2};...Q_{i,k}$, то требование когерентности состоит не только в том, что при всех вычислениях P ни одна команда взаимодействия среди операторов $Q_{r,j}$ при всех выполнениях P не будет синхронизироваться (сочетаться) с командами взаимодействия из операторов $Q_{t,i}$ при $i \neq j$, но и в том, что каждая такая команда взаимодействия *обязательно будет сочетаться* с некоторой командой взаимодействия из операторов того же самого слоя. Для параллельной программы *P=[Small || Large]* такая синхронизация показана на рис. 4.5.

- Очевидно, что требование когерентности для этой программы выполняется тогда и только тогда, когда условия прекращения циклов в программах Small и Large тождественны при всех прохождениях циклов.
- Некогерентность, с другой стороны, ведет к блокировке этой программы, т.е., к возникновению ситуации, когда один из процессов выходит из цикла и переходит в заключительное состояние, а другой не выходит из своего цикла и “зависает” на операции синхронного взаимодействия внутри цикла, бесконечно ожидая партнера.

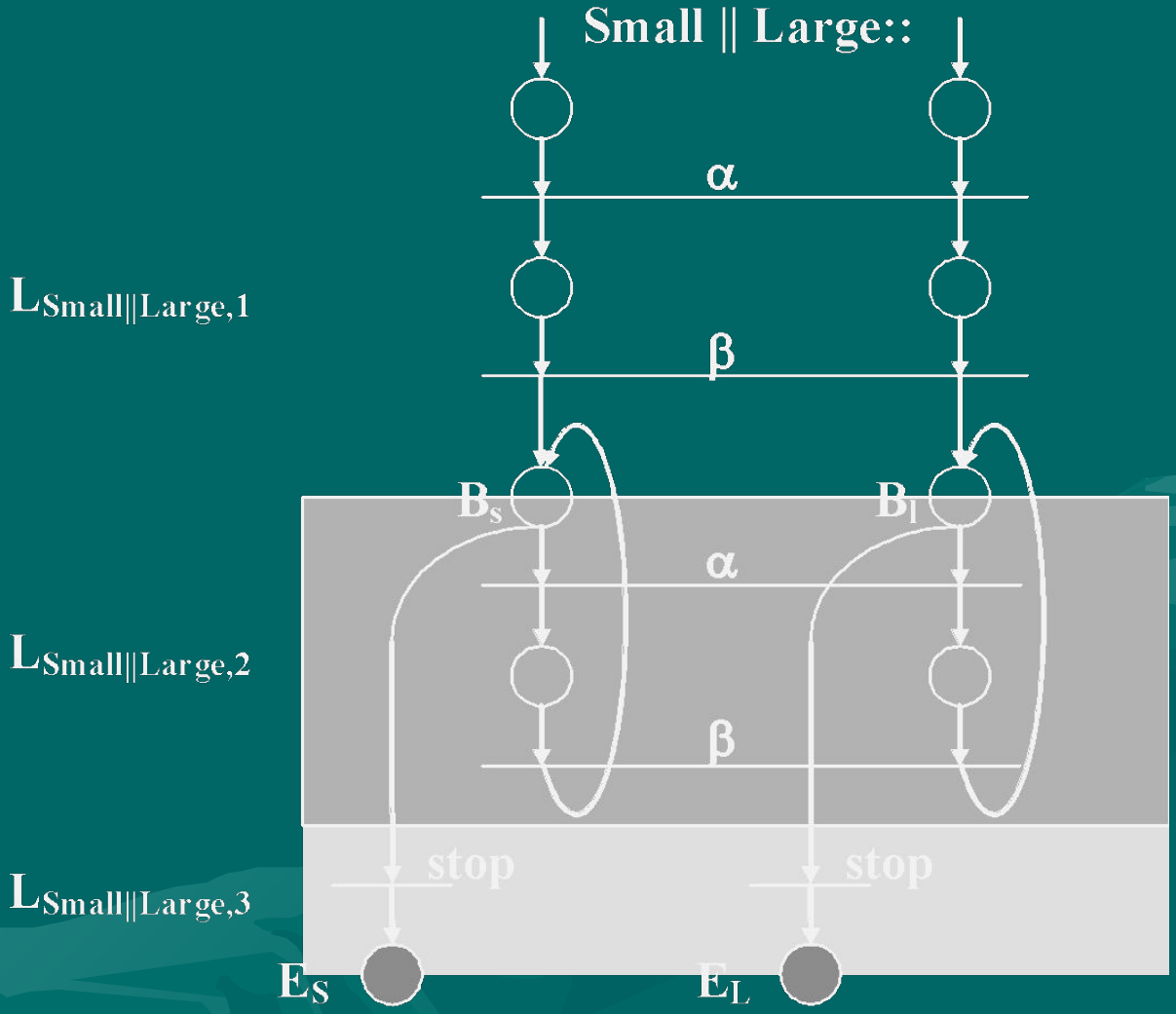

Small || Large::
α
β
$L_{Small||Large,1}$
B_s
B_l
α
$L_{Small||Large,2}$
β
stop
stop
$L_{Small||Large,3}$
E_S
E_L

Рис. 4.5.

- Таким образом, условие $B_S \equiv B_L \equiv И$, или, что то же, $mx \leq x \equiv mn \geq y$ при *каждой* проверке условий циклов в обоих программах, является необходимым условием тотальной корректности этой параллельной программы.

4.2.5.Анализ программы разделения множеств

- Для анализа программы используем “истории” взаимодействий. Вводятся вспомогательные переменные, которые хранят истории взаимодействия по каждому каналу программы.
- *Историческая переменная* - это просто массив значений, последовательно переданных по соответствующему каналу. Пусть h_α и h_β такие исторические переменные для каналов α и β соответственно, тогда компонент $h_\alpha[k]$ содержит k-е значение, посланное по каналу α при выполнении операции α!e.

- **Проведем анализ первого слоя :**

$Q_{Small,1}$::

mx:=max(S); α! mx; S:=S-{mx};
β? x; S:=S ∪ {x};
mx:=max(S)

$Q_{Large,1}$::

α?y; T:=T ∪ {y}; mn:=min(T);
β! mn; T:=T-{mn};
mn:=min(T)

- Для того, чтобы процессы QSmall,1 и QLarge,1 завершились, необходимо и достаточно, чтобы множество S содержало хотя бы один элемент, т.е. $|S|>0$. По завершении каждого из этих параллельных процессов первого слоя будут справедливы следующие соотношения:

для $Q_{Small,1}$:

$mx^0 = max(S^0)$;

$h_\alpha[0] = mx^0$;

$x^1 = h_\beta[0]$;

$S^1=(S^0-\{max(S^0)\}) \cup \{x^1\}$;

$mx^1 = max(S^1)$;

для $Q_{Large,1}$:

$y^1 = h_\alpha[0]$;

$mn^0 = min(T^0 \cup \{ y^1 \})$;

$h_\beta[0] = mn^0$;

$T^1=T^0 \cup \{y^1\}-\{min(T^0 \cup \{y^1\})\}$;

$mn^1 = min(T^1)$;

- Эти соотношения просто описывают, что было сделано при исполнении операторов первого слоя.

- Рассмотрим теперь второй слой:

$Q_{Small,2}$::

*[mx > x →
 α! mx; S:=S-{mx};
 β? x; S:=S∪{x}; mx:=max(S);
]

$Q_{Large,2}$::

*[mn < y →
 α? y; T:=T∪{y}; mn:=min(T);
 β! mn; T:=T-{mn}; mn:=min(T)
]

- Перед *i*-м выполнением каждого цикла для процессов $Q_{Small,2}$ и $Q_{Large,2}$ истинны следующие инварианты, что можно проверить непосредственно (где a^i -значение переменной *a* перед *i*-ым выполнением цикла):

для $Q_{Small,2}$:

$I_{Small,2} \equiv h_\alpha[i-1] = mx^{i-1} \wedge$
$x^i = h_\beta[i-1] \wedge$
$S^i=(S^{i-1}-\{max(S^{i-1})\}) \cup \{x^i\} \wedge$
$mx^i= max(S^i);$

для $Q_{Large,2}$:

$I_{Large,2} \equiv y^i = h_\alpha[i-1] \wedge$
$h_\beta[i-1] = min(T^{i-1} \cup \{y^i\}) \wedge$
$T^i=T^{i-1} \cup \{y^{i-1}\}-\{min(T^{i-1} \cup \{y^i\})\} \wedge$
$mn^i= min(T^i);$

- Как уже говорилось, требование когерентности в этой программе соответствует требованию общезначимости формулы $mx^i \le x^i \equiv mn^i \ge y^i$. При некоторых значениях исходных множеств S и T она нарушается. Возможны два случая некогерентности:

а) $mx^i \le x^i$; или, что то же: $\max(S^i) \le \min(T^{i-1} \cup \{ \max(S^{i-1})\})$;
$mn^i < y^i$; $\min(T^i) < \max(S^{i-1})$;

- при этом процесс *Small* завершается, а процесс *Large* продолжает выполнять цикл, что приводит к его блокировке;

б) $mx^i > x^i$; или, что то же: $\max(S^i) > \min(T^{i-1} \cup \{ \max(S^{i-1})\})$;
$mn^i \ge y^i$; $\min(T^i) \ge \max(S^{i-1})$;

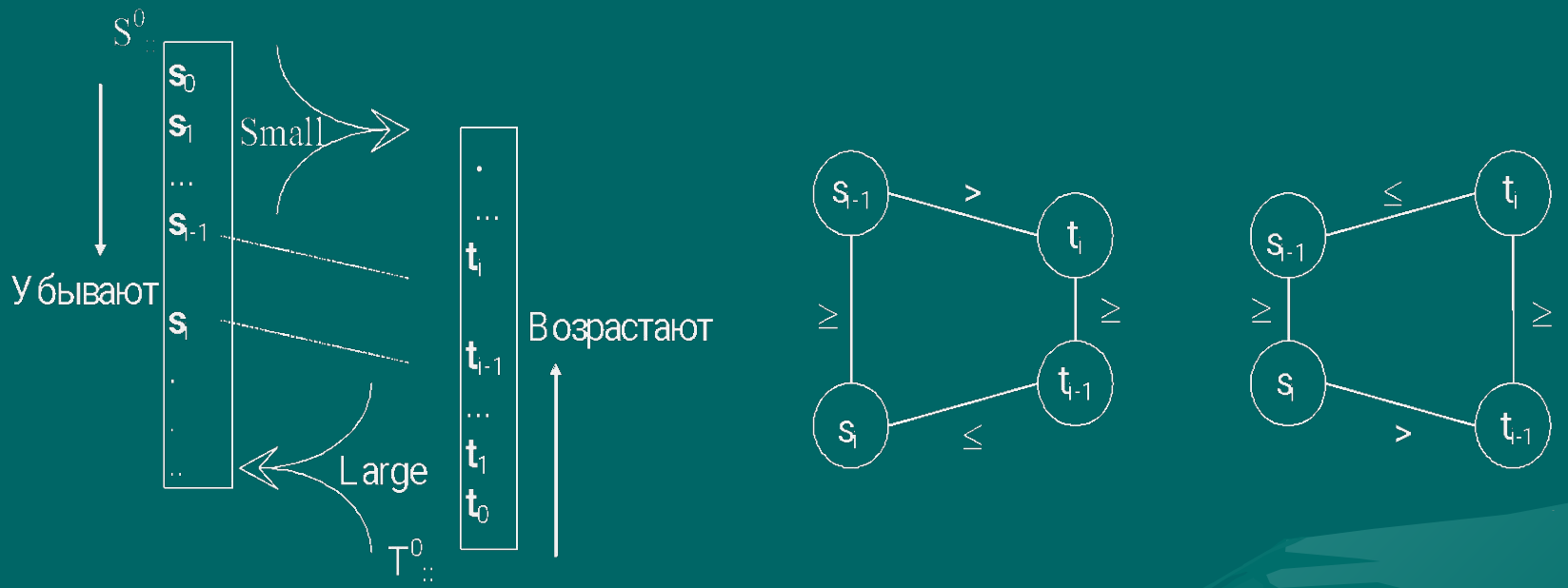
- при этом процесс *Large* завершается, а процесс *Small* продолжает выполнять цикл и блокируется, бесконечно ожидая взаимодействия с процессом *Large*.

- **Учитывая, что $\min(T^i) \geq \min(T^{i-1})$ и $\max(S^i) \leq \max(S^{i-1})$, условия некорректного поведения параллельной программы разделения множеств можно записать проще:**

а) $\max(S^i) \leq \min(T^{i-1})$;
$\min(T^i) < \max(S^{i-1})$;

б) $\max(S^i) > \min(T^{i-1})$;
$\min(T^i) \geq \max(S^{i-1})$.

- Поясним полученный результат. Исследуемая параллельная программа разделения множеств имеет целью собрать все минимальные элементы объединения двух множеств S и T в множестве S, а максимальные элементы S ∪ T - в множестве T, причем мощности множеств не должны измениться. Упорядочим элементы исходных множеств: множества S по убыванию, а множества T по возрастанию. На рис. 4.6а. показано, что процесс *Small*, работая на множестве S, пересылает его максимальные элементы в множество T, а процесс *Large*, работая на множестве T, пересылает его минимальные элементы в множество S.



- **Рис. 4.6.**

- **Полученные выше условия некорректного поведения программы определены для (i-1)-го и i-го максимальных значений множества S и для (i-1)-го и i-го минимальных значений множества T, (i =1,2, ...). Эти условия представлены диаграммами на рис. 4.6.б) и 4.6.в).**

- **Иными словами, если между упорядоченными по убыванию элементами множества S и упорядоченными по возрастанию элементами множества T на одном и том же расстоянии от начала выполнится одно из отношений рис. 4.6,б) или рис. 4.6,в), то исследуемая программа будет работать некорректно: она входит в дедлок.**

- Можно указать тестовый пример, на котором эта программа работает некорректно: S={5,10,15,20}, T={17,18,30,40,60}. Этот пример относится к первому типу некорректностей при i=1: первое же вхождение процесса Large в цикл приводит к дедлоку, поскольку Small завершится, не входя в цикл. Однако, полагаться на возможность обнаружения этой некорректности с помощью тестов нельзя. Добавление, например, к множеству T любого числа элементов, меньших 15, нарушит это соотношение и программа будет работать корректно.
- Ручной прокруткой можно проверить, что на множествах S={5,10,15,20}, T={14,17,18,30,40,60} программа работает правильно: сортирует множества и завершается после однократного прохождения циклов в обоих процессах.