ГЛАВА 2
ОСНОВЫ ПАРАЛЛЕЛЬНОГО ПРОГРАММИРОВАНИЯ

2.1. Основные понятия

Основой для разработки параллельных программ, реализующих параллельные методы решения задач, является понятия *процесса* и *потока*. Рассмотрение программы в виде набора процессов/потоков, выполняемых параллельно на разных процессорах или на одном процессоре в режиме разделения времени, позволяет сконцентрироваться на рассмотрении проблем организации *взаимодействия параллельных частей программы*, определить моменты и способы обеспечения с*инхронизации и взаимоисключения процессов/потоков*, изучить условия возникновения или доказать отсутствие *тупиков* в ходе выполнения программ (ситуаций, в которых все или часть параллельных участков программы не могут быть выполнены при любых вариантах продолжения вычислений).

2.1.1. Концепция процесса

Понятие *процесса* является одним из основополагающих в теории и практике параллельного программирования. В научно-технической литературе дается ряд определений процесса, но в целом большинство определений сводится к пониманию процесса как «*некоторой последовательности команд, претендующей наравне с другими процессами программы на использование процессора для своего выполнения*».

Конкретизация понятия процесса зависит от целей исследования параллельных программ. Для анализа проблем организации взаимодействия процессов можно рассматривать процесс как последовательность команд

$$p_n = (i_1, i_2, ..., i_n)$$

(для простоты изложения материала будем предполагать, что процесс описывается единственной командной последовательностью). Динамика развития процесса определяется моментами времен начала выполнения команд

$$t(p_n) = t_p = (\tau_1, \tau_2, ..., \tau_n),$$

где τ_j, $1 \le j \le n$, есть время начала выполнения команды j. Последовательность t_p представляет временну́ю *траекторию* развития процесса. Предполагая, что команды процесса исполняются строго последовательно, в ходе своей реализации не могут быть приостановлены (т. е. являются неделимыми) и имеют одинаковую длительность выполнения, равную 1 (в тех или иных временных единицах), получим, что моменты времени траектории процесса должны удовлетворять соотношениям

$$\forall i,\ 1 \le i < n \quad \Rightarrow \quad \tau_{i+1} \ge \tau_i + 1.$$

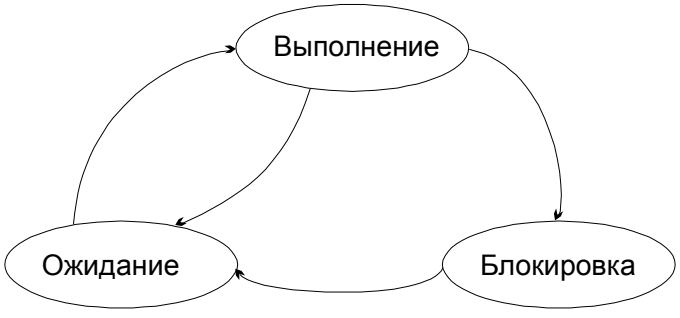

Рис. 2.1. Диаграмма переходов процесса из состояния в состояние

Равенство $\tau_{i+1} = \tau_i + 1$ достигается, если для выполнения процесса выделен процессор и после завершения очередной команды процесса сразу же начинается выполнение следующей команды. В этом случае говорят, что процесс является *активным* и находится в *состоянии выполнения*. Соотношение $\tau_{i+1} > \tau_i + 1$ означает, что после выполнения очередной команды процесс *приостановлен* и ожидает возможности для своего продолжения. Данная приостановка может быть вызвана необходимостью разделения использования единственного процессора между одновременно исполняемыми процессами. В этом случае приостановленный процесс находится в *состоянии ожидания* момента предоставления процессора для своего выполнения. Кроме того, приостановка процесса может быть вызвана и временной неготовностью процесса к дальнейшему выполнению (например, процесс может быть продолжен только после завершения операции ввода-вывода данных). В подобных ситуациях говорят, что процесс является *блокированным* и находится в *состоянии блокировки*.

В ходе своего выполнения состояние процесса может многократно изменяться; возможные варианты смены состояний показаны на диаграмме переходов рис. 2.1.

2.1.2. Определение потока

Как уже отмечалось ранее, понятие процесса является чрезвычайно важным для организации вычислений на ЭВМ. Выполняемые программы с точки зрения операционной системы представляют собой процессы, которые параллельно выполняются, взаимодействуют между собой и конкурируют за использование процессоров вычислительной системы. Вместе с этим, понятие процесса является несколько «тяжеловесным» – создание процесса, переключение процессоров на использование других процессов и выполнение других подобных действий занимает достаточно много процессорного времени. Кроме того, процессы выполняются в разных адресных пространствах и, как результат, организация их взаимодействия требует определенных усилий.

Все выше отмеченные моменты приводят к необходимости определения *потока* (*thread*) как более простой альтернативы понятию процесса. Поток (точно так же, как и процесс) можно представлять как последовательность команд программы, которая может претендовать на использование процессора вычислительной системы для своего выполнения. Однако – в отличие от процесса – потоки одной и той же программы работают в общем адресном пространстве и, тем самым, разделяют данные программы. Следует отметить, что общность данных, с одной стороны, существенно упрощает организацию взаимодействия потоков (результат, вычисленный одним потоком, сразу становится доступным всем остальным потокам программы), но, с другой стороны, требует соблюдения определенных правил использования разделяемых данных – данный аспект обсуждается далее в п. 2.1.4.

Процессы и потоки в равной степени используются для организации вычислений. Процессы применяются для представления отдельных программ (заданий для операционной системы) и для формирования многопроцессных программ, исполняемых на вычислительных системах с распределенной памятью. Потоки используются для представления программ как множества частей (потоков), которые могут выполняться независимо друг от друга (параллельно).

Дополнительная информация по более детальному представлению процессов и потоков может быть получена в [5,27,31,33].

2.1.3. Понятие ресурса

Понятие *ресурса* обычно используется для обозначения любых объектов вычислительной системы, которые могут быть использованы процессом для своего выполнения. В качестве ресурса может рассматриваться процесс, память, программы, данные и т. п. По характеру использования могут различаться следующие категории ресурсов:

– *выделяемые* (монопольно используемые, неперераспределяемые) ресурсы характеризуются тем, что выделяются процессам в момент их возникновения и освобождаются только в момент завершения процессов; в качестве такого ресурса может рассматриваться, например, устройство чтения на магнитных лентах;

– *повторно распределяемые ресурсы* отличаются возможностью динамического запрашивания, выделения и освобождения в ходе выполнения процессов (таковым ресурсом является, например, оперативная память);

– *разделяемые* ресурсы, особенность которых состоит в том, что они постоянно остаются в общем использовании и выделяются процессам для использования в режиме разделения времени (как, например, процессор, разделяемые файлы и т. п.);

– *многократно используемые* (реентерабельные) ресурсы отличаются возможностью одновременного использования несколькими процессами (что может быть обеспечено, например, при неизменяемости ресурса при его использовании; в качестве примеров таких ресурсов могут рассматриваться реентерабельные программы, файлы, используемые только для чтения и т. д.).

Следует отметить, что тип ресурса определяется не только его конкретными характеристиками, но и зависит от применяемого способа использования. Например, оперативная память может рассматриваться как повторно распределяемый, так и разделяемый ресурс; использование программ может быть организовано в виде ресурса любого рассмотренного типа.

2.1.4. Организация параллельных программ как системы потоков

Ориентируясь в данном учебном издании на проблемы разработки параллельных программ для вычислительных систем с общей памятью, далее все рассматриваемые вопросы будут даваться на примере потоков. Подобный подход позволяет снизить сложность изучения излагаемого учебного материала, но при этом следует понимать, что данный материал является общим для параллельного программирования в целом.

Итак, понятие потока может быть использовано в качестве основного *конструктивного элемента* для построения параллельных программ в виде совокупности *взаимодействующих потоков*. Такая агрегация программы позволяет получить более компактные (поддающиеся анализу) вычислительные схемы реализуемых методов, скрыть при выборе способов распараллеливания несущественные детали программной реализации, обеспечивает концентрацию усилий на решение основных проблем параллельного функционирования программ.

Существование нескольких одновременно выполняемых потоков приводит к появлению дополнительных соотношений, которые должны выполняться для величин временных траекторий потоков. Возможные типовые варианты таких соотношений на примере двух потоков p и q состоят в следующем (см. рис. 2.2):

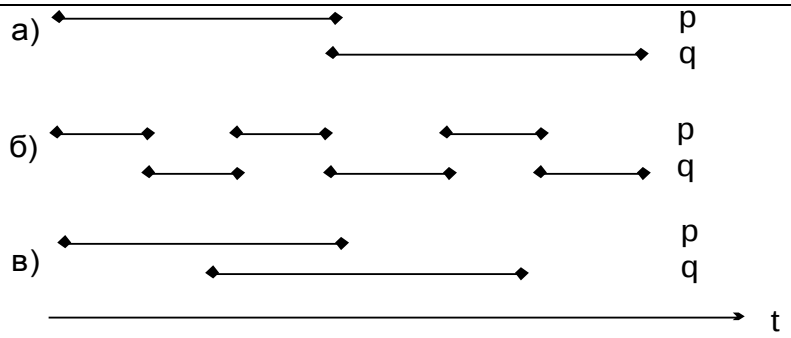

Рис. 2.2. Варианты взаиморасположения траекторий одновременно исполняемых потоков (отрезки линий изображают фрагменты командных последовательностей потоков)

– выполнение потоков осуществляется строго последовательно, т. е. поток *q* начинает свое выполнение только после полного завершения потока *p* (*однопрограммный режим* работы ЭВМ – см. рис. 2.2а),

– выполнение потоков может осуществляться одновременно, но в каждый момент времени могут исполняться команды только какого-либо одного потока (*режим разделения времени* или *многопрограммный режим* работы ЭВМ – см. рис. 2.2б),

– *параллельное выполнение* потоков, когда одновременно могут выполняться команды нескольких потоков (данный режим исполнения потоков осуществим только при наличии в вычислительной системе нескольких процессоров – см. рис. 2.2в).

Приведенные варианты взаиморасположения траекторий потоков определяются не требованиями необходимых функциональных взаимодействий потоков, а являются лишь следствием технической реализации одновременной работы нескольких потоков. С другой стороны, возможность чередования по времени командных последовательностей разных потоков следует учитывать при разработке многопоточной программы. Рассмотрим для примера два потока с идентичным программным кодом.

Поток 1	**Поток 2**
N = N +1	N = N +1
печать N	печать N

Пусть начальное значение переменной *N* равно 1. Тогда при последовательном исполнении поток 1 напечатает значение 2, а поток 2 – значение 3. Однако возможна и другая последовательность исполнения потоков в режиме разделения времени (с учетом того, что сложение *N* = *N* +1 выполняется при помощи нескольких машинных команд)

Время	**Поток 1**	**Поток 2**
1	Чтение *N* (1)	
2		Чтение *N* (1)
3		Прибавление 1 (2)
4	Прибавление 1 (2)	
5	Запись *N* (2)	
6	Печать *N* (2)	
7		Запись *N* (2)
8		Печать *N* (2)

(в скобках для каждой команды указывается значение переменной *N*). Как следует из приведенного примера, результат одновременного выполнения нескольких потоков, если не предпринимать специальных мер, может зависеть от порядка исполнения команд. Ситуа-

ция, когда два или более потоков используют разделяемый ресурс и конечный результат зависит от соотношения скоростей потоков, называется *состязанием* или *гонками* (*race conditions*).

Выполним анализ возможных командных последовательностей, которые могут получаться для программ, образованных в виде набора потоков. Рассмотрим для простоты два потока

$$p_n = (i_1, i_2, \cdots, i_n),\ q_m = (j_1, j_2, \cdots, j_m).$$

Командная последовательность программы образуется чередованием команд отдельных потоков и тем самым имеет вид:

$$r_s = (l_1, l_2, \cdots, l_s),\ s = n + m.$$

Фиксация способа образования последовательности r_s из команд отдельных потоков может быть обеспечена при помощи характеристического вектора

$$x_s = (\chi_1, \chi_2, \cdots, \chi_s),$$

в котором следует положить $\chi_k = p_n$, если команда l_k получена из потока p_n (иначе $\chi_k = q_m$). Порядок следования команд потоков в r_s должен соответствовать порядку расположения этих команд в исходных потоках

$$\forall u, v : (u < v), (\chi_u = \chi_v = p_n) \Rightarrow p_n(l_u) < p_n(l_v),$$

где $p_n(l_k)$ есть команда потока p_n, соответствующая команде l_k в r_s.

С учетом введенных обозначений, под программой, образованной из потоков p_n и q_m, можно понимать множество всех возможных командных последовательностей

$$R_s = \{< r_s, x_s >\}.$$

Данный подход позволяет рассматривать программу так же, как некоторый обобщенный (*агрегированный*) поток, получаемый путем параллельного объединения составляющих потоков

$$R_s = p_n \otimes q_m.$$

Выделенные особенности одновременного выполнения нескольких потоков могут быть сформулированы в виде ряда **принципиальных положений**, которые должны учитываться при разработке параллельных программ:

– моменты выполнения командных последовательностей разных потоков могут чередоваться по времени;

– между моментами исполнения команд разных потоков могут выполняться различные временные соотношения (отношения следования); характер этих соотношений зависит от количества и быстродействия процессоров и загрузки вычислительной системы и, тем самым, не может быть определен заранее;

– временные соотношения между моментами исполнения команд могут различаться при разных запусках программ на выполнение, т. е. одной и той же программе при одних и тех же исходных данных могут соответствовать разные командные последовательности вследствие разных вариантов чередования моментов работы разных потоков;

– доказательство правильности получаемых результатов должно проводиться для любых возможных временных соотношений для элементов временных траекторий потоков;

– для исключения зависимости результатов выполнения программы от порядка чередования команд разных потоков необходим анализ ситуаций взаимовлияния потоков и разработка методов для их исключения.

Перечисленные моменты свидетельствуют о *существенном повышении сложности параллельного программирования* по сравнению с разработкой «традиционных» последовательных программ.

В завершение следует отметить, что в самом общем случае параллельная программа может представлять собой набор процессов, каждый из которых может состоять из нескольких потов.

2.2. Взаимодействие и взаимоисключение потоков

Одной из причин зависимости результатов выполнения программ от порядка чередования команд может быть разделение одних и тех же данных между одновременно исполняемыми потоками (например, как это осуществляется в выше рассмотренном примере). Данная ситуация может рассматриваться как проявление общей *проблемы использования разделяемых ресурсов* (общих данных, файлов, устройств и т. п.). Для организации разделения ресурсов между несколькими потоками необходимо иметь возможность:

– определения доступности запрашиваемых ресурсов (ресурс свободен и может быть выделен для использования, ресурс уже занят одним из потоков программы и не может использоваться дополнительно каким-либо другим потоком);

– выделения свободного ресурса одному из потоков, запросивших ресурс для использования;

– приостановки (*блокировки*) потоков, выдавших запросы на ресурсы, занятые другими потоками.

Главное требование к механизмам разделения ресурсов является гарантированное обеспечение *использования каждого разделяемого ресурса только одним потоком* от момента выделения ресурса этому потоку до момента освобождения ресурса. Данное требование в литературе обычно именуется *взаимоисключением потоков* (*mutual exclusion*); командные последовательности потоков, в ходе которых поток использует ресурс на условиях взаимоисключения, называется *критической секцией* потока. С использованием последнего понятия условие взаимоисключения потоков может быть сформулировано как требование *нахождения в критических секциях по использованию одного и того же разделяемого ресурса не более чем одного потока*.

Требование взаимоисключения не является единственным к способам организации критических секций; дополнительный перечень необходимых свойств состоит в следующем:

– *Отсутствие взаимной блокировки*. Потоки (по отдельности или совместно) не могут мешать каким-либо потокам обращаться к выполнению своих критических секций.

– *Эффективность*. При наличии нескольких потоков, пытающихся начать выполнение своих критических секций, выбор единственного потока для продолжения осуществляется за конечное (малое) время.

– *Отсутствие бесконечного ожидания*. Поток, пытающийся начать выполнение своей критической секции, гарантированно должен когда-либо получить такую возможность.

При разработке способов обеспечения критических секций обычно предполагается также, что относительные скорости выполнения потоков неизвестны и произвольны, а длительность нахождения потоков в своих критических секциях является конечной.

2.2.1. Разработка алгоритма взаимоисключения

Рассмотрим несколько вариантов программного решения проблемы взаимоисключения (для записи программ используется псевдокод, близкий к языку программирования C++). В каждом из вариантов будет предлагаться некоторый частный способ взаимоис-

ключения потоков с целью демонстрации всех возможных ситуаций при использовании общих разделяемых ресурсов. Последовательное усовершенствование механизма взаимоисключения при рассмотрении вариантов приведет к изложению *алгоритма Деккера*, обеспечивающего взаимоисключение для двух параллельных потоков. Обсуждение способов взаимоисключения будет продолжено далее в пп. 2.2.2– 2.2.3 рассмотрением *концепции семафоров и мониторов*, которые могут быть использованы для общего решение проблемы взаимоисключения любого количества взаимодействующих потоков.

Вариант 1 – Жесткая синхронизация

В первом варианте для взаимоисключения используется управляющая переменная для задания номера потока, имеющего право на использование общего разделяемого ресурса.

```
int ThreadNum=1; // номер потока для доступа к ресурсу
Thread_1() {
  while (1) {
    // повторять, пока право доступа у потока 2
    while ( ThreadNum == 2 );
    <Использование общего ресурса>
    // передача права доступа к ресурсу потоку 2
    ThreadNum = 2;
  }
}
Thread_2() {
  while (1) {
    // повторять, пока право доступа у потока 1
    while ( ThreadNum == 1 );
    < Использование общего ресурса >
    // передача права доступа к ресурсу потоку 1
    ThreadNum = 1;
  }
}
```

Реализованный в программе способ гарантирует взаимоисключение, однако такому решению присущи два существенных недостатка:

– ресурс используется потоками строго последовательно (по очереди) и, как результат, при разном темпе развития потоков общая скорость выполнения программы будет определяться наиболее медленным потоком;

– при завершении работы какого-либо потока другой поток не сможет воспользоваться ресурсом и может оказаться в постоянно блокированном состоянии.

Решение проблемы взаимоисключения подобным образом известно в литературе как способ *жесткой синхронизации*.

Вариант 2 – Потеря взаимоисключения

В данном варианте для ухода от жесткой синхронизации используются две управляющие переменные, фиксирующие использование потоками разделяемого ресурса.

```
int ResourceThread1=0; // =1 – ресурс занят потоком 1
int ResourceThread2=0; // =1 – ресурс занят потоком 2
Thread_1() {
  while (1) {
```

```
    // повторять, пока ресурс используется потоком 2
    while ( ResourceThread2 == 1 );
    ResourceThread1 = 1;
    < Использование общего ресурса >
    ResourceThread1 = 0;
  }
}
Thread_2() {
  while (1) {
    // повторять, пока ресурс используется потоком 1
    while ( ResourceThread1 == 1 );
    ResourceThread2 = 1;
    < Использование общего ресурса >
    ResourceThread2 = 0;
  }
}
```

Предложенный способ разделения ресурсов устраняет недостатки жесткой синхронизации, однако при этом *теряется гарантия взаимоисключения* – оба потока могут оказаться одновременно в своих критических секциях (это может произойти, например, при переключении между потоками в момент завершения проверки занятости ресурса). Данная проблема возникает из-за различия моментов проверки и фиксации занятости ресурса.

Следует отметить, что в отдельных случаях взаимоисключение потоков в данном примере может произойти и корректно – все определяется конкретными моментами переключения потоков. Отсюда следует два важных вывода:

– успешность однократного выполнения не может служить доказательством правильности функционирования параллельной программы даже при неизменных параметрах решаемой задачи;

– для выявления ошибочных ситуаций необходима проверка разных временных траекторий выполнения параллельных потоков.

Вариант 3 – Возможность взаимоблокировки

Возможная попытка в восстановлении взаимоисключения может состоять в установке значений управляющих переменных перед циклом проверки занятости ресурса.

```
int ResourceThread1=0; // =1 - ресурс занят потоком 1
int ResourceThread2=0; // =1 - ресурс занят потоком 2
Thread_1() {
  while (1) {
    // установить, что поток 1 пытается занять ресурс
    ResourceThread1 = 1;
    // повторять, пока ресурс занят потоком 2
    while ( ResourceThread2 == 1 );
    < Использование общего ресурса >
    ResourceThread1 = 0;
  }
}
Thread_2() {
  while (1) {
    // установить, что поток 2 пытается занять ресурс
    ResourceThread2 = 1;
```

```
    // повторять, пока ресурс используется потоком 1
    while ( ResourceThread1 == 1 );
    < Использование общего ресурса >
    ResourceThread2 = 0;
  }
}
```

Представленный вариант восстанавливает взаимоисключение, однако при этом возникает новая проблема – оба потока могут оказаться заблокированными вследствие бесконечного повторения циклов ожидания освобождения ресурсов (что происходит при одновременной установке управляющих переменных в состояние «занято»). Данная проблема известна под названием ситуации *тупика* (*дедлока* или *смертельного объятия*) и исключение тупиков является одной из наиболее важных задач в теории и практике параллельных вычислений. Более подробное рассмотрение темы будет выполнено далее в пп. 2.4; дополнительная информация по проблеме может быть получена в [33].

Вариант 4 – Бесконечное откладывание

В данном предлагаемом подходе для устранения тупика организуется временное снятия значения занятости управляющих переменных потоков в цикле ожидания ресурса.

```
int ResourceThread1=0; // =1 - ресурс занят потоком 1
int ResourceThread2=0; // =1 - ресурс занят потоком 2
Thread_1() {
  while (1) {
    // поток 1 пытается занять ресурс
    ResourceThread1=1;
    // повторять, пока ресурс занят потоком 2
    while ( ResourceThread2 == 1 ) {
      ResourceThread1 = 0; // снятие занятости ресурса
      <временная задержка>
      ResourceThread1 = 1;
    }
    < Использование общего ресурса >
    ResourceThread1 = 0;
  }
}
Thread_2() {
  while (1) {
    // поток 2 пытается занять ресурс
    ResourceThread2=1;
    // повторять, пока ресурс используется потоком 1
    while ( ResourceThread1 == 1 ) {
      ResourceThread2 = 0; // снятие занятости ресурса
      <временная задержка>
      ResourceThread2 = 1;
    }
    < Использование общего ресурса >
    ResourceThread2 = 0;
  }
}
```

Длительность временно́й задержки в циклах ожидания должна определяться при помощи некоторого случайного датчика. При таких условиях реализованный алгоритм обес-

печивает взаимоисключение и исключает возникновение тупиков, но опять таки не лишен существенного недостатка (перед чтением следующего текста попытайтесь определить этот недостаток). Проблема состоит в том, что потенциально решение вопроса о выделении может откладываться до бесконечности (при синхронном выполнении потоков). Данная ситуация известна под наименованием *бесконечное откладывание* (*starvation*) или *длительной блокировкой* (*live lock*).

Алгоритм Деккера

В алгоритме, впервые предложенным Деккером [31], предлагается объединение предложений вариантов 1 и 4 решения проблемы взаимоисключения.

```
int ThreadNum=1; // номер потока для доступа к ресурсу
int ResourceThread1=0; // =1 - ресурс занят потоком 1
int ResourceThread2=0; // =1 - ресурс занят потоком 2
Thread_1() {
 while (1) {
   // поток 1 пытается занять ресурс
   ResourceThread1=1;
   // цикл ожидания доступа к ресурсу
   while ( ResourceThread2 == 1 ) {
     if ( ThreadNum == 2 ) {
       ResourceThread1 = 0;
       // повторять, пока ресурс занят потоком 2
       while ( ThreadNum == 2 );
       ResourceThread1 = 1;
     }
   }
   < Использование общего ресурса >
   ThreadNum    = 2;
   ResourceThread1 = 0;
 }
}
Thread_2() {
 while (1) {
   // поток 2 пытается занять ресурс
   ResourceThread2=1;
   // цикл ожидания доступа к ресурсу
   while ( ResourceThread1 == 1 ) {
     if ( ThreadNum == 1 ) {
       ResourceThread2 = 0;
       // повторять, пока ресурс занят потоком 1
       while ( ThreadNum == 1 );
       ResourceThread2 = 1;
     }
   }
   < Использование общего ресурса >
   ThreadNum    = 1;
   ResourceThread2 = 0;
 }
}
```

Алгоритм Деккера гарантирует корректное решение проблемы взаимоисключения для двух потоков. Управляющие переменные *ResourceThread*1, *ResourceThread*1 обеспечивают взаимоисключение, переменная *ThreadNum* исключает возможность бесконечного откладывания. Если оба потока пытаются получить доступ к ресурсу, то поток, номер которого указан в *ThreadNum*, продолжает проверку возможности доступа к ресурсу (внешний цикл ожидания ресурса). Другой же поток в этом случае снимает свой запрос на ресурс, ожидает своей очереди доступа к ресурсу (внутренний цикл ожидания) и возобновляет свой запрос на ресурс.

Алгоритм Деккера может быть обобщен на случай произвольного количества потоков, однако такое обобщение приводит к заметному усложнению выполняемых действий. Кроме того, программное решение проблемы взаимоисключения потоков приводит к нерациональному использованию процессорного времени ЭВМ (потоку, ожидающему освобождения ресурса, постоянно требуется процессор для проверки возможности продолжения – *активное ожидание* (*busy wait*)).

2.2.2. Семафоры

Приведенные примеры показывают, что организация критических секций обычными средствами требует определенных усилий. Как результат, для решения вопросов взаимоисключения может оказаться целесообразной разработка новых (специальных) механизмов. Один из классических подходов в этом ряду – семафоры, предложенные Дейкстрой еще в середине 1960-х годов.

Под *семафором S* понимается [5] переменная особого типа, значение которой может опрашиваться и изменяться только при помощи специальных операций *P(S)* и *V(S)*, реализуемых в соответствии со следующими алгоритмами:

- операция ***P(S)***

если S > 0
  то S = S – 1
  иначе < ожидать S >

- операция ***V(S)***

если < один или несколько потоков ожидают S >
  то < снять ожидание у одного из ожидающих потоков >
  иначе S = S + 1

Принципиальным в понимании семафоров является то, *что операции P(S) и V(S) предполагаются неделимыми* (*атомарными*), что гарантирует взаимоисключение при использовании общих семафоров (для обеспечения неделимости операции обслуживания семафоров обычно реализуются средствами операционной системы).

Различают два основных типа семафоров. *Двоичные семафоры* принимают только значения 0 и 1, область значений *общих семафоров* – неотрицательные целые значения. В момент создания семафоры инициализируются некоторым целым значением.

Семафоры широко используются для синхронизации и взаимоисключения потоков. Так, например, проблема взаимоисключения при помощи семафоров может иметь следующее простое решение.

```
Semaphore Sem=1; // семафор взаимоисключения потоков
Thread_1() {
  while (1) {
    // проверить семафор и ждать, если ресурс занят
    P(Sem);
    < Использование общего ресурса >
```

```
    // освободить один из ожидающих ресурса потоков
    // увеличить семафор, если нет ожидающих потоков
    V(Sem);
  }
}
```

```
Thread_2() {
  while (1) {
    // проверить семафор и ждать, если ресурс занят
    P(Sem);
    < Использование общего ресурса >
    // освободить один из ожидающих ресурса потоков
    // увеличить семафор, если нет ожидающих потоков
    V(Sem);
  }
}
```

Приведенный пример рассматривает взаимоисключение только двух потоков, но, как можно заметить, совершенно аналогично может быть организовано взаимоисключение произвольного количества потоков.

Завершая рассмотрение данной темы, отметим, что на практике в разных средах выполнения наряду с поддержкой «стандартных» семафоров реализуются некоторые их разновидности – *мьютексы* (*mutex*), *замки* (*lock*) и др. Вариации касаются допустимого набора значений для переменных семафоров, рекурсивности вызова, введения дополнительного набора операций и т. д.

2.2.3. Мониторы

Несмотря на то, что семафоры в значительной степени упрощают проблему организации критических секций, тем не менее, семафоры являются достаточно низкоуровневым средством синхронизации потоков. Взаимосвязь семафоров и критических секций обеспечивается только на логическом уровне. Нерегламентированное использование семафоров приводит к усложнению схемы параллельного выполнения разрабатываемой программы. Более высокоуровневым механизмом синхронизации являются мониторы.

Мониторы представляют собой программные модули (объекты), которые реализуют (*инкапсулируют*) все необходимые действия с разделяемым ресурсом (см., например, [5]). Общий формат определения монитора может быть представлен следующим образом:

```
Monitor <Name> {
  <объявления переменных>
  <операторы инициализации>
  <процедуры монитора>
}
```

Как можно заметить, описание монитора достаточно близко совпадает с описанием класса в алгоритмическом языке C++. Принципиальное отличие состоит в том, что процедуры монитора, в обязательном порядке, выполняются *в режиме взаимоисключения*, т. е. при выполнении какой-либо процедуры монитора все остальные попытки вызова других процедур этого же монитора блокируются. Обеспечение такого правила выполнения процедур монитора должно осуществляться средой выполнения, в которой поддерживается концепция мониторов.

Как пример использования монитора можно рассмотреть задачу организации доступа к общей переменной – возможный вариант монитора для этой цели может быть реализован в виде:

```
Monitor SharedMem {
  int N=0;  // Общая переменная
  procedure Set (int v) {   // Операция записи
    N = v;
  }
  procedure Get (int &v) {  // Операция чтения
    v = N;
  }
  procedure Inc () {  // Операция изменения
    N = N + 1;
  }
}
```

Помимо взаимоисключения процедур, другим важным свойством понятия монитора является его полная изолированность от остального кода программы:

– Переменные монитора недоступны вне монитора и могут обрабатываться только процедурами монитора.

– Вне монитора доступны только процедуры монитора.

– Переменные, объявленные вне монитора, недоступны внутри монитора.

Подобная локализация (инкапсуляция) и объединение в рамках монитора всех критических секций потоков приводит к значительному снижению сложности логики параллельного выполнения.

В ряде случаев для процедур монитора может потребоваться приостановка (блокировка) до выполнения каких-либо условий – например, при реализации семафоров при помощи монитора процедура занятия семафора должна блокироваться, если данный семафор уже является занятым. Для решения таких проблем в мониторах вводится дополнительный механизм условных переменных.

Условные переменные – это объекты специального типа *cond*, используемые для организации приостановки работы процедур монитора до выполнения определенных логических условий. Действия с условными переменными осуществляются при помощи двух основных операций:

– **wait(cv)** – операция ожидания наступления события; при этом для процедуры, выполнившую данную операцию, снимается блокировка и процедуры монитора снова становятся доступными для использования (несмотря на то, что приостановленные при помощи операции *wait* процедуры еще не завершили свое выполнение);

– **signal(cv)** – операция для объявления наступления события; в результате выполнения данной операции одна из процедур, ожидающих данного события, становится готовой для продолжения (если ожидающих данного события процедур нет, сигнал о наступлении события теряется)

(отметим, что взаимосвязь условных переменных и событий, им соответствующих, устанавливается только на логическом уровне).

С использованием механизма условных переменных можно провести, например, реализацию семафоров при помощи монитора:

```
Monitor Semaphore {
  int  s = 1;  // Счетчик семафора
  cond cv;     // Переменная для события s>0
```

```
  procedure Psem () {  // Занятие семафора
    while ( s == 0 ) wait(cv);
    s = s - 1;
  }
  procedure Vsem () {  // Освобождение семафора
    s = s + 1;
    signal(cv)
  }
}
```

(следует заметить, что и обратно, монитор может быть реализован с использованием семафоров).

Понятие условных переменных достаточно близко понятию семафора, однако есть и существенные различия. Операция *wait* всегда приостанавливает поток, а операция *P* семафора блокирует поток только в случае, если переменная семафора равна нулю. В свою очередь, операция *signal* не выполняет никаких действий, если нет ожидающих потоков, в то время как операция *V* семафора либо активизирует один из блокированных потоков, либо увеличивает значение переменной семафора.

Различают две различные схемы продолжения работы монитора после выполнения операции *signal*:

– *синхронный* способ, при котором процедура монитора, выполнившая операцию *signal*, приостанавливается, а для продолжения работы выбирается одна из процедур монитора, ожидавших данного события – в литературе мониторы такого типа обычно называют *мониторами Хоара*;

– *асинхронный* способ, для которого порядок действий является обратным по сравнению с мониторами Хоара: одна из процедур монитора, ожидавших данного события, переводится в состояние готовности для выполнения, и далее продолжается работа процедуры монитора, выполнившая операцию *signal* – мониторы такого типа обычно называют *мониторами Мека*.

Наибольшее распространение получила вторая схема – именно такой подход используется в операционной систем Unix, языке программирования Java и библиотеке Pthreads.

Подводя итог, можно отметить, что мониторы являются предпочтительным способом организации взаимоисключения. При этом, если в среде выполнения параллельных программ мониторы в явном виде не поддерживаются, использование мониторов можно промоделировать при помощи семафоров.

2.3. Синхронизация потоков

Рассмотренная в предыдущем разделе проблема взаимоисключения на самом деле является частным случаем проблемы синхронизации параллельно выполняемых потоков. Необходимость синхронизации обуславливается тем обстоятельством, что не все возможные траектории совместно выполняемых потоков являются допустимыми (так, например, при использовании общих ресурсов требуется обеспечить взаимоисключение). В самом общем виде, *синхронизация* может быть обеспечена при помощи задания необходимых логических условий, которые должны выполняться в соответствующих точках траекторий потоков.

Организация синхронизации является важной частью разработки параллельной программы. Принципиальный момент состоит в определении полного набора синхронизирующих действий, обеспечивающих гарантированное исключение недопустимых траекторий параллельной программы – в этом случае говорят, что программа обладает *свойством безопасности*. Строгость данного свойства может быть несколько снижена – так, набор син-

хронизирующих действий можно ограничить требованием обеспечения достижимости допустимых траекторий программы – такое поведение обычно именуется *свойством живучести*.

В числе наиболее широко используемых общих механизмов синхронизации (помимо средств взаимоисключения) – *условные переменные* и *барьерная синхронизация*.

2.3.1. Условные переменные

Условные переменные, используемые для синхронизации потоков, практически совпадают с аналогичными средствами мониторов (см. п. 2.2.3). Т. е., как и ранее, условные переменные – это объекты некоторого специального типа, используемые для организации синхронизации потоков. Основные операции с условными переменными:

– **wait(cv)** – ожидание события, связанного с условной переменной;

– **signal(cv)** – объявление наступления события.

При выполнении операции *wait* поток блокируется; в результате выполнения операции *signal* один из потоков, ожидающих данного события, переводится в состояние готовности для выполнения (как и ранее, при отсутствии ожидающих потоков сигнал о наступлении события теряется).

При реализации механизма условных переменных в разных средах выполнения рассмотренный выше набор операций обычно расширяется.
В числе подобных расширений:

– **empty(cv)** – проверка наличия потоков, ожидающих события;

– **broadcast(cv)** – объявление наступления события для всех ожидающих потоков (все ожидающие данного события потоки переводятся в состояние готовности для выполнения).

2.3.2. Барьерная синхронизация

Барьерная синхронизация является частным вариантом организации согласованного выполнения потоков и состоит в выделении точек в траекториях параллельно выполняемых потоков, таких, что выполнение потоков может быть продолжено только в случае, когда все потоки достигнут своих точек барьерной синхронизации. В качестве примера необходимости такой синхронизации можно привести параллельный алгоритм, итерации которого выполняются отдельными потоками и перед переходом к каждой следующей итерации все потоки должны завершить свои текущие итерации.

Операция барьерной синхронизации обычно именуется как *barrier*(). Важно отметить, что данная операция является коллективной и должна быть выполнена в каждом потоке, участвующем в барьерной синхронизации. Как правило, в барьерной синхронизации принимают участие все параллельно выполняемые потоки программы, хотя в различных средах выполнения может быть предусмотрена возможность барьерной синхронизации и для отдельных групп выполняемых потоков.

2.4. Взаимоблокировка потоков

В самом общем виде *взаимоблокировка* может быть определена [33] как ситуация, в которой один или несколько потоков ожидают какого-либо события, которое никогда не произойдет (в научно-технической литературе такая ситуация чаще всего называется как *тупик*, а также *дедлок* или *смертельное объятие*). Важно отметить, что состояние тупика может наступить не только вследствие логических ошибок, допущенных при разработке параллельных программ, но и в результате возникновения тех или иных событий в вычислительной системе (выход из строя отдельных устройств, нехватка ресурсов и т. п.). Простой пример тупика может состоять в следующем. Пусть имеется два потока, каждый из

которых в монопольном режиме обрабатывает собственный файл данных. Ситуация тупика возникнет, например, если первому потоку для продолжения работы потребуется файл второго потока и одновременно второму потоку окажется необходимым файл первого потока (см. рис. 2.3).

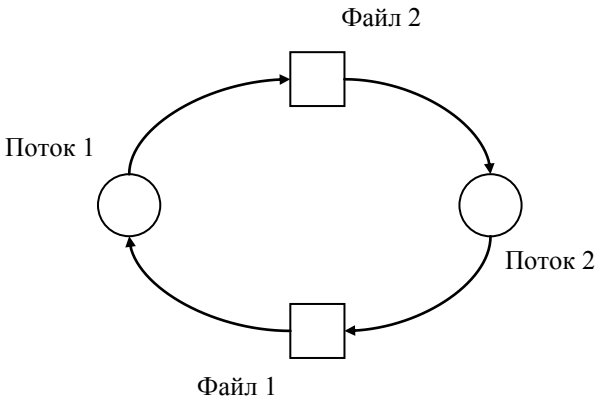

Рис. 2.3. Пример ситуации тупика

Проблема тупиков имеет многоплановый характер. Это и сложность диагностирования состояния тупика (система выполняет длительные расчеты или «зависла» из-за тупика), и необходимость определенных специальных действий для выхода из тупика, и возможность потери данных при восстановлении системы при устранении тупика.

В данном разделе будет рассмотрен один из аспектов проблемы тупика – анализ причин возникновения тупиковых ситуаций при использовании разделяемых ресурсов и разработка на этой основе методов предотвращения тупиков. Дополнительная информация по теме может быть получена в [33].

Могут быть выделены следующие **необходимые условия тупика** [33]:

– потоки требуют предоставления им права монопольного управления ресурсами, которые им выделяются (*условие взаимоисключения*);

– потоки удерживают за собой ресурсы, уже выделенные им, ожидая в то же время выделения дополнительных ресурсов (условие *ожидания* ресурсов);

– ресурсы нельзя отобрать у потоков, удерживающих их, пока эти ресурсы не будут использованы для завершения работы (условие *неперераспределяемости*);

– существует кольцевая цепь потоков, в которой каждый поток удерживает за собой один или более ресурсов, требующихся следующему потоку цепи (условие *кругового ожидания*).

Как результат, для обеспечения отсутствия тупиков необходимо исключить возникновение, по крайней мере, одного из рассмотренных условий. Далее будет предложена модель программы в виде графа «поток–ресурс», позволяющего обнаруживать ситуации кругового ожидания [36].

2.4.1. Модель программы в виде графа «поток–ресурс»

Состояние программы может быть представлено в виде ориентированного графа (*V*, *E*) со следующей интерпретацией и условиями:

1. Множество *V* разделено на два взаимно пересекающихся подмножества *P* и *R*, представляющие *потоки*

$$P = (p_1, p_2, \cdots, p_n)$$

и *ресурсы*

$$R = (R_1, R_2, \cdots, R_m)$$

программы.

2. Граф является «двудольным» по отношению к подмножествам вершин *P* и *R*, т. е. каждое ребро $e \in E$ соединяет вершину *P* с вершиной *R*. Если ребро e имеет вид $e = (p_i, R_j)$, то e есть ребро *запроса* и интерпретируется как запрос от потока p_i на единицу ресурса R_j. Если ребро e имеет вид $e = (R_j, p_i)$, то e есть ребро *назначения* и выражает назначение единицы ресурса R_j потоку p_i.

3. Для каждого ресурса $R_j \in R$ существует целое $k_j \geq 0$, обозначающее количество единиц ресурса R_j.

4. Пусть $|(a,b)|$ - число ребер, направленных от вершины a к вершине b. Тогда при принятых обозначениях для ребер графа должны выполняться условия:

– Может быть сделано не более k_j назначений (распределений) для ресурса R_j, т. е.

$$\sum_i |(R_j, p_i)| \leq k_j, \ \ 1 \leq j \leq m;$$

– Сумма запросов и распределений относительно любого потока для конкретного ресурса не может превышать количества доступных единиц, т. е.

$$|(R_j, p_i)| + |(p_i, R_j)| \leq k_j, \ \ 1 \leq i \leq n, \ \ 1 \leq j \leq m.$$

Граф, построенный с соблюдением всех перечисленных правил, именуется в литературе как *граф «поток-ресурс»*. Для примера на рис. 2.3 приведен граф программы, в которой ресурс 1 (файл 1) выделен потоку 1, который, в свою очередь, выдал запрос на ресурс 2 (файл 2). Поток 2 владеет ресурсом 2 и нуждается для своего продолжения в ресурсе 1.

Состояние программы, представленное в виде графа «поток-ресурс», изменяется только в результате *запросов, освобождений* или *приобретений* ресурсов каким-либо из потоков программы.

Запрос. Если программа находится в состоянии *S* и поток p_i не имеет невыполненных запросов, то p_i может запросить любое число ресурсов (в пределах ограничения 4). Тогда программа переходит в состояние *T*

$$S \xrightarrow{i} T.$$

Состояние *T* отличается от *S* только дополнительными ребрами запроса от p_i к затребованным ресурсам.

Приобретение. Операционная система может изменить состояние программы *S* на состояние *T* в результате операции приобретения ресурсов потоком p_i тогда и только тогда, когда p_i имеет запросы на выделение ресурсов и все такие запросы могут быть удовлетворены, т. е. если

$$\forall R_j : (p_i, R_j) \in E \ \ \Rightarrow \ \ (p_i, R_j) + \sum_l |(R_j, p_l)| \leq k_j.$$

Граф *T* идентичен *S* за исключением того, что все ребра запроса (p_i, R_j) для p_i обратны ребрам (R_j, p_i), что отражает выполненное распределение ресурсов.

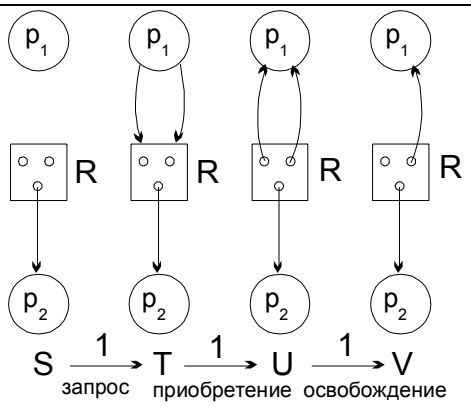

Рис. 2.4. Пример переходов программы из состояния
в состояние

Освобождение. Поток p_i может вызвать переход из состояния *S* в состояние *T* с помощью освобождения ресурсов тогда и только тогда, когда p_i не имеет запросов, а имеет некоторые распределенные ресурсы, т. е.

$$\forall R_j : (p_i, R_j) \notin E, \quad \exists R_j : (R_j, p_i) \in E.$$

В этой операции p_i может освободить любое непустое подмножество своих ресурсов. Результирующее состояние *T* идентично исходному состоянию *S* за исключением того, что в *T* отсутствуют некоторые ребра приобретения из *S* (из *S* удаляются ребра (R_j, p_i) каждой освобожденной единицы ресурса R_j).

Для примера на рис. 2.4 показаны состояния программы с одним ресурсом емкости 3 и двумя потоками после выполнения операций запроса, приобретения и освобождения ресурсов для первого потока.

При рассмотрении переходов программы из состояния в состояние важно отметить, что поведение потоков является недетерминированным – при соблюдении приведенных выше ограничений выполнение любой операции любого потока возможно в любое время.

2.4.2. Описание возможных изменений состояния программы

Определение состояния программы и операций перехода между состояниями позволяет сформировать модель параллельной программы следующего вида.

Под *программой* будем понимать систему

$$\langle \Sigma, P \rangle,$$

где Σ есть множество состояний программы (S, T, U,…), а P представляет множество потоков $(p_1, p_2, \cdots, p_n)$. *Поток* $p_i \in P$ есть частичная функция, отображающая состояния программы в непустые подмножества состояний

$$p_i : \Sigma \to \{\Sigma\},$$

где $\{\Sigma\}$ есть множество всех подмножеств Σ. Обозначим множество состояний, в которые может перейти программа при помощи потока p_i (*область значений потока* p_i) при нахождении программы в состоянии S через $p_i(S)$. Возможность перехода программы из состояния S в состояние T в результате некоторой операции над ресурсами в потоке p_i (т.е. $T \in p_i(S)$) будем пояснять при помощи записи

$$S \xrightarrow{i} T.$$

Обобщим данное обозначение для указания достижимости состояния T из состояния S в результате выполнения некоторого произвольного количества переходов в программе

$$S \xrightarrow{*} T \quad \Leftrightarrow \quad (S = T) \vee (\exists p_i \in P : S \xrightarrow{i} T) \vee$$
$$(\exists p_i \in P, U \in \Sigma : S \xrightarrow{i} U, U \xrightarrow{*} T)$$

2.4.3. Обнаружение и исключение тупиков

С учетом построенной модели и введенных обозначений можно выделить ряд ситуаций, возникающих при выполнении программы и представляющих интерес при рассмотрении проблемы тупика:

- *поток* p_i *заблокирован* в состоянии S, если программа не может изменить свое состояние при помощи этого потока, т.е. если $p_i(S) = \varnothing$;
- *поток* p_i находится в *тупике* в состоянии S, если этот поток является заблокированным в любом состоянии T, достижимом из состояния S, т. е.

$$\forall T : S \xrightarrow{*} T \Rightarrow p_i(T) = \varnothing;$$

- *состояние* S называется *тупиковым*, если существует поток p_i, находящийся в тупике в этом состоянии;
- *состояние* S есть *безопасное состояние*, если любое состояние T, достижимое из S, не является тупиковым.

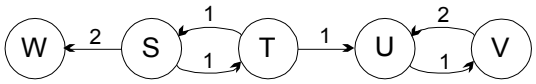

Рис. 2.5. Пример графа переходов программы

Для примера на рис. 2.5 приведен граф переходов программы, в котором состояния *U* и *V* являются безопасными, состояния S, *T* и *W* не являются безопасными, а состояние *W* есть состояние тупика.

Рассмотренная модель программы может быть использована для определения возможных состояний программы, обнаружения и недопущения тупиков. В качестве возможных теоретических результатов такого анализа может быть приведена теорема [36].

Теорема. Граф «поток–ресурс» для состояния программы с ресурсами единичной емкости указывает на состояние тупика тогда и только тогда, когда он содержит цикл.

Дополнительный материал по исследованию данной модели может быть получен в [36].

2.5. Классические задачи синхронизации

В ходе изучения проблем параллельного программирования и разработки методов их решения сформировался набор некоторых «классических» задач, на которых принято демонстрировать результаты применения новых разрабатываемых подходов. В числе этих задач:

- Задача «Производители–Потребители» (*Producer–Consumer problem*);
- Задача «Читатели–Писатели» (*Readers-Writers problem*);
- Задача «Обедающие философы» (*Dining Philosopher problem*);
- Задача «Спящий брадобрей» (*Sleeping Barber problem*).

Далее будет представлено описание этих задач и рассмотрены примеры их возможных решений.

2.5.1. Задача «Производители–Потребители»

В научно-технической литературе существует достаточно большое количество вариантов постановки данной задачи. В наиболее простом случае предполагается, что существует два потока, один из которых (*производитель*) генерирует сообщения (*изделия*), а второй поток (*потребитель*) их принимает для последующей обработки. Потоки взаимодействуют через некоторую область памяти (*хранилище*), в которой производитель размещает свои генерируемые сообщения и из которой эти сообщения извлекаются потребителем (см. рис. 2.5).

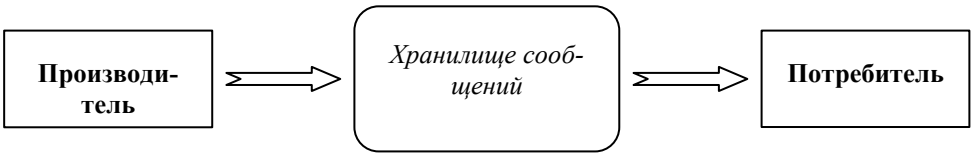

Рис. 2.5. Общая схема задачи
«Производители–Потребители»

Рассмотрев постановку данной задачи, можно заметить, что хранилище сообщений представляет собой не что иное, как общий разделяемый ресурс, и использование этого ресурса должно быть построено по правилам взаимоисключения. Кроме того, следует учитывать, что потребление ресурса иногда может оказаться невозможным (отсутствие сообщений в хранилище), а при добавлении сообщений в хранилище могут происходить задержки (в случае полного заполнения хранилища).

Далее будет представлено решение этой задачи с использованием семафоров, однако перед ознакомлением с этим решением полезно попытаться разработать решение самостоятельно. Дополнительно можно порекомендовать ознакомиться и с другими возможными решениями данной задачи, используя для этого, например, работу [33]. Кроме того, крайне полезно не ограничиваться только «теоретическим» знакомством с этими алгоритмами, а попытаться их реализовать и проверить в той или иной среде выполнения параллельных программ (можно заметить, что все приведенные замечания будут справедливы и при рассмотрении всех последующих задач).

Для организации работы используем три семафора:

– **Access** – двоичный семафор для организации взаимоисключения при доступе к хранилищу;

– **Full** – общий семафор, блокирующий поток-производитель при попытке записи сообщения в полностью заполненное хранилище (в переменной семафора будет хранится количество имеющихся свободных мест для сообщений в хранилище);

– **Empty** – общий семафор, блокирующий поток-потребитель при попытке чтения сообщения из пустого хранилище (в переменной семафора будет хранится количество имеющихся сообщений в хранилище).

Возможная реализация взаимоисключения потоков может состоять в следующем.

```
// Семафор взаимоисключения доступа
Semaphore Access = 1;
// Семафор блокировки записи в полное хранилище
Semaphore Full   = n;
// Семафор блокировки чтения из пустого хранилища
Semaphore Empty  = 0;
Producer(){
  <Генерация нового сообщения>
  // Доступ только при наличии пустых мест
  P(Full);
  P(Access);  // Блокировка доступа к хранилищу
  <Запись сообщения в хранилище>
  // Снятие блокировки доступа к хранилищу
  V(Access);
  // Отметка наличия сообщений в хранилище
  V(Empty);
}
Consumer(){
  // Доступ только при наличии сообщений
  P(Empty);
  P(Access); // Блокировка доступа к хранилищу
  <Чтение сообщения из хранилища>
  // Снятие блокировки доступа к хранилищу
  V(Access);
  // Отметка наличия пустых мест в хранилище
  V(Full);
  <Обработка полученного сообщения>
}
```

В качестве самостоятельного упражнения может быть предпринята попытка разработка решения задачи при помощи монитора и/или рассмотрение более усложненных постановок задачи «Производитель–Потребитель» (произвольное количество потоков, двустороння передача сообщений и т. п.).

2.5.2. Задача «Читатели–Писатели»

Возможная простая постановка этой задачи состоит в следующем. Пусть имеется некоторая область памяти – хранилище данных, с которым одновременно работают несколько потоков. По характеру использования хранилища потоки могут быть разделены на два типа. Одна группа – это *потоки-читатели*, которые осуществляют только чтение (без удаления) хранилища. Другая – оставшаяся часть потоков – это *потоки-писатели*, которые выполняют запись новых значений имеющихся в хранилище данных (см. рис. 2.6).

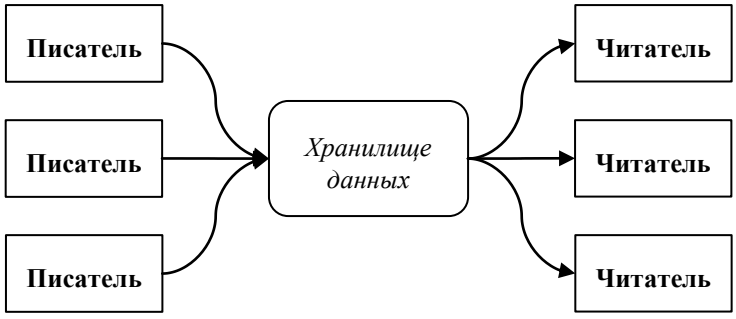

Рис. 2.6. Общая схема задачи «Читатели–Писатели»

При рассмотрении этой задачи предполагается, что потоки-читатели не изменяют каких-либо параметров хранилища и могут, тем самым, работать одновременно, не мешая друг другу. Для потоков-писателей ситуация обратная – предполагается, что запись не может выполняться несколькими потоками одновременно и, понятно, во время записи не допускаются какие-либо операции чтения.

Постановка задачи может различаться в правилах разрешения ситуации обращения потока-писателя к хранилищу. Если при попытке записи имеются активные потоки-читатели, то поток-писатель должен быть заблокирован до завершения работы потоков-читателей. Вопрос состоит в том, что делать с новыми поступающими запросами на чтение. Возможный вариант – отдать предпочтение потокам-читателям (т. е. новые потоки-читатели могут начинать свою работу, не обращая внимания на блокированный процесс-писатель), однако в этом случае блокировка потока-писателя может продолжаться бесконечно долго. Альтернативный подход – блокировка новых потоков-читателей, появившихся после блокировки потока-писателя.

Для решения поставленной задачи снова используем семафоры. Введем следующие переменные:

– **ReadCount** – переменная-счетчик количества активных потоков-читателей;

– **ReadSem** – двоичный семафор для взаимоисключения доступа к переменной *ReadCount*;

– **Access** – двоичный семафор для организации взаимоисключения при доступе к хранилищу.

Возможная реализация синхронизации взаимодействия потоков может состоять в следующем.

```
// Счетчик количества активных потоков-читателей
int ReadCount = 0;
// Семафор доступа к переменной ReadCount
Semaphore ReadSem = 1;
// Семафор доступа к хранилищу
Semaphore Access  = 1;
Writer(){ // Поток-писатель
  // Блокировка доступа к хранилищу
  P(Access);
  <Выполнение операции записи>
  // Снятие блокировки доступа к хранилищу
  V(Access);
}
Reader(){ // Поток-читатель
  // Блокировка доступа к переменной ReadCount
  P(ReadSem);
  // Изменение счетчика активных читателей
  ReadCount++;
  if( ReadCount == 1 )
  // Блокировка доступа к хранилищу
  // (если поток-читатель первый)
    P(Access);
// Снятие блокировки доступа к ReadCount
  V(ReadSem);
  <Выполнение операции чтения>
  // Блокировка доступа к переменной ReadCount
  P(ReadSem);
  // Изменение счетчика активных читателей
  ReadCount--;
  // Снятие блокировка доступа к хранилищу
  // (если завершается последний поток-читатель)
  if( ReadCount == 0 )
    V(Access);
  // Снятие блокировки доступа к ReadCount
  V(ReadSem);
}
```

В качестве самостоятельного задания предлагается провести анализ приведенного решения и определить, какой вариант поведения новых потоков-читателей при наличии блокированных потоков-писателей в данном алгоритме обеспечивается. Если предлагаемый алгоритм отдает предпочтение потокам-читателям, следует попытаться разработать решение, справедливое по отношению к потокам-писателям.

Дополнительная информация по данной задаче может быть получена, например, в [33].

2.5.3. Задача «Обедающие философы»

Данная задача является одной из наиболее известных в области параллельного программирования. Если задача «Читатели–Писатели» помогает демонстрировать методы параллельного и исключительного доступа к одному общему ресурсу, то задача «Обедающие философы» позволяет рассмотреть способы доступа нескольких потоков к нескольким разделяемых ресурсов.

Исходная формулировка задачи, впервые предложенная Э. Дейкстрой, выглядит следующим образом. Представляется ситуация, в которой пять философов располагаются за круглым столом. При этом философы либо размышляют, либо кушают. Для приема пищи в центре стола большое блюдо с неограниченным количеством спагетти, и тарелки, по одной перед каждым философом. Предполагается, что поесть спагетти можно только с использованием двух вилок. Для этого на столе располагается ровно пять вилок – по одной между тарелками философов (см. рис. 2.7).

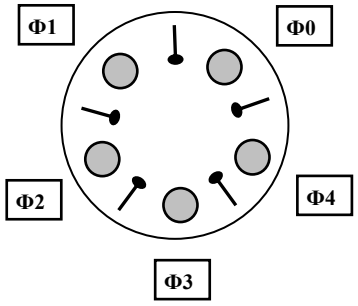

Рис. 2.7. Общая схема задачи «Обедающие философы»

Для того, чтобы приступить к еде, философ должен взять вилки слева и справа (если они не заняты), наложить спагетти из большого блюда в свою тарелку, поесть, а затем обязательно положить вилки на свои места для их повторного использования (проблема чистоты вилок в задаче не рассматривается).

Нетрудно заметить, что в данной задаче философы представляют собой потоки, а вилки – общие разделяемые ресурсы. Тогда первое очевидное, на первый взгляд, решение состоит в том, чтобы для каждой вилки (ресурса) ввести отдельный семафор для блокировки философа (потока) в ситуации, когда нужная для еды вилка уже занята соседним философом. Кроме того, можно применить некоторое регламентирующее правило порядка взятия вилок – например, философ сначала берет левую вилку, затем правую.

Итак, получаемый в результате алгоритм деятельности каждого философа состоит в следующем: как только философ приступает к еде, он пытается взять левую вилку. Если она занята, философ ждет ее освобождения и в конце концов ее получает. Затем философ пытается взять правую вилку. И опять же, если вилка занята, философ снова ждет ее освобождения (при этом левую вилку он по-прежнему хранит у себя). После получения правой вилки философ ест спагетти, после чего освобождает обе вилки.

Возможная реализация предложенной схемы (опять же с использованием семафоров) может состоять в следующем.

```
// Семафоры доступа к вилкам
Semaphore fork[5] = { 1, 1, 1, 1, 1 };
// Поток -философ (для всех философов одинаковый)
Prilosopher(){
  // i - номер философа
  while (1) {
    P(fork[i]);         // Доступ к левой вилке
    P(fork[(i+1)%5]);  // Доступ к правой вилке
    <Питание>
    // Освобождение вилок
    V(fork[i]); V(fork[(i+1)%5])
    <Размышление>
  }
}
```

(выражение (*i*+1)%5 определяет номер правой вилки, % есть операция получения остатка от целого деления в алгоритмическом языке С).

Внимательно проанализируйте представленный алгоритм. После тщательного изучения можно увидеть, что данное решение может приводить к тупиковым ситуациям – например, когда все философы одновременно проголодаются и каждый из них возьмет свои левые вилки. В результате правые вилки для всех философов окажутся занятыми и философы перейдут к бесконечному ожиданию (отметим, как сложно выявить подобную ситуацию при помощи тестов; кроме того, подобную ошибочную ситуацию сложно повторить при повторных запусках программы).

Возможны различные варианты исправления рассмотренного алгоритма. Для этого надо устранить одно из условий возникновения тупика (см. начало данного подраздела) – например, попытаться избежать кругового ожидания. Для этого можно изменить порядок взятия вилок для одного из философов (например, для четвертого), который должен брать сначала правую вилку, а только затем левую. Получаемое в результате решение выглядит следующим образом.

```
// Семафоры доступа к вилкам
Semaphore fork[5] = { 1, 1, 1, 1, 1 };
// Поток-философ (для всех, кроме четвертого)
Prilosopher(){
  // i - номер философа
  while (1) {
    P(fork[i]);         // Доступ к левой вилке
    P(fork[(i+1)%5]);  // Доступ к правой вилке
    <Питание>
    // Освобождение вилок
    V(fork[i]); V(fork[(i+1)%5])
    <Размышление>
  }
Prilosopher4(){ // Поток для четвертого философа)
  while (1) {
    P(fork[0]); // Доступ к правой вилке
    P(fork[4];  // Доступ к левой вилке
    <Питание>
    V(fork[0]); V(fork[4)  // Освобождение вилок
    <Размышление>
  }
```

Изучение нового варианта алгоритма синхронизации показывает, что он гарантирует отсутствие тупиков.

В качестве самостоятельного задания можно предложить выполнить некоторую вариацию постановки задачи и разработать свои варианты алгоритмов синхронизации (так, можно предложить правило, по которому философ берет вилки только в том случае, если они обе свободны, и т. п.).

Дополнительная информация по данной задаче может быть получена, например, в [33].

2.5.4. Задача «Спящий парикмахер»

Данная задача также в числе широко используемых примеров для демонстрации проблем синхронизации. На примере этой задачи можно показать методы последовательного доступа к набору разделяемых ресурсов и рассмотреть организацию вычислений в соответствии со схемой «клиент–сервер».

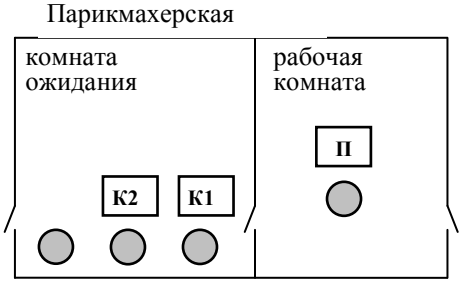

Рис. 2.8. Общая схема задачи «Спящий парикмахер»
(К – клиенты, П – парикмахер)

Смысловая окраска задачи «Спящий парикмахер» состоит в следующем (см. рис. 2.8). Обсуждается проблема обслуживания клиентов парикмахерской. В парикмахерской имеется два помещения: комната ожидания, в которой ограниченное количество мест, и рабочая комната с единственным креслом, в котором располагается обслуживаемый клиент.

Посетители заходят в парикмахерскую – если комната ожидания заполнена, то поворачиваются и уходят; иначе занимают свободные места и засыпают, ожидая своей очереди к парикмахеру. Парикмахер, если есть клиенты, приглашает одного из них в рабочую комнату и подстригает его. После стрижки клиент покидает парикмахерскую, а парикмахер приглашает следующего посетителя и т. д. Если клиентов нет (комната ожидания пуста), парикмахер садится в свое рабочее кресло и засыпает. Будит его очередной появляющийся посетитель парикмахерской.

В данной задаче ресурсами являются места ожидания и рабочее кресло. Потоки-клиенты должны получать эти ресурсы строго последовательно: сначала посетитель должен найти место в комнате ожидания и только затем занять очередь к парикмахеру. При этом предоставление рабочего кресла для обслуживания производит специальный процесс-парикмахер. В этом плане, парикмахера можно интерпретировать как сервер, предоставляющий требуемый сервис.

Для сравнения представим решение данной задачи при помощи монитора. Определим события, которые происходят в процессе вычислений (для каждого события сразу укажем условные переменные, которые будут использоваться для объявления этих событий):

– **Client** – событие, означающее, что есть ожидающие посетители; событие объявляется каждый раз при появлении нового клиента; данное событие пробуждает спящего парикмахера, заснувшего при отсутствии клиентов;

– **Barber** – событие при освобождении парикмахера; по данному событии пробуждается один из ожидающих клиентов, который и переходит в рабочую комнату для обслуживания;

– **Service** – событие при завершении обслуживания очередного клиента; клиент может покинуть парикмахерскую.

С учетом введенных условных переменных решение задачи с использованием монитора может состоять в следующем.

```
Monitor BarberShop {
  // Максимальное количество посетителей
  const int ClientMax = 10;
  // Текущее количество клиентов
  int ClientNum = 0;
  // Событие: Имеются ожидающие посетители
  cond Client;
  // Событие: Парикмахер свободен
  cond Barber;
  // Событие: Обслуживание завершено
  cond Service;
  Client(){ // Поток клиента
    // Ждать только если есть места
    if ( ClientNum < ClientMax ) {
      ClientNum++;
      // Есть посетители –разбудить парикмахера
      signal(Client);
      wait(Barber);    // Ждать парикмахера
      wait(Service);   // Ждать окончания стрижки
    }
  }
  Barber(){ // Поток парикмахера
    while (1) {
      // Ждать посетителей (спать)
```

```
      if ( ClientNum == 0 ) wait(Client);
      // Парикмахер свободный - пригласить клиента
      signal(Barber);
      ClientNum--;
      <Обслуживание>
      // Обслуживание завершено – клиент может уйти
      signal(Service);
    }
  }
}
```

Рассматривая данную задачу, попытайтесь разработать другие возможные способы решения (в частности, с использованием семафоров). Может быть также изменена или расширена постановка задачи (можно, например, увеличить количество парикмахеров).

Дополнительная информация по данной задаче может быть получена, например, в [33].

2.6. Методы повышения эффективности параллельных программ

Соблюдение правил синхронизации, взаимоисключения и недопущения взаимоблокировки приводит к построению корректных параллельных программ. В данном разделе будут кратко рассмотрены методы повышения эффективности параллельных вычислений.

2.6.1. Оптимизация количества потоков

Количество используемых потоков существенным образом сказывается на эффективности параллельных вычислений. При малом количестве потоков (меньшим, чем число ядер/процессоров) не будет задействован полностью потенциал компьютерного оборудования. Однако избыток потоков также может негативно сказаться на эффективности, и основные причины этого состоят в следующем:

– При большом количестве потоков могут увеличиться затраты на их обслуживание – потоки надо создавать и завершать; при нехватке вычислительных устройств (ядер/процессоров) для потоков нужно прерывать выполнение активных потоков, сохранять их состояния и передавать на выполнение новые потоки; следует отметить, что необходимое время на переключение потоков обычно является небольшим, а затраты на создание и завершение потоков можно снизить, если выполнять эти действия только при старте и завершении параллельной программ.

– Более серьезная причина потери эффективности может состоять в ухудшении продуктивного использования кэш-памяти; как известно, кэш-память является во много (10–100) раз быстрее обычной оперативной памяти и быстродействие программы можно значительно повысить, если обеспечить присутствие обрабатываемых данных в кэш-памяти; в ситуациях, когда из-за избытка потоков необходимо переключать вычислительные устройства (ядра/процессоры) на выполнение разных потоков, потоки должны восстанавливать состояние кэш-памяти при каждом своем новом возобновлении (что и приводит к появлению дополнительных задержек вычислений); аналогичная проблема возникает и при использовании виртуальной памяти (часть данных приостановленных потоков может быть вытеснена в медленную внешнюю память).

– Значительная проблема при большом количестве потоков состоит в существенном повышении затрат на организацию синхронизации и взаимоисключения потоков и избыток потоков в силу чрезмерной синхронизации может привести не к ускорению, а к достаточно заметному замедлению вычислений. Наихудшая ситуация может состоять в приостановке потоков, которые должны снять блокировку с общих разделяемых ресурсов при переключении ядре/процессоров на выполнение новых потоков – в этом случае потоки,

которые активизированы для выполнения, не могут продолжаться из-за блокировки ресурсов, а потоки, которые могут снять эту блокировку, ждут своей очереди на выполнение.

Все вышесказанное говорит о том, что количество потоков должно выбираться тщательно. Обычно число потоков определяется как параметр запуска параллельной программы, и в лучшем случае это параметр должен задаваться самой средой выполнения автоматически (например, средствами OpenMP). Выбор количества потоков должен осуществляться с учетом числа вычислительных элементов (ядер/процессоров), количеством устройств кэш-памяти (количества вычислительных элементов и устройств кэш-памяти могут не совпадать) и объемом выполняемых вычислений в потоках. В частности, целесообразно, чтобы количество вычислительно-интенсивных потоков совпадало с числом ядер/процессоров или с имеющимся количеством устройств кэш-памяти.

2.6.2. Минимизация взаимодействия потоков

Безусловно, максимальная эффективность параллельности достигается при полной независимости параллельно выполняемых потоков (при условии равного распределения вычислительной нагрузки). Любое взаимодействие и, как результат, синхронизация деятельности потоков приводит к появлению задержек и снижению быстродействия вычислений. Поскольку полностью избежать взаимодействия потоков невозможно (параллельные потоки занимаются решением единой задачи), то общие рекомендации по снижению потерь от синхронизации состоят в следующем:

– Уменьшение, по мере возможности, количества необходимых взаимодействий потоков – так, например, если решается задача суммирования значений некоторого числового набора данных, то вместо использования единственной общей переменной для накопления суммы можно сначала вычислить частные суммы для каждого потока в отдельности (без каких-либо синхронизаций), и только потом собрать эти частные суммы вместе.

– Повышение эффективности алгоритмов, выполняемых в критических секциях потоков, и, тем самым, сокращение времени блокировки общих ресурсов; например, если в критической секции необходимо упорядочить данные, то, конечно же, необходимо использовать быстрые алгоритмы сортировки.

– Разделение общих ресурсов для разбиения единственной критической секции на множество раздельных и более редко используемых подсекций – так, при использовании таблицы данных можно организовать блокировку для каждой строки таблицы в отдельности.

– Обеспечение более быстрого выполнения потоков, которые находятся в своих критических секциях – для этого, в частности, можно запретить приостановку таких потоков или повысить их приоритет; применение таких методов обеспечивается обычно средами выполнения параллельных программ.

2.6.3. Оптимизация работы с памятью

Обеспечение эффективного использования памяти является общей проблемой программирования. И ключевой момент в этой проблеме – оптимизация работы с кэш-памятью, поскольку время доступа в этой памяти существенно (в 10–100 раз) меньше по сравнению с оперативной памятью. Основной способ достижения эффективности – это обеспечение локальности использования данных, когда информация после считывания в кэш-память многократно используется без обращения к медленной оперативной памяти. При этом следует учитывать, что перемещение данных в кэш осуществляется небольшими блоками – *строками* (*cache line*). Размер строк кэш-памяти обычно составляет 64–128 байт. Как результат, после считывания некоторого значения в кэш, соседние элементы также оказываются в кэш и их обработка уже не потребует доступа к оперативной памяти.

Учет этого момента также позволяет значительно повысить эффективность работы программы. Так, например, при работе с матрицами данных более эффективно проводить обработку элементов по строкам, а не по столбцам (данное утверждение справедливо для алгоритмического языка С, в котором матрицы располагаются в памяти по строкам).

Полное рассмотрение вопросов оптимизации использования памяти выходит за пределы данной книги – дополнительная информация по этой проблеме может быть получена, например, в [16].

При параллельном программировании для систем с общей памятью возникают дополнительные аспекты эффективного использования памяти, связанные с перемещением данных между имеющимися несколькими устройствами кэш-памяти.

Обеспечение однозначности кэш-памяти

Первый дополнительный аспект – это упоминавшаяся уже в главе 1 проблема обеспечения *однозначности* (*когерентности*) памяти при наличии в системе нескольких устройств кэш-памяти. Напомним суть проблемы: при изменении каким-либо вычислительным ядром/процессором значения общей переменной, копии которой находятся в нескольких устройствах кэш-памяти, необходимо обновить значение этой переменной для всех ее копий (или запретить использование «устаревших» копий). Данный аспект связан, скорее всего, не с эффективностью, а с корректностью выполнения параллельных программ, и обычно обеспечивается на аппаратном уровне.

Уменьшение миграции потоков между ядрами/процессорами

Другой дополнительный аспект состоит в возможности возникновения дополнительных перемещений данных между разными устройствами кэш-памяти при смене вычислительного ядра/процессора для выполнения потоков. Для устранения этого эффекта в современных системах обычно имеются средства для обеспечения связанности (*processor affinity*) потоков и используемых для их выполнения вычислительных устройств. Однако использование таких средств является весьма непростым делом, поскольку одновременно необходимо обеспечить и масштабируемость вычислений (эффективность выполнения для разных конфигурациях имеющихся вычислительных ресурсов). Более простой способ снижения подобного эффекта может состоять в использовании количества потоков, совпадающего с числом имеющихся вычислительных устройств (см. также п. 2.6.1).

Устранение эффекта ложного разделения данных

Еще один очень интересный аспект, связанный с наличием нескольких устройств кэш-памяти, состоит в возможности появления так называемого эффекта *ложного разделения данных* (*false sharing*), когда части одной и той же строки кэш-памяти оказываются разделенными между разными кэшами. В этом случае любое изменение данных приводит к необходимости передачи строк кэш-памяти между устройствами кэш-памяти (однозначность памяти обеспечивается не на уровне отдельных переменных, а для полных строк кэш-памяти). Подобная синхронизация является излишней, если обрабатываемых данные на разных ядрах/процессорах не пересекаются. Возникновение такого эффекта может существенно снизить эффективность вычислений и для его недопущения достаточно обеспечить несмежное размещение в памяти данных, обрабатываемых разными вычислительными устройствами.

2.6.4. Использование потоко-ориентированных библиотек

В завершение рассмотрения вопросов повышения эффективности многопоточных параллельных программ – общая рекомендация по полезности максимально-возможного использования программных библиотек, специально разработанных для вычислительных

систем с общей памятью. Использование таких библиотек может быть и обязательным условием для корректности выполнения многопоточных программ – так, например, при построении программы компилятор должен использовать потоко-ориентированные версии служебных программ среды выполнения (необходимость поддержки многопоточности указывается компилятору при помощи соответствующих управляющий ключей). А с другой стороны, применение уже имеющихся многопоточных библиотек крайне полезно – как правило, имеющиеся в этих библиотеках реализации параллельных методов являются высокоэффективными. Кроме того, использование библиотек позволяет существенно снизить затраты на разработку необходимого параллельного программного обеспечения.

2.7. Краткий обзор главы

Данная глава посвящена рассмотрению основных аспектов параллельного программирования.

В 2.1 рассмотрен ряд понятий и определений, являющихся основополагающими для параллельного программирования. Среди таких понятий – концепция *процессов*, *потоков* и *ресурсов*. С использованием введенных понятий *параллельные программы* могут быть представлены как *системы параллельно выполняемых процессов и потоков*.

В 2.2 проведено последовательное рассмотрение возможных способов организации взаимоисключения параллельно выполняемых потоков. Сначала в разделе даны «очевидные», на первый взгляд, решения, которые на самом деле являются неполными, однако позволяют продемонстрировать ряд проблем, которые могут возникать при разработке параллельных программ – *жесткая синхронизация*, *потеря взаимоисключения*, *возможность блокировки*, *бесконечное откладывание*. Далее приведено полное решение задачи взаимоисключения – *алгоритм Деккера*. И в завершение в разделе рассматриваются классические механизмы организации взаимоисключения – *семафоры Дейкстры* и *мониторы Хоара*.

В 2.3 проведено изучение проблемы синхронизации параллельно выполняемых потоков, для решения которой приводятся два основных наиболее широко используемых подхода – использование *условных переменных* и организация б*арьерной синхронизации*.

В 2.4 рассмотрена одна из основных проблем параллельного программирования – возникновения в ходе параллельных вычислений ситуаций *взаимоблокировки* потоков, когда потоки не могут продолжить свое выполнение из-за конкуренции за общие разделяемые ресурсы (такие ситуации в литературе называются также как *тупики*, *дедлоки* или *смертельные объятия*). При изучении проблемы определяются условия возникновения тупиковых ситуаций. Далее была рассмотрена модель программы в виде *графа* «*поток–ресурс*», которая позволяет анализировать процесс выполнения параллельных программ и определять наличие условий возникновения тупиков.

В 2.5 рассмотрен ряд иллюстративных примеров, которые принято считать в качестве классических задач параллельного программирования, поскольку они позволяют продемонстрировать многие проблемы, возникающие при разработке параллельных алгоритмов и программ, и предоставляют возможность наглядно показать основные способы решения этих проблем. В числе этих задач:

- Задача «Производители–Потребители» (*Producer-Consumer problem*);
- Задача «Читатели–Писатели» (*Readers-Writers problem*);
- Задача «Обедающие философы» (*Dining Philosopher problem*);
- Задача «Спящий брадобрей» (*Sleeping Barber problem*).

В 2.6 излагается ряд практических методов повышения эффективности параллельных программ. К числу рассматриваемых методов относится оптимизация количества потоков,

минимизация взаимодействия потоков, оптимизация работы с памятью и широкое использование ранее разработанных библиотек параллельных методов.

2.8. Обзор литературы

Дополнительная информация по вопросам, изложенным в данной главе, может быть получена, например, в [39]. Проблемы параллельного программирования применительно к тематике операционных систем широко рассмотрены в [5,15,27,31–33]. Вопросы моделирования процессов выполнения параллельных программ излагаются в [36].

Для рассмотрения вопросов организации параллельных программ применительно к операционной системе Windows могут быть рекомендованы работы [22,25], для изучения этих же вопросов для ОС Unix может быть рекомендована работа [26].

2.9. Контрольные вопросы

1. В чем состоят понятия процесса и потока? Укажите схожесть и различия этих понятий.

2. В чем состоит понятие ресурса? Приведите примеры различных типов ресурсов.

3. Дайте общую характеристику представления параллельных программ как системы параллельно выполняемых потоков.

4. Какие основные предположения могут быть сделаны о характере временных соотношений между выполняемыми командными последовательностями разных потоков?

5. Чем определяется повышенная сложность параллельного программирования?

6. В чем состоит проблема взаимоисключения потоков? Какие основные требования к методам решения этой проблемы?

7. В чем состоит недостаток метода жесткой синхронизации при организации взаимоисключения потоков?

8. Приведите примеры некорректного решения проблемы взаимоисключения потоков, при котором происходит потеря взаимоисключения.

9. Приведите примеры некорректного решения проблемы взаимоисключения потоков, при котором возможно возникновение взаимоблокировки потоков.

10. Приведите примеры некорректного решения проблемы взаимоисключения потоков, при котором возможна ситуация бесконечного откладывания доступа к критическим секциям.

11. В чем состоит алгоритм Деккера для решения проблемы взаимоисключения потоков?

12. В чем состоит концепция семафоров Дейкстры? Приведите пример решения проблемы взаимоисключения потоков с использованием семафоров.

13. В чем состоит концепция мониторов Хоара? Приведите пример решения проблемы взаимоисключения потоков с использованием мониторов.

14. В чем состоит проблема синхронизации потоков?

15. Как решается проблема синхронизации потоков при помощи условных переменных?

16. Как решается проблема синхронизации потоков при помощи метода барьерной синхронизации?

17. В чем состоит проблема взаимоблокировки потоков? Укажите необходимые условия возникновения тупиков.

18. В чем состоит модель параллельных программ в виде графа «поток–ресурс»?

19. Какие свойства параллельных программ могут быть получены в результате анализа графа «поток–ресурс»?

20. Дайте общую характеристику и приведите возможное решение задачи «Производители–Потребители» (*Producer-Consumer problem*).

21. Дайте общую характеристику и приведите возможное решение задачи «Читатели–Писатели» (*Readers-Writers problem*).

22. Дайте общую характеристику и приведите возможное решение задачи «Обедающие философы» (*Dining Philosopher problem*).

23. Дайте общую характеристику и приведите возможное решение задачи «Спящий брадобрей» (*Sleeping Barber problem*).

24. Каким образом оптимизация количества потоков может повысить эффективность выполнения параллельных программ?

25. Каким образом минимизация взаимодействия потоков может повысить эффективность выполнения параллельных программ?

26. Каким образом оптимизация работы с памятью может повысить эффективность выполнения параллельных программ?

27. Каким образом использование библиотек параллельных методов может повысить эффективность выполнения параллельных программ?

2.10. Задачи и упражнения

1. Изучите методы синхронизации и взаимоисключения потоков для операционной системы Windows и разработайте ряд демонстрационных параллельных программ.

2. Изучите методы синхронизации и взаимоисключения потоков для операционной системы Unix/Linux и разработайте ряд демонстрационных параллельных программ.

3. Изучите методы синхронизации и взаимоисключения потоков для стандарта POSIX.

4. Изучите методы синхронизации и взаимоисключения потоков для технологии OpenMP.

5. Разработайте несколько параллельных программ для решения задачи «Производители–Потребители» с использованием разных механизмов синхронизации и взаимоисключения потоков.

6. Разработайте несколько параллельных программ для решения задачи «Читатели–Писатели» с использованием разных механизмов синхронизации и взаимоисключения потоков.

7. Разработайте несколько параллельных программ для решения задачи «Обедающие философы» с использованием разных механизмов синхронизации и взаимоисключения потоков.

8. Разработайте несколько параллельных программ для решения задачи «Спящий брадобрей» с использованием разных механизмов синхронизации и взаимоисключения потоков.

9. Разработайте несколько параллельных программ для демонстрации способов оптимизации количества потоков для повышения эффективности выполнения параллельных программ.

10. Разработайте несколько параллельных программ для демонстрации способов минимизации взаимодействия потоков для повышения эффективности выполнения параллельных программ.

11. Разработайте несколько параллельных программ для демонстрации способов оптимизации работы с памятью для повышения эффективности выполнения параллельных программ.

12. Разработайте несколько параллельных программ для демонстрации результативности использования библиотек параллельных методов для повышения эффективности выполнения параллельных программ.