Глава 7. Организация параллельных взаимодействующих вычислений

Мультипрограммные и мультизадачные операционные системы позволяют организовать не только независимые, но взаимодействующие вычисления. Сама операционная система как комплекс управляющих и обрабатывающих программных модулей также функционирует как множество взаимодействующих вычислений. Проблема синхронизации взаимодействия параллельных вычислительных процессов, обмена данными между ними является одной из важнейших. Существующие методы синхронизации вычислений и обмена сообщениями различаются по таким параметрам, как удобство программирования параллельных процессов, стоимость реализации, эффективность функционирования созданных приложений и всей вычислительной системы в целом. Операционные системы имеют в своем составе различные средства синхронизации. Знание этих средств и их правильное использование позволяет создавать программы, которые при работе осуществляют корректный обмен информацией, а также исключают возможность возникновения тупиковых ситуаций.

В этой главе рассматриваются основные понятия и проблемы, характерные для параллельных процессов. Описываются основные механизмы синхронизации, дается их сравнительный анализ, приводятся характерные примеры программ, использующих данные механизмы.

Независимые и взаимодействующие вычислительные процессы

Основной особенностью мультипрограммных операционных систем является то, что в их среде параллельно развивается несколько (последовательных) вычислительных процессов. С точки зрения внешнего наблюдателя эти последовательные

вычислительные процессы выполняются одновременно, мы же будем говорить «параллельно». При этом под *параллельными* понимаются не только процессы, одновременно развивающиеся на различных процессорах, каналах и устройствах ввода-вывода, но и те последовательные процессы, которые разделяют центральный процессор и в своем выполнении хотя бы частично перекрываются во времени. Любая мультизадачная операционная система вместе с параллельно выполняющимися в ней задачами может быть логически представлена как совокупность последовательных вычислений, которые, с одной стороны, состязаются за ресурсы, переходя из одного состояния в другое, а с другой — действуют почти независимо один от другого, но при этом образуя единую систему посредством установления разного рода связей между собой (путем пересылки сообщений и синхронизирующих сигналов).

Итак, *параллельными* мы будем называть такие последовательные вычислительные процессы, которые одновременно находятся в каком-нибудь активном состоянии. Два параллельных процесса могут быть *независимыми* (independed processes) либо *взаимодействующими* (cooperating processes).

Независимыми являются процессы, множества переменных которых не пересекаются. Под переменными в этом случае понимают файлы данных, а также области оперативной памяти, сопоставленные промежуточным и определенным в программе переменным. Независимые процессы не влияют на результаты работы друг друга, так как не могут изменять значения переменных друг у друга. Они могут только явиться причиной в задержках исполнения друг друга, так как вынуждены разделять ресурсы системы.

Взаимодействующие процессы совместно используют некоторые (общие) переменные, и выполнение одного процесса может повлиять на выполнение другого. Как мы уже говорили, при выполнении вычислительные процессы разделяют ресурсы системы. Подчеркнем, что при рассмотрении вопросов синхронизации вычислительных процессов из числа разделяемых ими ресурсов исключаются центральный процессор и программы, реализующие эти процессы, то есть с логической точки зрения каждому процессу соответствуют свои процессор и программа, хотя в реальных системах обычно несколько процессов разделяют один процессор и одну или несколько программ. Многие ресурсы вычислительной системы могут совместно использоваться несколькими процессами, но в каждый момент времени к разделяемому ресурсу может иметь доступ только один процесс. Ресурсы, которые не допускают одновременного использования несколькими процессами, называются *критическими*.

Если несколько вычислительных процессов хотят пользоваться критическим ресурсом в режиме разделения, им следует синхронизировать свои действия таким образом, чтобы ресурс всегда находился в распоряжении не более чем одного из них. Если один процесс пользуется в данный момент критическим ресурсом, то все остальные процессы, которым нужен этот ресурс, должны ждать, пока он не освободится. Если в операционной системе не предусмотрена защита от одновременного доступа процессов к критическим ресурсам, в ней могут возникать ошибки, которые трудно обнаружить и исправить. Основной причиной возникновения этих ошибок является то, что процессы в мультипрограммных операционных сис-

темах развиваются с различными скоростями и относительные скорости развития каждого из взаимодействующих процессов не подвластны и не известны ни одному из них. Более того, на их скорости могут влиять решения планировщиков, касающиеся других процессов, с которыми ни одна из этих программ не взаимодействует. Кроме того, содержание и скорость исполнения одного из них обычно не известны другому процессу. Поэтому влияние, которое оказывают друг на друга взаимодействующие процессы, не всегда предсказуемо и воспроизводимо.

Взаимодействовать могут либо конкурирующие процессы, либо процессы, обрабатывающие информацию совместно. *Конкурирующие процессы* действуют относительно независимо друг от друга, однако они имеют доступ к некоторым общим переменным. Их независимость заключается в том, что они могут работать друг без друга, поодиночке. Но при своем выполнении они могут работать и параллельно, и тогда они иногда начинают конкурировать при обращении к этим общим переменным. Таким образом, их независимость относительна.

Приведем несколько наиболее известных примеров конкурирующих процессов и продемонстрируем появление ошибок. В качестве первого примера рассмотрим работу двух процессов P1 и P2 с общей переменной X. Пусть оба процесса асинхронно, независимо один от другого, изменяют (например, увеличивают) значение переменной X, считывая ее значение в локальную область памяти Ri[1], при этом каждый процесс выполняет во времени некоторые последовательности операций (табл. 7. 1). Здесь мы рассмотрим не все операторы каждого из процессов, а только те, в которых осуществляется работа с общей переменной X. Каждому из операторов мы присвоили некоторый условный номер.

Таблица 7. 1. Пример конкурирующих процессов

Номер оператора	Процесс P1	Номер оператора	Процесс P2
1	R1 := X	4	R2 := X
2	R1 := R1 + 1	5	R2 := R2 + 1
3	X := R1	6	X := R2

Поскольку при мультипрограммировании процессы могут иметь различные скорости исполнения, то может иметь место любая последовательность выполнения операций во времени. Например, если сначала будут выполнены все операции процесса P1, а уже потом — все операции процесса P2 (рис. 7. 1) или, наоборот, сначала — операции 4-6, а затем — операции 1-3, то в итоге переменная X получит значение, равное X + 2.

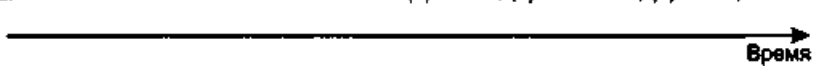

Рис. 7. 1. Первый вариант развития событии при выполнении процессов

[1] Ri — это просто имя переменной для процесса с номером i.

Однако если в промежуток времени между выполнением операций 1 и 3 будет выполнена хотя бы одна из операций 4-6 (рис. 7.2), то значение переменной X после выполнения всех операций будет не (X + 2), а (X + 1).

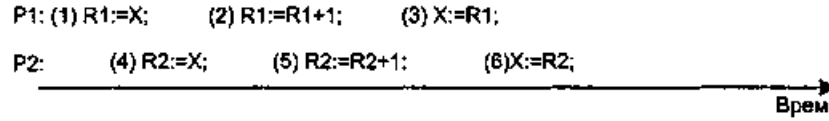

Рис. 7.2. Второй вариант развития событий при выполнении процессов

Понятно, что это очень серьезная ошибка. Например, если бы процессы P1 и P2 осуществляли продажу билетов и переменная X фиксировала количество уже проданных, то в результате некорректного взаимодействия было бы продано несколько билетов на одно и то же место. К сожалению, такого рода ошибки трудноуловимы, поскольку они иногда возникают, иногда нет.

В качестве второго примера рассмотрим ситуацию, которая еще совсем недавно была достаточно актуальной для первых персональных компьютеров. Пусть на персональном компьютере с простейшей однопрограммной операционной системой (типа MS DOS) установлена некоторая резидентная программа с условным названием TIME, которая по нажатию комбинации клавиш (например, Ctrl+T) воспроизводит на экране дисплея время в виде 18:20:59, и допустим, что значения переменных, обозначающих час, минуты и секунды, равны 18, 20 и 59 соответственно, причем вывод на дисплей осуществляется справа налево (сначала секунды, затем минуты и, наконец, часы). Пусть сразу же после передачи программой TIME на дисплей информации «59 секунд» генерируется прерывание от таймера, и значение времени обновляется: 18:21:00.

После этого программа TIME, прерванная таймером, продолжит свое выполнение, и на дисплей будут выданы значения: минуты (21), часы (18). В итоге на экране мы увидим: 18:21:59.

Рассмотрим теперь несколько иной случай развития событий обновления значений времени по сигналу таймера. Если программа ведения системных часов после вычислений количества секунд 59 + 1 = 60 и замены его на 00 прерывается от нажатия клавиш Ctrl+T, то есть программа не успевает осуществить пересчет количества минут, то время, индицируемое на дисплее, станет равным 18:20:00. И в этом случае мы получим неверное значение времени.

Наконец, в качестве третьего примера приведем пару процессов, которые изменяют различные поля записей служащих какого-либо предприятия [17]. Пусть процесс АДРЕС изменяет домашний адрес служащего, а процесс СТАТУС — его должность и зарплату. Пусть каждый процесс для выполнения своей функции копирует всю запись СЛУЖАЩИЙ в свою рабочую область. Предположим, что каждый процесс должен обработать некоторую запись ИВАНОВ. Предположим также, что после того как процесс АДРЕС скопировал запись ИВАНОВ в свою рабочую область, но до того как он записал скорректированную запись обратно, процесс СТАТУС скопировал первоначальную запись ИВАНОВ в свою рабочую область.

Изменения, выполненные тем из процессов, который первым запишет скорректированную запись назад в файл СЛУЖАЩИЕ, будут утеряны, и, возможно, никто об этом не узнает.

Чтобы предотвратить некорректное исполнение конкурирующих процессов вследствие нерегламентированного доступа к разделяемым переменным, необходимо ввести понятие *взаимного исключения,* согласно которому два процесса не должны одновременно обращаться к разделяемым переменным.

Процессы, выполняющие общую совместную работу таким образом, что результаты вычислений одного процесса в явном виде передаются другому, то есть они обмениваются данными и именно на этом построена их работа, называются *сотрудничающими.* Взаимодействие сотрудничающих процессов удобнее всего рассматривать в схеме *производитель-потребитель* (produces-consumer), или, как часто говорят, *поставщик-потребитель.*

Кроме реализации в операционной системе средств, организующих взаимное исключение и, тем самым, регулирующих доступ процессов к критическим ресурсам, в ней должны быть предусмотрены средства, синхронизирующие работу взаимодействующих процессов. Другими словами, процессы должны обращаться друг к другу не только ради синхронизации с целью взаимного исключения при обращении к критическим ресурсам, но и ради обмена данными.

Допустим, что «поставщик» — это процесс, который отправляет порции информации (сообщения) другому процессу, имя которого — «потребитель». Например, некоторая задача пользователя, порождающая данные для вывода их на печать, может выступать как поставщик, а системный процесс, который выводит эти строки на устройство печати, — как потребитель. Один из методов, применяемых при передаче сообщений, состоит в том, что заводится *пул* (pool)[1] свободных буферов, каждый из которых может содержать одно сообщение. Заметим, что длина сообщения может быть произвольной, но ограниченной размером буфера.

В этом случае между процессами «поставщик» и «потребитель» будем иметь очередь заполненных буферов, содержащих сообщения. Когда поставщик хочет послать очередное сообщение, он добавляет в конец этой очереди еще один буфер. Потребитель, чтобы получить сообщение, забирает из очереди буфер, который стоит в ее начале. Такое решение, хотя и кажется тривиальным, требует, чтобы поставщик и потребитель синхронизировали свои действия. Например, они должны следить за количеством свободных и заполненных буферов. Поставщик может передавать сообщения только до тех пор, пока имеются свободные буферы. Аналогично, потребитель может получать сообщения, только если очередь не пуста. Ясно, что для учета заполненных и свободных буферов нужны разделяемые переменные, поэтому, так же как и для конкурирующих процессов, для сотрудничающих процессов тоже возникает необходимость во взаимном исключении.

Таким образом, до окончания обращения одной задачи к общим переменным следует исключить возможность обращения к ним другой задачи. Эта ситуация и на-

[1] Совокупность однородных динамически распределяемых объектов, например блоков памяти одинаковой длины.

зывается взаимным исключением. Другими словами, при организации различного рода взаимодействующих процессов приходится организовывать взаимное исключение и решать проблему корректного доступа к общим переменным (критическим ресурсам). Те места в программах, в которых происходит обращение к критическим ресурсам, называются *критическими секциями* (Critical Section, CS). Решение проблемы заключается в организации такого доступа к критическому ресурсу, при котором только одному процессу разрешается входить в критическую секцию. Данная задача только на первый взгляд кажется простой, ибо критическая секция, вообще говоря, не является последовательностью операторов программы, а является процессом, то есть последовательностью действий, которые выполняются этими операторами. Другими словами, несколько процессов могут выполнять критические секции, использующие одну и ту же последовательность операторов программы.

Когда какой-либо процесс находится в своей критической секции, другие процессы могут, конечно, продолжать свое исполнение, но без входа в их критические секции. Взаимное исключение необходимо только в том случае, когда процессы обращаются к разделяемым (общим) данным. Если же они выполняют операции, которые не ведут к конфликтным ситуациям, процессы должны иметь возможность работать параллельно. Когда процесс выходит из своей критической секции, то одному из остальных процессов, ожидающих входа в свои критические секции, должно быть разрешено продолжить работу (если в этот момент действительно есть процесс в состоянии ожидания входа в свою критическую секцию).

Обеспечение взаимного исключения является одной из ключевых проблем параллельного программирования. При этом можно перечислить требования к критическим секциям [17, 54].

* В любой момент времени только один процесс должен находиться в своей критической секции.
* Ни один процесс не должен бесконечно долго находиться в своей критической секции.
* Ни один процесс не должен бесконечно долго ожидать разрешение на вход в свою критическую секцию. В частности:
  - никакой процесс, бесконечно долго находящийся вне своей критической секции (что допустимо), не должен задерживать выполнение других процессов, ожидающих входа в свои критические секции (другими словами, процесс, работающий вне своей критической секции, не должен блокировать критическую секцию другого процесса);
  - если два процесса хотят войти в свои критические секции, то принятие решения о том, кто первым войдет в критическую секцию, не должно быть бесконечно долгим.
* Если процесс, находящийся в своей критической секции, завершается естественным или аварийным путем, то режим взаимного исключения должен быть отменен, с тем чтобы другие процессы получили возможность входить в свои критические секции.

Было предложено несколько способов решения этой проблемы: программных и аппаратных; локальных низкоуровневых и глобальных высокоуровневых; предусматривающих свободное взаимодействие между процессами и требующих строгого соблюдения протоколов.

Средства синхронизации и связи взаимодействующих вычислительных процессов

Все известные средства решения проблемы взаимного исключения основаны на использовании специально введенных аппаратных возможностей. К этим аппаратным возможностям относятся: блокировка памяти, специальные команды типа «проверка и установка» и механизмы управления системой прерываний, которые позволяют организовать такие средства, как семафорные операции, мониторы, почтовые ящики и др. С помощью перечисленных средств можно разрабатывать взаимодействующие процессы, при исполнении которых будут корректно решаться все задачи, связанные с проблемой критических секций. Рассмотрим эти средства в следующем порядке по мере их усложнения, перехода к функциям операционной системы и увеличения предоставляемых ими удобств, опираясь на уже древнюю, но все же еще достаточно актуальную работу Дейкстры [10]. Заметим, что этот материал позволяет в полной мере осознать проблемы, возникающие при организации параллельных взаимодействующих вычислений. Эта работа Дейкстры полезна, прежде всего, с методической точки зрения, поскольку она позволяет понять наиболее тонкие моменты в этой проблематике.

Использование блокировки памяти при синхронизации параллельных процессов

Все вычислительные машины и системы (в том числе и с многопортовыми блоками оперативной памяти) имеют средство для организации взаимного исключения, называемое *блокировкой памяти*. Блокировка памяти запрещает одновременное исполнение двух (и более) команд, которые обращаются к одной и той же ячейке памяти. Блокировка памяти имеет место всегда, то есть это обязательное условие функционирования компьютера. Соответственно, поскольку в некоторой ячейке памяти хранится значение разделяемой переменной, то получить доступ к ней может только один процесс, несмотря на возможное совмещение выполнения команд во времени на различных процессорах (или на одном процессоре, но с конвейерной организацией параллельного выполнения команд).

Механизм блокировки памяти предотвращает одновременный доступ к разделяемой переменной, но не предотвращает чередование доступа. Таким образом, если критические секции исчерпываются одной командой обращения к памяти, данное средство может быть достаточным для непосредственной реализации взаимного исключения. Если же критические секции требуют более одного обращения к памяти, то задача становится сложной, но алгоритмически разрешимой. Рассмотрим

различные попытки использования механизма блокировки памяти для организации взаимного исключения при выполнении критических секций и покажем некоторые важные моменты, пренебрежение которыми приводит к неприемлемым или даже к ошибочным решениям.

Возможные проблемы при организации взаимного исключения при условии использования только блокировки памяти

Пусть имеется два или более циклических процессов с абстрактными критическими секциями, то есть каждый процесс состоит из двух частей: некоторой критической секции и оставшейся части кода, которая не работает с общими (критическими) переменными. Пусть эти два процесса асинхронно разделяют во времени единственный процессор либо выполняются на отдельных процессорах, то есть каждый из них имеет доступ к некоторой общей области памяти, с которой и работают критические секции. Проиллюстрируем эту ситуацию с помощью рис. 7.3.

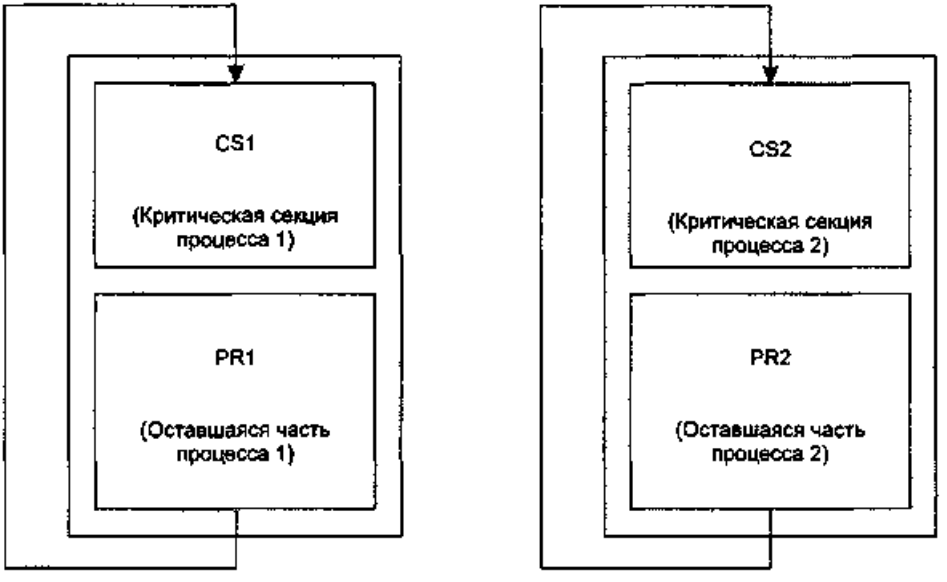

Рис. 7.3. Модель взаимодействующих процессов

Задача вроде бы легко решается, если потребовать, чтобы процессы ПР1 и ПР2 входили в свои критические секции попеременно. Для этого одна общая переменная может хранить указатель на тот процесс, чья очередь войти в критическую секцию. Текст этого решения на языке, близком к Паскалю, приведен в листинге 7.1.

Листинг 7.1. Текст программы для первого решения

```
Var перекл   integer
Begin перекл = 1  (при перекл=1 в критической секции находится процесс ПР1}
Parbegin
```

```
  While true do
    Begin
      while перекл = 2 do begin end
      CS1  { критическая секция процесса ПР1 }
      перекл  =  2.
      PR1  { оставшаяся часть процесса ПР1 }
    End
And
  While true do
    Begin
      while перекл = 1 do begin end.
      CS2. { критическая секция процесса ПР2 }
      перекл  = 1.
      PR2. { оставшаяся часть процесса ПР2 }
    End
Parend
End
```

Здесь и далее языковая конструкция следующего типа означает параллельность выполнения K описываемых последовательных процессов:

```
parbegin   S11   S 1 2 . .  S1N1
  and     S21   S 2 2 . .  S2N2

  and    SK1    S K 2 . .  SKNk
parend
```

Конструкция из операторов S11; S12;...; S1N1 выполняется последовательно (оператор за оператором), о чем свидетельствует наличие точки с запятой между ними.

Следующая языковая конструкция означает, что каждый процесс может выполняться неопределенно долгое время фактически бесконечное количество раз:

```
while true do
begin S1, S2, SN end
```

Наконец, конструкция типа begin end означает просто «пустой» оператор.

Итак, решение, представленное в листинге 7.1, обеспечивает нам взаимное исключение в работе критических секций. Однако если бы фрагмент программы PR1 был намного длиннее, чем фрагмент PR2, или если бы процесс ПР1 был заблокирован в секции PR1, или если бы процессор для ПР2 обладал более высоким быстродействием, то процесс ПР2 вскоре вынужден был бы ждать входа в свою критическую секцию CS2, хотя процесс ПР1 и был бы вне CS1. Такое ожидание могло бы оказаться слишком долгим, то есть для этого решения один процесс вне своей критической секции может помешать другому войти в свою критическую секцию.

Попробуем устранить это блокирование с помощью двух общих переменных, которые будут использоваться как флаги, указывая, находится или нет соответствующий процесс в своей критической секции. То есть с каждым из процессов ПР1 и ПР2 будет связана переменная, которая принимает значение true, когда процесс находится в своей критической секции, и false — в противном случае. Прежде чем войти в свою критическую секцию, процесс проверит значение флага другого процесса. Если это значение равно true, процессу не разрешается входить в свою критическую секцию. В противном случае процесс установит собственный флаг и вой-

дет в критическую секцию. Этот алгоритм взаимного исключения представлен в листинге 7.2.

Листинг 7.2. Второй вариант реализации взаимного исключения

```
Var перекл1, перекл2   boolean
begin перекл1 = false
  перекл2 = false,
parbegin
   while true do
    begin
      while перекл2 do
        begin
        end
      перекл1 =true
      CS1 { критическая секция процесса ПР1 }
      перекл1 = false
      PR1 { процесс ПР1 после критической секции }
    end
and
   while true do
    begin
      while перекл1 do
        begin
        end
      перекл2 =true
        CS2 { Критическая секция процесса ПР2 }
      перекл2 = false
        PR2 { процесс ПР2 после критической секции }
    end
parend
end
```

Данный алгоритм, увы, не гарантирует полного выполнения условия нахождения только одного процесса внутри критической секции. Отсутствие гарантий связано с различными, в общем случае, скоростями развития процессов. Поэтому, например, между проверкой значения переменной перекл2 процессом ПР1 и последующей установкой им значения переменной перекл1 параллельно выполняющийся процесс ПР2 может установить перекл2 в значение true, так как переменная перекл! еще не успела установиться в значение true Отсюда следует, что оба процесса могут войти в свои критические секции одновременно.

Следующий (третий) вариант решения этой задачи (листинг 7.3) усиливает взаимное исключение, так как в процессе ПР1 проверка значения переменной перекл2 выполняется только после установки переменной перекл1 в значение true (аналогично для ПР2).

Листинг 7.3. Третий вариант реализации взаимного исключения

```
var перекл1 , перекл2     boolean
begin перекл1 =false   перекл2 = false
parbegin
  ПР1  while true do
       begin
         перекл1 = true
```

```
        while перекл2 do
          begin  end.
        CS1 { критическая секция процесса ПР1 }
        перекл1 =false,
        PR1 { ПР1 после критической секции }
      end
and
  ПР2  while true do
      begin
        перекл2 = true,
        while перекл1 do
          begin  end,
        CS2 { критическая секция процесса ПР2 }
        перекл2 = false,
        PR2 { ПР2 после критической секции }
      end
parend
end
```

Алгоритм, приведенный в листинге 7.3, также имеет свои недостатки. Действительно, возможна ситуация, когда оба процесса одновременно установят свои флаги в значение true и войдут в бесконечный цикл. В этом случае будет нарушено требование отсутствия бесконечного ожидания входа в свою критическую секцию. То есть, предположив, что скорости исполнения процессов произвольны, мы получили такую последовательность событий, при которой процессы вообще перестанут нормально выполняться.

Все рассмотренные попытки решить задачу взаимного исключения при выполнении критических секций иллюстрируют нам некоторые тонкие моменты, лежащие в основе этой проблемы, и показывают, что не все так просто.

Последний рассматриваемый вариант решения задачи взаимного исключения, опирающийся только на блокировку памяти, — это известный алгоритм, предложенный математиком Деккером.

Алгоритм Деккера

Алгоритм Деккера основан на использовании трех переменных (листинг 7.4): перекл1, перекл2 и ОЧЕРЕДЬ. Пусть по-прежнему переменная перекл1 устанавливается в true тогда, когда процесс ПР1 хочет войти в свою критическую секцию (для ПР2 аналогично), а значение переменной ОЧЕРЕДЬ указывает, чье сейчас право сделать попытку входа при условии, что оба процесса хотят выполнить свои критические секции.

Листинг 7.4. Алгоритм Деккера

```
label 1, 2.
var перекл1, перекл2 boolean,
  ОЧЕРЕДЬ integer.
Begin перекл1 = false, перекл2 =false
  ОЧЕРЕДЬ =1.
  parbegin
    while true do
      begin                перекл1              =true.          продолжение &
```

Листинг 7.4 *(продолжение)*

```
    1   if перекл2 =true then
          if ОЧЕРЕДЬ=1 then go to 1
            else begin перекл1 =false
                 while ОЧЕРЕДЬ=2 do
                   begin    end
                 end
        else begin
               CS1 { критическая секция ПР1 }
               ОЧЕРЕДЬ =2 перекл1 =false
             end
      end
  and
    while true do
      begin перекл2 =1
    2   if перекл1 =true then
          if ОЧЕРЕДЬ=2 then go to 2
            else begin перекл2 =false
                   while ОЧЕРЕДЬ=1 do
                   begin    end
                 end
        else begin
               CS2 { критическая секция ПР2 }
               ОЧЕРЕДЬ =1 перекл2 =false
             end
      end
  parend
end
```

Если перекл2 = true и перекл1 = false, то выполняется критическая секция процесса ПР2 независимо от значения переменной ОЧЕРЕДЬ. Аналогично для случая перекл2 = false и перекл1 =true.

Если же оба процесса хотят выполнить свои критические секции, то есть перекл2 = = true и перекл1 = true, то выполняется критическая секция того процесса, на который указывает значение переменной ОЧЕРЕДЬ, независимо от скоростей развития обоих процессов. Использование переменной ОЧЕРЕДЬ совместно с переменными перекл1 и перекл2 в алгоритме Деккера позволяет гарантированно решать проблему критических секций. То есть переменные перекл1 и перекл2 гарантируют, что взаимное выполнение не может иметь места; переменная ОЧЕРЕДЬ гарантирует, что не может быть взаимной блокировки, так как переменная ОЧЕРЕДЬ не меняет своего значения во время выполнения программы принятия решения о том, кому же сейчас проходить свою критическую секцию.

Тем не менее реализаций критических секций на основе описанного алгоритма практически не встречается из-за их чрезмерной сложности, особенно тогда, когда требуется обобщить алгоритм Деккера с двух до N процессов.

Синхронизация процессов с помощью операции проверки и установки

Операция проверки и установки является, так же как и блокировка памяти, одним из аппаратных средств, которые могут быть использованы для решения задачи вза-

имного исключения при выполнении критической секции. Данная операция реализована во многих компьютерах. Так, в знаменитой машине IBM 360 (370) эта команда называлась TS (Test and Set — проверка и установка). Команда TS является двухадресной (двухоперандной). Ее действие заключается в том, что процессор присваивает значение второго операнда первому, после чего второму операнду присваивается значение, равное единице. Команда TS является неделимой операцией, то есть между ее началом и концом не могут выполняться никакие другие команды.

Чтобы использовать команду TS для решения проблемы критической секции, свяжем с ней переменную common, которая будет общей для всех процессов, использующих некоторый критический ресурс. Данная переменная будет принимать единичное значение, если какой-либо из взаимодействующих процессов находится в своей критической секции. Кроме того, с каждым процессом свяжем свою локальную переменную, которая принимает значение, равное единице, если данный процесс хочет войти в свою критическую секцию. Операция TS будет присваивать значение common локальной переменной и устанавливать common в единицу. Соответствующая программа решения проблемы критической секции на примере двух параллельных процессов приведена в листинге 7 5.

Листинг 7. 5. Взаимное исключение с помощью операции проверки и установки

```
var common, local1, local2   integer
begin
  common =0.
  parbegin
  ПР1 while true do
      begin
        local1 =1.
        while local1=1 do TS(local1, common),
        CS1. { критическая секция процесса ПР1 }
        common =0
        PR1. { ПР1 после критической секции }
      end
  and
  ПР2 while true do
      begin
        local2 =1.
        while local2=1 do TS( local2, common)
        CS2, { критическая секция процесса ПР2 }
        common =0
        PR2  { ПР2 после критической секции }
      end
  parend
end
```

Предположим, что первым хочет войти в свою критическую секцию процесс ПР1 В этом случае значение local1 сначала установится в единицу, а после цикла проверки с помощью команды TS(local1,common) — в нуль. При этом значение common станет равным единице. Процесс ПР1 войдет в свою критическую секцию. После выполнения критической секции переменная common примет значение, равное нулю, что даст возможность второму процессу ПР2 войти в свою критическую секцию.

Безусловно, мы предполагаем, что в компьютере реализована блокировка памяти, то есть операция c o m m o n := 0 неделима. Команда проверки и установки значительно упрощает решение проблемы критических секций. Главная черта этой команды — ее неделимость.

Основной недостаток использования команд типа проверки и установки состоит в следующем: находясь в цикле проверки переменной common, процессы впустую потребляют время центрального процессора и другие ресурсы. Действительно, если предположить, что произошло прерывание процесса ПР1 во время выполнения своей критической секции в соответствии с некоторой дисциплиной обслуживания, и начал выполняться процесс ПР2, то он войдет в цикл проверки, впустую тратя процессорное время. В этом случае, до тех пор пока диспетчер супервизора не поставит на выполнение процесс ПР1 и не даст ему закончиться, процесс ПР2 не сможет войти в свою критическую секцию.

В микропроцессорах архитектуры ia32, с которыми мы теперь сталкиваемся постоянно, работая на персональных компьютерах, имеются специальные команды BTC, BTS, BTR, которые как раз и являются вариантами реализации команды проверки и установки. Рассмотрим одну из них — BTS.

Команда BTS (Bit Test and Reset — проверка и установка бита) является двухадресной [20]. По этой команде процессор сохраняет значение бита из первого операнда со смещением, указанным вторым операндом, во флаге CF (Carry Flag — флаг переноса)[1] регистра флагов, а затем устанавливает указанный бит в 1. Значение индекса выбираемого бита может быть представлено постоянным непосредственным значением в команде BTS или значением в общем регистре. В этой команде используется только 8-разрядное непосредственное значение. Значение этого операнда берется по модулю 32, таким образом, смещение битов находится в диапазоне от О до 31. Это позволяет выбрать любой бит внутри регистра. Для битовых строк в памяти это поле непосредственного значения дает только смещение внутри слова или двойного слова.

С учетом изложенного можно привести фрагмент кода, в котором данная команда используется для решения проблемы взаимного исключения (листинг 7.6).

Листинг 7.6. Фрагмент программы с критической секцией на ассемблере

```
L:   BTC  M, 1
     JC   L

          .; критическая  секция

     AND M. 0B
```

Однако здесь следует заметить, что некоторые ассемблеры поддерживают значения битовых смещений больше 31, используя поле непосредственного значения

[1] Располагается в слове состояния программы.

в комбинации с полем смещения операнда в памяти. В этом случае младшие 3 или 5 битов (3 — для 16-разрядных операндов, 5 — для 32-разрядных операндов), определяющие смещение бита (второй операнд команды), сохраняются в поле непосредственного операнда, а старшие биты сдвигаются и комбинируются с полем смещения. Процессор же игнорирует ненулевые значения старших битов поля второго операнда [20]. При доступе к памяти процессор обращается к четырем байтам (для 32-разрядного операнда), начинающимся по полученному следующим образом адресу:

Effective Address + (4 x (BitOffset DIV 32))

Либо (для 16-разрядного операнда) процессор обращается к двум байтам, начинающимся по адресу:

Effective Address + (2 x (BitOffset DIV 16))

Такое обращение происходит, даже если необходим доступ только к одному байту. Поэтому избегайте ссылок к областям памяти, близким к «пустым» адресным пространствам. В частности, избегайте ссылок на распределенные в памяти регистры ввода-вывода. Вместо этого используйте команду MOV для загрузки и сохранения значений по таким адресам и регистровую форму команды BTS для работы с данными.

Несмотря на то, что и алгоритм Деккера, основанный только на блокировке памяти, и операция проверки и установки пригодны для реализации взаимного исключения, оба эти приема очень неэффективны. Всякий раз, когда один из процессов выполняет свою критическую секцию, любой другой процесс, который пытается войти в свою критическую секцию, попадает в цикл проверки соответствующих переменных-флагов, регламентирующих доступ к критическим переменным. При таком неопределенном пребывании в цикле, которое называется *активным ожиданием,* напрасно расходуется процессорное время, поскольку процесс имеет доступ к тем общим переменным, которые и определяют возможность или невозможность входа в критическую секцию. При этом процесс отнимает ценное время центрального процессора, на самом деле ничего реально не выполняя. Как результат мы получаем общее замедление работы вычислительной системы.

До тех пор пока процесс, занимающий в данный момент критический ресурс, не решит его уступить, все другие процессы, ожидающие этого ресурса, могли бы вообще не конкурировать за процессорное время. Для этого их нужно перевести в состояние ожидания (заблокировать). Когда вход в критическую секцию снова освободится, можно будет опять перевести заблокированный процесс в состояние готовности к выполнению и дать ему возможность получить процессорное время. Самый простой способ предоставить процессорное время только одному вычислительному процессу — отключить систему прерываний, поскольку тогда никакое внешнее событие не сможет прервать выполняющийся процесс. Однако это, как мы уже знаем, приведет к тому, что система не сможет реагировать на внешние события.

Вместо того чтобы связывать с каждым процессом собственную переменную, как это было в рассмотренных нами решениях, можно со всем множеством конкури-

рующих критических секций связать одну переменную, которую Дейкстра предложил рассматривать как некоторый «ключ». Вначале доступ к критической секции открыт. Однако перед входом в свою критическую секцию процесс забирает ключ и тем самым блокирует другие процессы. Покидая критическую секцию, процесс открывает доступ, возвращая ключ на место. Если процесс, который хочет войти в свою критическую секцию, обнаруживает отсутствие ключа, он должен быть переведен в состояние блокирования до тех пор, пока процесс, имеющий ключ, не вернет его. Таким образом, каждый процесс, входящий в критическую секцию, должен вначале проверить, доступен ли ключ, и если это так, то сделать его недоступным для других процессов. Причем самым главным должно быть то, что эти два действия должны быть неделимыми, чтобы два или более процессов не могли одновременно получить доступ к ключу. Более того, проверку возможности входа в критическую секцию лучше всего выполнять не самим конкурирующим процессам, так как это приводит к активному ожиданию, а возложить эту функцию на операционную систему. Таким образом, мы подошли к одному из самых главных механизмов решения проблемы взаимного исключения — семафорам Дейкстры.

Семафорные примитивы Дейкстры

Понятие семафорных механизмов было введено Дейкстрой [10]. *Семафор* (semaphore) — это переменная специального типа, которая доступна параллельным процессам только для двух операций — закрытия и открытия, названных соответственно операциями P и V[1]. Эти операции являются примитивами относительно семафора, который указывается в качестве параметра операций. Здесь семафор играет роль вспомогательного критического ресурса, так как операции P и V неделимы при своем выполнении и взаимно исключают друг друга.

Семафорный механизм работает по схеме, в которой сначала исследуется состояние критического ресурса, идентифицируемое значением семафора, а затем уже осуществляется допуск к критическому ресурсу или отказ от него на некоторое время. При отказе доступа к критическому ресурсу используется режим *пассивного ожидания.* Поэтому в состав механизма включаются средства формирования и обслуживания очереди ожидающих процессов. Эти средства реализуются супервизором операционной системы. Необходимо отметить, что в силу взаимного исключения примитивов попытка в различных параллельных процессах одновременно выполнить примитив над одним и тем же семафором приведет к тому, что она окажется успешной только для одного процесса. Все остальные процессы на время выполнения примитива будут взаимно исключены.

Основным достоинством семафорных операций является отсутствие состояния активного ожидания, что может существенно повысить эффективность работы мультизадачной системы.

В настоящее время на практике используется много различных видов семафорных механизмов [41]. Варьируемыми параметрами, которые отличают различные виды примитивов, являются начальное значение и диапазон изменения значений

[1] P — от голландского Proberen (проверить), V — от голландского Verhogen (увеличить).

семафора, логика действий семафорных операций, количество семафоров, доступных для обработки при исполнении отдельного примитива.

Обобщенный смысл примитива P(S) состоит в проверке текущего значения семафора S. Если оно не меньше нуля, то осуществляется переход к следующей за примитивом операции. В противном случае процесс снимается на некоторое время с выполнения и переводится в состояние пассивного ожидания. Находясь в списке заблокированных, ожидающий процесс не проверяет семафор непрерывно, как в случае активного ожидания. Вместо него процессор может исполнять другой процесс, реально выполняющий какую-то полезную работу.

Операция V(S) связана с увеличением значения семафора на единицу и переводом одного или нескольких процессов в состояние готовности к исполнению центральным процессором.

Отметим еще раз, что операции P и V выполняются операционной системой в ответ на запрос, выданный некоторым процессом и содержащий имя семафора в качестве параметра.

Рассмотрим первый вариант алгоритма работы семафорных операций (листинг 7.7). Допустимыми значениями семафоров являются только целые числа. Двоичным семафором будем называть семафор, максимально возможное значение которого равно единице. Двоичный семафор[1] либо открыт, либо закрыт. В случае, когда семафор может принимать более двух значений, его называют N-ичным. Есть реализации, в которых семафорные переменные не могут быть отрицательными, а есть и такие, где отрицательное значение указывает на длину очереди процессов, стоящих в состоянии ожидания открытия семафора.

Листинг 7.7. Вариант реализации семафорных примитивов

```
P(S)  S =S-1.
    if S<0 then { остановить процесс и поместить его в очередь ожидания к семафору S }.

V(s)  if S<0 then { поместить один из ожидающих процессов очереди семафора S в очередь
готовности }.
      S =S+1.
```

Заметим, что для работы с семафорными переменными необходимо еще иметь операцию инициализации самого семафора, то есть задания ему начального значения. Обычно эту операцию называют InitSem; как правило, она имеет два параметра — имя семафорной переменной и ее начальное значение, то есть обращение имеет вид

```
InitSem ( Имя_семафора. Начальное_значение_семафора ).
```

Попытаемся на нашем примере двух конкурирующих процессов ПР1 и ПР2 проанализировать использование данных семафорных примитивов для решения проблемы критической секции. Программа, иллюстрирующая это решение, представлена в листинге 7.8.

Листинг 7.8. Взаимное исключение с помощью семафорных операций

```
var S semafor
begin                InitSem(S,              1).          продолжение &
```

[1] Двоичные семафоры иногда называют мьютексами (см далее)

Листинг 7.8 (продолжение)

```
parbegin
  ПР1  while true do
       begin
         P(S),
         CS1.    { критическая секция процесса ПР1 }
         V(S)
         PR1
       end
and
  ПР2  while true do
       begin
         P(S).
         CS2.    { критическая секция процесса ПР2 }
         V(S)
         PR1
       end
parend
end
```

Семафор S имеет начальное значение, равное 1. Если процессы ПР1 и ПР2 попытаются одновременно выполнить примитив P(S), то это удастся успешно сделать только одному из них. Предположим, это сделал процесс ПР2, тогда он закрывает семафор S, после чего выполняется его критическая секция. Процесс ПР1 в рассматриваемой ситуации будет заблокирован на семафоре S. Тем самым гарантируется взаимное исключение.

После выполнения примитива V(S) процессом ПР2 семафор S открывается, указывая на возможность захвата каким-либо процессом освободившегося критического ресурса. При этом производится перевод процесса ПР1 из заблокированного состояния в состояние готовности.

На уровне реализации возможно одно из двух решений в отношении процессов, которые переводятся из очереди ожидания в очередь готовности при выполнении примитива V:

* процесс при его активизации (выборка из очереди готовности) вновь пытается выполнить примитив P, считая предыдущую попытку неуспешной;
* процесс при помещении его в очередь готовности отмечается как успешно выполнивший примитив P, тогда при его активизации управление будет передано не на повторное выполнение примитива P, а на команду, следующую за ним.

Рассмотрим первый способ реализации. Пусть процесс ПР2 в некоторый момент времени выполняет операцию P(S). Тогда семафор S становится равным нулю. Пусть далее процесс ПР1 пытается выполнить операцию P(S). Процесс ПР1 в этом случае блокируется на семафоре S, так как значение семафора S равнялось нулю, а теперь станет равным -1. После выполнения критической секции процесс ПР2 выполняет операцию V(S), при этом значение семафора S становится равным нулю, а процесс ПР1 переводится в очередь готовности. Пусть через некоторое время процесс ПР1 будет активизирован, то есть выведен из состояния ожидания, и сможет продолжить свое исполнение. Он повторно попытается выполнить операцию P(S), однако это ему не удастся, так как S=0. Процесс ПР1 блокируется на семафоре, а его значение становится равным -1. Если через некоторое время процесс ПР2 попытается выполнить P(S), то он тоже заблокируется. Таким образом, возникнет

так называемая *тупиковая ситуация,* так как разблокировать процессы ПР1 и ПР2 некому.

При втором способе реализации тупика не будет. Действительно, пусть все происходит так же до момента окончания исполнения процессом ПР2 примитива V(S). Пусть примитив V(S) выполнен, и S=0. Через некоторое время процесс ПР1 активизируется. Согласно данному способу реализации он сразу же попадет в свою критическую секцию. При этом никакой другой процесс не попадет в свою критическую секцию, так как семафор остается закрытым. После исполнения своей критической секции процесс ПР1 выполнит V(S). Если за время выполнения критической секции процесса ПР1 процесс ПР2 не сделает попыток выполнить операцию P(S), семафор S установится в единицу. В противном случае значение семафора будет не больше нуля. Но в любом варианте после завершения операции V(S) процессом ПР1 доступ к критическому ресурсу со стороны процесса ПР2 будет разрешен.

Заметим, что возникновение тупиков возможно в случае несогласованного выбора механизма извлечения процессов из очереди, с одной стороны, и выбора алгоритмов семафорных операций, с другой.

Возможен другой алгоритм работы семафорных операций:

```
P(S)    if S>=1 then S =S-1
        else WAIT(S){ остановить процесс и поместить в очередь ожидания к семафору S }
V(S)    if S<0 then RELEASE(S){ поместить один из ожидающих процессов очереди семафора S
в очередь готовности }
        S =S+1
```

Здесь вызов WAIT(S) означает, что супервизор ОС должен перевести задачу в состояние ожидания, причем очередь процессов связана с семафором S. Вызов RELEASE(S) означает обращение к диспетчеру задач с просьбой перевести первый из процессов, стоящих в очереди S, в состояние готовности к исполнению.

Использование семафорных операций, выполненных подобным образом, позволяет решать проблему критических секций на основе первого способа реализации, причем без опасности возникновения тупиков. Действительно, пусть ПР2 в некоторый момент времени выполнит операцию P(S). Тогда семафор S становится равным нулю. Пусть далее процесс ПР1 пытается выполнить операцию P(S). Процесс ПР1 в этом случае блокируется на семафоре S, так как S=0, причем значение S не изменится. После выполнения своей критической секции процесс ПР2 выполнит операцию V(S), при этом значение семафора S станет равным единице, а процесс ПР1 переведется в очередь готовности. Если через некоторое время процесс ПР1 продолжит свое исполнение, он успешно выполнит примитив P(S) и войдет в свою критическую секцию.

В однопроцессорной вычислительной системе неделимость операций P и V можно обеспечить с помощью простого запрета прерываний. Сам же семафор S можно реализовать в виде записи с двумя полями (листинг 7.9.). В одном поле будет храниться целое значение S, во втором — указатель на список процессов, заблокированных на семафоре S.

Листинг 7.9. Реализация операций P и V для однопроцессорной системы

```
type Semaphore = record
        счетчик     integer.
        указатель pointer
```

продолжение &

Листинг 7.9 (продолжение)

```
        end
var S   Semaphore

procedure P ( var S   Semaphore)
begin ЗАПРЕТИТЬ_ПРЕРЫВАНИЯ
  S счетчик = S счетчик 1
  if S счетчик < O then
     WAIT(S)  { вставить обратившийся процесс в список по S указатель и передать
                на процессор готовый к выполнению процесс }
  РАЗРЕШИТЬ_ПРЕРЫВАНИЯ
end

procedure V ( var S   Semaphore)
begin  ЗАПРЕТИТЬ ПРЕРЫВАНИЯ
  S счетчик = S счетчик+1
  if S счетчик <= 0 then
    RELEASE (S)     { деблокировать первый процесс из списка по S указатель }
  РАЗРЕШИТЬ_ПРЕРЫВАНИЯ
end

procedure InitSem (var S   Semaphore)
begin
  S счетчик =1
  S указатель =nil
end
```

Реализация семафоров в мультипроцессорных системах сложнее, чем в однопроцессорных Одновременный доступ к семафору S двух процессов, выполняющихся на однопроцессорной вычислительной системе, предотвращается запретом прерывании Однако этот механизм не подходи г для мультипроцессорных систем, так как он не препятствует двум или более процессам одновременно обращаться к одному семафору В силу того что такой доступ должен реализовываться через критическую секцию, необходимо дополнительное аппаратное взаимное исключение доступа для различных процессоров Одним из решений является использование уже знакомых нам неделимых команд проверки и установки (TS) Двухкомпонентный семафор в этом случае расширяется включением третьего компонента — логического признака взаимоискл (листинг 7 10)

Листинг 7.10. Реализация операций P и V для мультипроцессорной системы

```
type Semaphore = record
           счетчик     integer
           указатель   pointer
           взаимоискл     boolean
         end
var S    Semaphore

procedure InitSem (var S    Semaphore)
begin
With S do
  begin
    счетчик  =1
    указатель   =nil
    взаимоискл=true
  end
end
```

```
procedure P ( var S   Semaphore),
var разрешено    boolean,
begin
  ЗАПРЕТИТЬ_ПРЕРЫВАНИЯ.
  repeat  TS(разрешено.  S взаимоискл) until  разрешено.
  S счетчик =S счетчик-1.
  if S счетчик < 0 then WAIT(S). { вставить обратившийся процесс в список по S указатель
                                   и передать на процессор готовый к выполнению процесс }
  S взаимоискл =true.
  РАЗРЕШИТЬ_ПРЕРЫВАНИЯ
end.

procedure V ( var S   Semaphore ).
var разрешено    boolean.
begin
  ЗАПРЕТИТЬ_ПРЕРЫВАНИЯ,
  repeat TS(разрешено. S взаимоискл) until разрешено
  S счетчик =S счетчик+1,
  if S счетчик <= 0 then RELEASE(S),    { деблокировать первый процесс из списка
                                          по S указатель }
  S взаимоискл =true,
  РАЗРЕШИТЬ_ПРЕРЫВАНИЯ.
end
```

Обратите внимание, что в данном тексте команда проверки и установки — TS(разрешено,S.взаимоискл) — работает не с целочисленными, а с булевыми значениями. Практически это ничего не меняет, ибо текст программы и ее машинная (двоичная) реализация — это разные вещи.

Мьютексы

Одним из вариантов реализации семафорных механизмов для организации взаимного исключения является так называемый *мьютекс* (mutex). Термин «mutex» произошел от словосочетания «mutual exclusion semaphore», что дословно переводится с английского как «семафор взаимного исключения». Мьютексы реализованы во многих операционных системах, их основное назначение — организация взаимного исключения для задач (потоков выполнения) одного или нескольких процессов. Мьютексы — это простейшие двоичные семафоры, которые могут находиться в одном из двух состояний — отмеченном и неотмеченном (открыт и закрыт соответственно). Когда какая-либо задача, принадлежащая любому процессу, становится владельцем объекта мьютекс, последний переводится в неотмеченное состояние. Если задача освобождает мьютекс, его состояние становится отмеченным.

Организация последовательного (а не параллельного) доступа к ресурсам с использованием мьютексов становится несложной, поскольку в каждый конкретный момент только одна задача может владеть этим объектом. Для того чтобы мьютекс стал доступен задачам (потокам выполнения), принадлежащим разным процессам, при создании ему необходимо присвоить имя, впоследствии передаваемое «по наследству» задачам, которые должны его использовать для взаимодействия. Для этого вводятся специальные системные вызовы (CreateMutex), в которых указываются начальное значение мьютекса, его имя и, возможно, атри-

буты защиты. Если начальное значение мьютекса равно true, считается, что задача, создающая этот объект, сразу будет им владеть. Можно указать в качестве начального значение false — в этом случае мьютекс не будет принадлежать ни одной из задач, и только специальным обращением к нему удастся изменить его состояние.

Для работы с мьютексом имеется несколько функций. Помимо уже упомянутой функции создания такого объекта (CreateMutex), есть функции открытия (OpenMutex), ожидания событий (WaitForSingleObject и WaitForMultipleObjects) и, наконец, освобождения этого объекта (ReleaseMutex).

Конкретные обращения к этим функциям и перечни передаваемых и получаемых параметров имеются в документации на соответствующую операционную систему.

Использование семафоров при проектировании взаимодействующих вычислительных процессов

Семафорные примитивы чрезвычайно широко используются при проектировании разнообразных вычислительных процессов. При этом некоторые задачи являются настолько «типичными», что их детальное рассмотрение уже стало классическим в соответствующих учебных пособиях. Не будем делать исключений и мы.

Задача «поставщик-потребитель»

Решение задачи «поставщик-потребитель» является характерным примером использования семафорных операций. Содержательная постановка этой задачи уже была нами описана в начале этой главы. Разделяемыми переменными здесь являются счетчики свободных и занятых буферов, которые должны быть защищены со стороны обоих процессов, то есть действия по посылке и получению сообщений должны быть синхронизированы.

Использование семафоров для решения данной задачи иллюстрирует листинг 7. 11.

Листинг 7. 11. Решение задачи «поставщик-потребитель»

```
var S_свободно S_заполнено S_взаимоискл semaphore
begin
  InitSem(S_Свободно. N),
  InitSem(S_Заполнено 0)
  InitSem(S_взаимоискл 1)
parbegin
  ПОСТАВЩИК while true do
          begin
                  { подготовить сообщение }
            P(S_свободно)
            P(S_взаимоискл),
                  { послать сообщение }
            V(S_заполнено)
            V(S_взаимоискл).
          end
and
```

```
    ПОТРЕБИТЕЛЬ  while true do
            begin
              P(S_заполнено).
              P(S_взаимоискл).
                    { получить сообщение }
              V(S_свободно).
              V(S_взаимоискл).
                    { обработать сообщение }
            end
  parend
  end
```

Здесь переменные S_свободно, S_заполнено являются числовыми семафорами, S_взаимоискл — двоичный семафор. Переменная S_свободно имеет начальное значение, равное N, где N — количество буферов, с помощью которых процессы сотрудничают. Предполагается, что в начальный момент количество свободных буферов равно N; соответственно, количество занятых равно нулю. Двоичный семафор S_взаимоискл гарантирует, что в каждый момент только один процесс сможет работать с критическим ресурсом, выполняя свою критическую секцию. Семафоры S_свободно и S_заполнено используются как счетчики свободных и заполненных буферов.

Действительно, перед посылкой сообщения поставщик уменьшает значение S_свободно на единицу в результате выполнения операции P(S_свободно), а после посылки сообщения увеличивает значение S_заполнено на единицу в результате выполнения операции V(S_заполнено). Аналогично, перед получением сообщения потребитель уменьшает значение S_заполнено в результате выполнения операции P(S_заполнено), а после получения сообщения увеличивает значение S_свободно в результате выполнения операции V(S_свободно). Семафоры S_заполнено, S_свободно могут также использоваться для блокировки соответствующих процессов. Если пул буферов оказывается пустым, и к нему первым обратится процесс «потребитель», он заблокируется на семафоре S_заполнено в результате выполнения операции P(S_заполнено). Если пул буферов заполнится и к нему обратится процесс «поставщик», то он будет заблокирован на семафоре S_свободно в результате выполнения операции P(S_свободно).

В решении задачи о поставщике и потребителе общие семафоры применены для учета свободных и заполненных буферов. Их можно также применить и для распределения иных ресурсов.

Синхронизация взаимодействующих процессов с помощью семафоров

Можно использовать семафорные операции для решения таких задач, в которых успешное завершение одного процесса связано с ожиданием завершения другого. Предположим, что существуют два процесса ПР1 и ПР2. Необходимо, чтобы процесс ПР1 запускал процесс ПР2 с ожиданием его выполнения, то есть ПР1 не будет продолжать свое выполнение до тех пор, пока процесс ПР2 до конца не выполнит свою работу. Программа, реализующая такое взаимодействие, представлена в листинге 7.12.

Листинг 7. 12. Пример синхронизации процессов

```
var S    Semaphore
begin
  InitSem(S, 0)

  ПР1  begin
       ПР11: { первая часть ПР1 }
       ON ( ПР2 ). { поставить на выполнение ПР2 }
       P(S).
       ПР12. { оставшаяся часть ПР1 }
       STOP
     end,


  ПР2:   begin
       ПР2. { вся работа программы ПР2 }
       V(S),
       STOP
     end
end
```

Начальное значение семафора *S* равно нулю. Если процесс ПР1 начал выполняться первым, то через некоторое время он поставит на выполнение процесс ПР2, после чего выполнит операцию P(S) и «заснет» на семафоре, перейдя в состояние пассивного ожидания. Процесс ПР2, осуществив все необходимые действия, выполнит примитив V(S) и откроет семафор, после чего процесс ПР1 будет готов к дальнейшему выполнению.

Задача «читатели-писатели»

Другой важной и часто встречающейся задачей, решение которой также требует синхронизации, является задача «читатели-писатели». Эта задача имеет много вариантов. Наиболее характерная область ее использования — построение систем управления файлами и базами данных, информационно-справочных систем. Два класса процессов имеют доступ к некоторому ресурсу (области памяти, файлам). «Читатели» — это процессы, которые могут параллельно считывать информацию из некоторой общей области памяти, являющейся критическим ресурсом. «Писатели» — это процессы, записывающие информацию в эту область памяти, исключая друг друга, а также процессы «читатели». Имеются различные варианты взаимодействия между писателями и читателями. Наиболее широко распространены следующие условия.

Устанавливается приоритет в использование критического ресурса процессам «читатели». Это означает, что если хотя бы один читатель пользуется ресурсом, то он закрыт для всех писателей и доступен для всех читателей. Во втором варианте, наоборот, больший приоритет у процессов «писатели». При появлении запроса от писателя необходимо закрыть дальнейший доступ всем тем читателям, которые запросят критический ресурс после него.

Помимо системы управления файлами другим типичным примером решения задачи «читатели-писатели» может служить система автоматизированной продажи билетов. Процессы «читатели» обеспечивают нас справочной информацией о наличии свободных билетов на тот или иной рейс. Процессы «писатели» запускают-

ся с пульта кассира, когда он оформляет для нас тот или иной билет. Имеется большое количество как читателей, так и писателей.

Пример программы, реализующей решение данной задачи в первой постановке, представлен в листинге 7. 13. Процессы «читатели» и «писатели» описаны в виде соответствующих процедур.

Листинг 7. 13. Решение задачи «читатели-писатели» с приоритетом в доступе к критическому ресурсу читателей

```
var R, W    semaphore.
      NR   integer,
procedure ЧИТАТЕЛЬ,
begin
  P(R).
  Inc(NR)       { NR =NR +1 }
  if NR = 1 then P(W)
  V(R)
  Read_Data. { критическая секция }
  P(R).
  Dec(NR).
  if N_R = 0 then V(W),
  V(R)
end

procedure ПИСАТЕЛЬ,
begin
  P(W),
  Write_Data   { критическая секция }
  V(W)
end

begin
  NR =0
  InitSem(S. l)  InitSem(W  1)
  parbegin
    while true do ЧИТАТЕЛЬ
  and
    while true do ЧИТАТЕЛЬ
  and


    while true do ЧИТАТЕЛЬ
  and
    while true do ПИСАТЕЛЬ
  and
    while true do ПИСАТЕЛЬ
  and


    while true do ПИСАТЕЛЬ
  parend
end
```

При решении данной задачи используются два семафора R и W, а также переменная NR, предназначенная для подсчета текущего числа процессов типа «читатели», находящихся в критической секции. Доступ к разделяемой области памяти осу-

ществляется через семафор W. Семафор R требуется для взаимного исключения процессов типа «читатели».

Если критический ресурс не используется, то первый появившийся процесс при входе в критическую секцию выполнит операцию P(W) и закроет семафор. Если процесс является читателем, то переменная NR увеличится на единицу, и последующие читатели будут обращаться к ресурсу, не проверяя значения семафора W, что обеспечит параллельность их доступа к памяти. Последний читатель, покидающий критическую секцию, является единственным, кто выполнит операцию V(W) и откроет семафор W. Семафор R предохраняет от некорректного изменения значения NR, а также от выполнения читателями операций P(W) и V(W). Если в критической секции находится писатель, то на семафоре W может быть заблокирован только один читатель, все остальные будут блокироваться на семафоре R. Другие писатели блокируются на семафоре W.

Когда писатель выполняет операцию V(W), неясно, какого типа процесс войдет в критическую секцию. Чтобы гарантировать получение читателями наиболее свежей информации, необходимо при постановке в очередь готовности использовать дисциплину обслуживания, учитывающую более высокий приоритет писателей. Однако этого оказывается недостаточно, ибо если в критической секции продолжает находиться по крайней мере один читатель, то он не даст обновить данные, но и не воспрепятствует вновь приходящим процессам «читателям» войти в свою критическую секцию. Необходим дополнительный семафор. Пример правильного решения этой задачи приведен в листинге 7.14.

Листинг 7.14. Решение задачи «читатели-писатели» с приоритетом в доступе к критическому ресурсу писателей

```
var S, W, R   semaphore.
    NR : integer,
procedure ЧИТАТЕЛЬ:
begin
  P(S): P(R):
  Inc(NR),
  if NR= 1 then P(W):
  V(S). V(R),
  Read_Data. { критическая секция }
  P(R):
  Dec(NR).
  if NR = 0 then V(W).
  V(R)
end;

procedure ПИСАТЕЛЬ;
begin
  P(S). P(W):
  Write_Data. { критическая секция }
  V(S); V(W)
end.

begin
  NR=0;
  InitSem(S. 1). InitSem(W. 1); InitSem(R. 1),
  parbegin
```

```
    while true do ЧИТАТЕЛЬ
  and
    while true do ЧИТАТЕЛЬ
  and


    while true do ЧИТАТЕЛЬ
  and
    while true do ПИСАТЕЛЬ
  and
    while true do ПИСАТЕЛЬ
  and


    while true do ПИСАТЕЛЬ
  parend
end
```

Как можно заметить, семафор S блокирует приход новых читателей, если появился хотя бы один писатель. Обратите внимание, что в процедуре ЧИТАТЕЛЬ использование семафора S имеет место только при входе в критическую секцию. После выполнения чтения уже категорически нельзя использовать этот семафор, ибо он тут же заблокирует первого же читателя, если хотя бы один писатель захочет войти в свою критическую секцию. И получится так называемая тупиковая ситуация, ибо писатель не сможет войти в критическую секцию, поскольку в ней уже находится читатель. А читатель не сможет покинуть критическую секцию, потому что писатель желает войти в свою критическую секцию.

Обычно программы, решающие проблему «читатели-писатели», используют как семафоры, так и мониторные схемы с взаимным исключением, то есть такие, которые блокируют доступ к критическим ресурсам для всех остальных процессов, если один из них модифицирует значения общих переменных. Взаимное исключение требует, чтобы писатель ждал завершения всех текущих операций чтения. При условии, что писатель имеет более высокий приоритет, чем читатель, такое ожидание в ряде случаев весьма нежелательно. Кроме того, реализация принципа взаимного исключения в многопроцессорных системах может вызвать определенную избыточность. Поэтому схема, представленная в листинге 7.15 и применяемая иногда для решения задачи «читатели-писатели», в случае одного писателя допускает одновременное выполнение операций чтения и записи. После чтения данных процесс «читатель» проверяет, мог ли он получить неправильное значение, некорректные данные (вследствие того, что параллельно с ним процесс «писатель» мог их изменить), и если обнаруживает, что это именно так, то операция чтения повторяется.

Листинг 7. 15. Синхронизация процессов «читатели» и «писатель» без взаимного исключения

```
Var V1 V2   integer.

Procedure ПИСАТЕЛЬ,
Begin
  Inc(V1).
  Write_Data.
  V2 =V1
End.
```

продолжение &

Листинг 7.15 *(продолжение)*

```
Procedure ЧИТАТЕЛЬ
Var V  integer
Begin
  Repeat    V - V2
       Read_Data
  Until V1 = V
End

Begin
  V1  = 0
  V2  = 0
  Parbegin
    while true do ЧИТАТЕЛЬ
  and
    while true do ЧИТАТЕЛЬ
  and


    while true do ЧИТАТЕЛЬ
  and
    while true do ПИСАТЕЛЬ
  parend
end
```

Этот алгоритм использует для данных два номера версий, которым соответствуют переменные V1 и V2. Перед записью порции новых данных процесс «писатель» увеличивает на 1 значение переменной V1, а после записи — переменной V2. Читатель обращается к V2 перед чтением данных, а к V1 — после. Если при этом переменные V1 и V2 равны, то очевидно, что получена правильная версия данных Если же данные обновлялись за время чтения, то операция повторяется. Этот алгоритм может быть использован в случае, если нежелательно заставлять процесс «писатель» ждать, пока читатели закончат операцию чтения, или если вероятность повторения операции чтения достаточно мала и обусловленное повторными операциями снижение эффективности системы меньше потерь, связанных с избыточностью решения с помощью взаимного исключения Однако необходимо иметь в виду ненулевую вероятность зацикливания чтения при высокой интенсивности операций записи. Наконец, если само чтение представляет собой достаточно длительную операцию, то оператор V:= V2 для процесса «читатель» может быть заменен следующим оператором:

```
Repeat V  = V2 Until V1 = V
```

Это предотвратит выполнение читателем операции чтения, если писатель уже начал запись.

Мониторы Хоара

Анализ рассмотренных задач показывает, что, несмотря на очевидные достоинства (простота, независимость от количества процессов, отсутствие активного ожидания), семафорные механизмы имеют и ряд недостатков Эти механизмы являются слишком примитивными, так как семафор не указывает непосредственно на синх-

ронизирующее условие, с которым он связан, или на критический ресурс. Поэтому при построении сложных схем синхронизации алгоритмы решения задач порой получаются весьма непростыми, ненаглядными и трудными для доказательства их правильности.

Необходимо иметь очевидные решения, которые позволят прикладным программистам без лишних усилий, связанных с доказательством правильности алгоритмов и отслеживанием большого числа взаимосвязанных объектов, создавать параллельные взаимодействующие программы. К таким решениям можно отнести так называемые мониторы, предложенные Хоаром [52].

В параллельном программировании *монитор* — это пассивный набор разделяемых переменных и повторно входимых процедур доступа к ним, которым процессы пользуются в режиме разделения, причем в каждый момент им может пользоваться только один процесс.

Рассмотрим, например, некоторый ресурс, который разделяется между процессами каким-либо планировщиком [17]. Каждый раз, когда процесс желает получить в свое распоряжение какие-то ресурсы, он должен обратиться к программе-планировщику. Этот планировщик должен иметь переменные, с помощью которых можно отслеживать, занят ресурс или свободен. Процедуру планировщика разделяют все процессы, и каждый процесс может в любой момент захотеть обратиться к планировщику. Но планировщик не в состоянии обслуживать более одного процесса одновременно. Такая процедура-планировщик и представляет собой пример монитора.

Таким образом, монитор — это механизм организации параллелизма, который содержит как данные, так и процедуры, необходимые для динамического распределения конкретного общего ресурса или группы общих ресурсов. Процесс, желающий получить доступ к разделяемым переменным, должен обратиться к монитору, который либо предоставит доступ, либо откажет в нем. Необходимость входа в монитор с обращением к какой-либо его процедуре (например, с запросом на выделение требуемого ресурса) может возникать у многих процессов. Однако вход в монитор находится под жестким контролем — здесь осуществляется взаимное исключение процессов, так что в каждый момент времени только одному процессу разрешается войти в монитор. Процессам, которые хотят войти в монитор, когда он уже занят, приходится ждать, причем режимом ожидания автоматически управляет сам монитор. При отказе в доступе монитор блокирует обратившийся к нему процесс и определяет условие ожидания. Проверка условия выполняется самим монитором, который и деблокирует ожидающий процесс. Поскольку механизм монитора гарантирует взаимное исключение процессов, исключаются серьезные проблемы, связанные с организацией параллельных взаимодействующих процессов.

Внутренние данные монитора могут быть либо глобальными (относящимися ко всем процедурам монитора), либо локальными (относящимися только к одной конкретной процедуре). Ко всем этим данным можно обращаться только изнутри монитора; процессы, находящиеся вне монитора и, по существу, только вызывающие его процедуры, просто не могут получить доступ к данным монитора. При

первом обращении монитор присваивает своим переменным начальные значения. При каждом последующем обращении используются те значения переменных, которые остались от предыдущего обращения.

Если процесс обращается к некоторой процедуре монитора, а соответствующий ресурс уже занят, эта процедура выдает команду ожидания WAIT с указанием условия ожидания. Процесс мог бы оставаться внутри монитора, однако, если в монитор затем войдет другой процесс, это будет противоречить принципу взаимного исключения. Поэтому процесс, переводящийся в режим ожидания, должен вне монитора ждать того момента, когда необходимый ему ресурс освободится.

Со временем процесс, который занимал данный ресурс, обратится к монитору, чтобы возвратить ресурс системе. Соответствующая процедура монитора при этом может просто принять уведомление о возвращении ресурса, а затем ждать, пока не поступит запрос от другого процесса, которому потребуется этот ресурс. Однако может оказаться, что уже имеются процессы, ожидающие освобождения данного ресурса. В этом случае монитор выполняет команду извещения (сигнализации) SIGNAL, чтобы один из ожидающих процессов мог получить данный ресурс и покинуть монитор. Если процесс сигнализирует о возвращении (иногда называемом освобождением) ресурса и в это время нет процессов, ожидающих данного ресурса, то подобное оповещение не вызывает никаких других последствий, кроме того, что монитор, естественно, вновь внесет ресурс в список свободных. Очевидно, что процесс, ожидающий освобождения некоторого ресурса, должен находиться вне монитора, чтобы другой процесс имел возможность войти в монитор и возвратить ему этот ресурс.

Чтобы гарантировать, что процесс, находящийся в ожидании некоторого ресурса, со временем получит этот ресурс, считается, что ожидающий процесс имеет более высокий приоритет, чем новый процесс, пытающийся войти в монитор. В противном случае новый процесс мог бы перехватить ожидаемый ресурс до того, как ожидающий процесс вновь войдет в монитор. Если допустить многократное повторение подобной нежелательной ситуации, то ожидающий процесс мог бы откладываться бесконечно. Для систем реального времени можно допустить использование дисциплины обслуживания на основе абсолютных или динамически изменяемых приоритетов.

В качестве примера рассмотрим простейший монитор для выделения одного ресурса (листинг 7. 16).

Листинг 7. 16. Пример монитора Хоара

```
monitor Resourse,
condition free. { условие - свободный }
var busy    boolean. { занят }

procedure REQUEST: { запрос }
begin
    if busy then WAIT ( free ).
    busy =true.
    TakeOff: { выдать ресурс }
end.
```

```
procedure RELEASE;
begin
    TakeOn; { взять ресурс }
    busy:=false;
    SIGNAL ( free )
end:

begin
  busy:=false;
end
```

Единственный ресурс динамически запрашивается и освобождается процессами, которые обращаются к процедурам REQUEST (запрос) и RELEASE (освободить). Если процесс обращается к процедуре REQUEST в тот момент, когда ресурс используется, значение переменной busy (занято) будет равно true, и процедура REQUEST выполнит операцию монитора WAIT(free). Эта операция блокирует не процедуру REQUEST, а обратившийся к ней процесс, который помещается в конец очереди процессов, ожидающих, пока не будет выполнено условие free (свободно).

Когда процесс, использующий ресурс, обращается к процедуре RELEASE, операция монитора SIGNAL деблокирует процесс, находящийся в начале очереди, не позволяя исполняться никакой другой процедуре внутри того же монитора. Этот деблокированный процесс будет готов возобновить исполнение процедуры REQUEST сразу же после операции WAIT(free), которая его и блокировала. Если операция SIGNAL(free) выполняется в то время, когда нет процесса, ожидающего условия free, то никаких действий не выполняется.

Использование монитора в качестве основного средства синхронизации и связи освобождает процессы от необходимости явно разделять между собой информацию. Напротив, доступ к разделяемым переменным всегда ограничен телом монитора, и, поскольку мониторы входят в состав ядра операционной системы, разделяемые переменные становятся системными переменными. Это автоматически исключает необходимость в критических секциях (так как в каждый момент монитором может пользоваться только один процесс, то два процесса никогда не смогут получить доступ к разделяемым переменным одновременно).

Монитор является пассивным объектом в том смысле, что это не процесс; его процедуры выполняются только по требованию процесса.

Хотя по сравнению с семафорами мониторы не представляют собой существенно более мощного инструмента для организации параллельных взаимодействующих вычислительных процессов, у них есть некоторые преимущества перед более примитивными синхронизирующими средствами. Во-первых, мониторы очень гибки. В форме мониторов можно реализовать не только семафоры, но и многие другие синхронизирующие операции. Например, разобранный в разделе «Средства синхронизации и связи взаимодействующих вычислительных процессов» механизм решения задачи «поставщик-потребитель» легко запрограммировать в виде монитора. Во-вторых, локализация всех разделяемых переменных внутри тела монитора позволяет избавиться от малопонятных конструкций в синхронизируемых процессах — сложные взаимодействия процессов можно синхронизировать наглядным образом. В-третьих, мониторы дают процессам возможность совместно ис-

пользовать программные модули, представляющие собой критические секции. Если несколько процессов совместно используют ресурс и работают с ним совершенно одинаково, то в мониторе достаточно только одной процедуры, тогда как решение с семафорами требует, чтобы в каждом процессе имелся собственный экземпляр критической секции. Таким образом, мониторы по сравнению с семафорами позволяют значительно упростить организацию взаимодействующих вычислительных процессов и дают большую наглядность при совсем незначительной потере в эффективности.

Почтовые ящики

Тесное взаимодействие между процессами предполагает не только синхронизацию — обмен временными сигналами, но также передачу и получение произвольных данных, то есть обмен сообщениями. В системе с одним процессором посылающий и получающий процессы не могут работать одновременно. В мультипроцессорных системах также нет никакой гарантии их одновременного исполнения. Следовательно, для хранения посланного, но еще не полученного сообщения необходимо место. Оно называется *буфером сообщений,* или *почтовым ящиком*[1].

Если процесс P1 хочет общаться с процессом P2, то P1 просит систему предоставить или образовать почтовый ящик, который свяжет эти два процесса так, чтобы они могли передавать друг другу сообщения. Для того чтобы послать процессу P2 какое-то сообщение, процесс P1 просто помещает это сообщение в почтовый ящик, откуда процесс P2 может его в любое время получить. При применении почтового ящика процесс P2 в конце концов обязательно получит сообщение, когда обратится за ним (если вообще обратится). Естественно, что процесс P2 должен знать о существовании почтового ящика. Поскольку в системе может быть много почтовых ящиков, необходимо обеспечить доступ процессу к конкретному почтовому ящику. Почтовые ящики являются системными объектами, и для пользования таким объектом необходимо получить его у операционной системы, что осуществляется с помощью соответствующих запросов.

Если объем передаваемых данных велик, то эффективнее не передавать их непосредственно, а отправлять в почтовый ящик сообщение, информирующее процесс-получатель о том, где можно их найти.

Почтовый ящик может быть связан с парой процессов, только с отправителем, только с получателем, или его можно получить из множества почтовых ящиков, которые используют все или несколько процессов. Почтовый ящик, связанный с процессом-получателем, облегчает посылку сообщений от нескольких процессов в фиксированный пункт назначения. Если почтовый ящик не связан жестко с процессами, то сообщение должно содержать идентификаторы и процесса-отправителя, и процесса-получателя.

[1] Название «почтовый ящик» происходит от обычного приспособления для отправки почты.

Итак, почтовый ящик — это информационная структура, поддерживаемая операционной системой. Она состоит из головного элемента, в котором находится информация о данном почтовом ящике, и нескольких буферов (гнезд), в которые помещают сообщения. Размер каждого буфера и их количество обычно задаются при образовании почтового ящика.

Правила работы почтового ящика могут быть различными в зависимости от его сложности [17]. В простейшем случае сообщения передаются только в одном направлении. Процесс P1 может посылать сообщения до тех пор, пока имеются свободные гнезда. Если все гнезда заполнены, то P1 может либо ждать, либо заняться другими делами и попытаться послать сообщение позже. Аналогично процесс P2 может получать сообщения до тех пор, пока имеются заполненные гнезда. Если сообщений нет, то он может либо ждать сообщений, либо продолжать свою работу. Эту простую схему работы почтового ящика можно усложнять в нескольких направлениях и получать более хитроумные системы общения — двунаправленные и многовходовые почтовые ящики.

Двунаправленный почтовый ящик, связанный с парой процессов, позволяет подтверждать прием сообщений. При наличии множества гнезд каждое из них хранит либо сообщение, либо подтверждение. Чтобы гарантировать передачу подтверждений, когда все гнезда заняты, подтверждение на сообщение помещается в то же гнездо, в котором находится сообщение, и это гнездо уже не используется для другого сообщения до тех пор, пока подтверждение не будет получено. Из-за того, что некоторые процессы не забрали свои сообщения, связь может быть приостановлена. Если каждое сообщение снабдить пометкой времени, то управляющая программа может периодически удалять старые сообщения.

Процессы могут быть также остановлены в связи с тем, что другие процессы не смогли послать им сообщения. Если время поступления каждого остановленного процесса в очередь заблокированных процессов регистрируется, то управляющая программа может периодически посылать им пустые сообщения, чтобы они не ждали чересчур долго.

Реализация почтовых ящиков требует использования примитивных операторов низкого уровня, таких как операции P и V или каких-либо других, но пользователям может дать средства более высокого уровня (наподобие мониторов Хоара), например, такие, как представлены ниже.

SEND_MESSAGE (Получатель, Сообщение, Буфер)

Эта операция переписывает сообщение в некоторый буфер, помещает его адрес в переменную Буфер и добавляет буфер к очереди Получатель. Процесс, выдавший операцию SEND_MESSAGE, продолжит свое исполнение.

WAIT_MESSAGE (Отправитель. Сообщение, Буфер)

Эта операция блокирует процесс, выдавший операцию, до тех пор, пока в его очереди не появится какое-либо сообщение. Когда процесс передается на процессор, он получает имя отправителя с помощью переменной Отправитель, текст сообщения через переменную Сообщение и адрес буфера в переменной Буфер. Затем буфер удаляется из очереди, и процесс может записать в него ответ отправителю.

SEND_ANSWER (Результат. Ответ. Буфер)

Эта операция записывает информацию, определяемую через переменную Ответ, в тот буфер, номер которого указывается переменной Буфер (из этого буфера было получено сообщение), и добавляет буфер к очереди отправителя. Если отправитель ждет ответ, он деблокируется.

WAIT_ANSWER (Результат, Ответ, Буфер)

Эта операция блокирует процесс, выдавший операцию, до тех пор, пока в буфер не поступит ответ; доступ к нему возможен через переменную Буфер. После того как ответ поступил и процесс передан на процессор, ответ, доступ к которому определяется через переменную Ответ, переписывается в память процессу, а буфер освобождается. Значение переменной Результат указывает, является ли ответ пустым, то есть выданным операционной системой, так как сообщение было адресовано несуществующему (или так и не ставшему активным) процессу.

Основные достоинства почтовых ящиков:

* процессу не нужно знать о существовании других процессов до тех пор, пока он не получит сообщения от них;
* два процесса могут обменяться более чем одним сообщением за один раз;
* операционная система может гарантировать, что никакой иной процесс не вмешается во взаимодействие процессов, ведущих между собой «переписку»;
* очереди буферов позволяют процессу-отправителю продолжать работу, не обращая внимания на получателя.

Основным недостатком буферизации сообщений является появление еще одного ресурса, которым нужно управлять. Этим ресурсом являются сами почтовые ящики.

К другому недостатку можно отнести статический характер этого ресурса: количество буферов для передачи сообщений через почтовый ящик фиксировано. Поэтому естественным стало появление механизмов, подобных почтовым ящикам, но реализованных на принципах динамического выделения памяти под передаваемые сообщения.

В операционных системах компании Microsoft тоже имеются почтовые ящики (mailslots). В частности, они достаточно часто используются при создании распределенных приложений для сети. При работе с ними в приложении, которое должно отправить сообщение другому приложению, необходимо указывать класс доставки сообщений. Различают два класса доставки. Первый класс (first-class delivery) гарантирует доставку сообщений; он ориентирован на сеансовое взаимодействие между процессами и позволяет организовать посылки типа «один к одному» и «один ко многим». Второй класс (second-class delivery) основан на механизме датаграмм, и он уже не гарантирует доставку сообщений получателю.

Конвейеры и очереди сообщений

Конвейеры

Программный *канал связи* (pipe), или, как его иногда называют, *конвейер, транспортер,* является средством, с помощью которого можно обмениваться данными

между процессами. Принцип работы конвейера основан на механизме ввода-вывода файлов в UNIX, то есть задача, передающая информацию, действует так, как будто она записывает данные в файл, в то время как задача, для которой предназначается эта информация, читает ее из этого файла. Операции записи и чтения осуществляются не записями, как это делается в обычных файлах, а потоком байтов, как это принято в UNIX-системах. Таким образом, функции, с помощью которых выполняется запись в канал и чтение из него, являются теми же самыми, что и при работе с файлами. По сути, канал представляет собой поток данных между двумя (или более) процессами. Это упрощает программирование и избавляет программистов от использования каких-то новых механизмов. На самом деле конвейеры не являются файлами на диске, а представляют собой буферную память, работающую по принципу FIFO, то есть по принципу обычной очереди. Однако не следует путать конвейеры с очередями сообщений; последние реализуются иначе и имеют другие возможности.

Конвейер имеет определенный размер[1], который не может превышать 64 Кбайт и работает циклически. Вспомните реализацию очереди на массивах, когда имеются указатели начала и конца очереди, которые перемещаются циклически по массиву. То есть имеется некий массив и два указателя: один показывает на первый элемент (указатель на начало — head), а второй — на последний (указатель на конец — tail).

В начальный момент оба указателя равны нулю. Добавление самого первого элемента в пустую очередь приводит к тому, что указатели на начало и на конец принимают значение, равное 1 (в массиве появляется первый элемент). В последующем добавление нового элемента вызывает изменение значения второго указателя, поскольку он отмечает расположение именно последнего элемента очереди. Чтение (и удаление) элемента (читается и удаляется всегда первый элемент из созданной очереди) приводит к необходимости модифицировать значение указателя на ее начало. В результате операций записи (добавления) и чтения (удаления) элементов в массиве, моделирующем очередь элементов, указатели будут перемещаться от начала массива к его концу. При достижении указателем значения индекса последнего элемента массива значение указателя вновь становится единичным (если при этом не произошло переполнение массива, то есть количество элементов в очереди не стало большим числа элементов в массиве). Можно сказать, что мы как бы замыкаем массив в кольцо, организуя круговое перемещение указателей на начало и на конец, которые отслеживают первый и последний элементы в очереди. Сказанное иллюстрирует рис. 7.4. Именно так функционирует конвейер.

Как информационная структура конвейер описывается идентификатором, размером и двумя указателями. Конвейеры представляют собой системный ресурс. Чтобы начать работу с конвейером, процесс сначала должен заказать его у операционной системы и получить в свое распоряжение. Процессы, знающие идентификатор конвейера, могут через него обмениваться данными.

[1] Механизм конвейеров, впервые введенный в UNIX-системах, имеет максимальный размер 64 Кбайт, поскольку в 16-разрядных мини-ЭВМ, для которых создавалась эта ОС, нельзя было иметь массив данных большего размера.

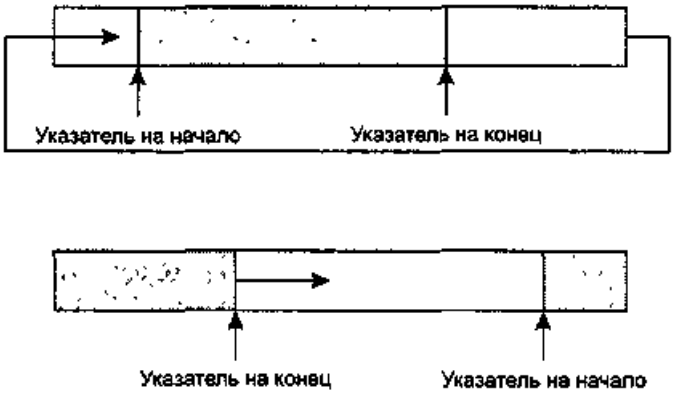

Рис. 7.4. Организация очереди в массиве

В качестве иллюстрации приведем основные системные запросы для работы с конвейерами, которые имеются в API OS/2.

* Функция создания конвейера:

  DosCreatePipe (&ReadHandle, &WriteHandle, PipeSize);

  Здесь ReadHandle — дескриптор чтения из конвейера, WriteHandle — дескриптор записи в конвейер, PipeSize — размер конвейера.

* Функция чтения из конвейера:

  DosRead (&ReadHandle, (PVOID)&Inform, sizeof(Inform). &BytesRead).

  Здесь ReadHandle — дескриптор чтения из конвейера, Inform — переменная любого типа, sizeof(Inform) — размер переменной Inform, BytesRead — количество прочитанных байтов. Данная функция при обращении к пустому конвейеру будет ожидать, пока в нем не появится информация для чтения.

* Функция записи в конвейер:

  DosWrite (&WriteHandle. (PVOID)&Inform. sizeof(Inform) &BytesWrite).

  Здесь WriteHandle — дескриптор записи в конвейер, BytesWrite — количество записанных байтов.

Читать из конвейера может только тот процесс, который знает идентификатор соответствующего конвейера. При работе с конвейером данные непосредственно помещаются в него. Еще раз отметим, что из-за ограничения на размер конвейера программисты сталкиваются и с ограничениями на размеры передаваемых через него сообщений.

Очереди сообщений

Очереди (queues) *сообщений* предлагают более удобный метод связи между взаимодействующими процессами по сравнению с каналами, но в своей реализации они сложнее. С помощью очередей также можно из одной или нескольких задач независимым образом посылать сообщения некоторой задаче-приемнику. При этом только процесс-приемник может читать и удалять сообщения из очереди, а про-

цессы-клиенты имеют право лишь помещать в очередь свои сообщения. Таким образом, очередь работает только в одном направлении. Если же необходима двухсторонняя связь, то можно создать две очереди.

Работа с очередями сообщений отличается от работы с конвейерами. Во-первых, очереди сообщений предоставляют возможность использовать несколько дисциплин обработки сообщений:

* FIFO — сообщение, записанное первым, будет первым и прочитано;
* LIFO — сообщение, записанное последним, будет прочитано первым;
* приоритетный доступ — сообщения читаются с учетом их приоритетов;
* произвольный доступ — сообщения читаются в произвольном порядке.

Тогда как канал обеспечивает только дисциплину FIFO.

Во-вторых, если при чтении сообщения оно удаляется из конвейера, то при чтении сообщения из очереди этого не происходит, и сообщение при желании может быть прочитано несколько раз.

В-третьих, в очередях присутствуют не непосредственно сами сообщения, а только их адреса в памяти и размер. Эта информация размещается системой в сегменте памяти, доступном для всех задач, общающихся с помощью данной очереди.

Каждый процесс, использующий очередь, должен предварительно получить разрешение на доступ в общий сегмент памяти с помощью системных запросов API, ибо очередь — это системный механизм, и для работы с ним требуются системные ресурсы и, соответственно, обращение к самой ОС. Во время чтения из очереди задача-приемник пользуется следующей информацией:

* идентификатор процесса (Process Identifier, PID), который передал сообщение;
* адрес и длина переданного сообщения;
* признак необходимости ждать, если очередь пуста;
* приоритет переданного сообщения;
* номер освобождаемого семафора, когда сообщение передается в очередь.

Наконец, приведем перечень основных функций, управляющих работой очереди (без подробного описания передаваемых параметров, поскольку в различных ОС обращения к этим функциям могут существенно различаться):

* CreateQueue — создание новой очереди;
* OpenQueue — открытие существующей очереди;
* ReadQueue — чтение и удаление сообщения из очереди;
* PeekQueue — чтение сообщения без его последующего удаления из очереди;
* WriteQueue — добавление сообщения в очередь;
* CloseQueue — завершение использования очереди;
* PurgeQueue — удаление из очереди всех сообщений;
* QueryQueue — определение числа элементов в очереди.

Контрольные вопросы и задачи

1. Какие последовательные вычислительные процессы мы называем параллельными и почему? Какие параллельные процессы называются независимыми, а какие — взаимодействующими?
2. Изложите алгоритм Деккера, позволяющий разрешить проблему взаимного исключения путем использования одной только блокировки памяти.
3. Объясните, как действует команда проверки и установки. Расскажите о работе команд BTS и BTR, которые имеются в процессорах с архитектурой ia32.
4. Расскажите о семафорах Дейкстры. Чем обеспечивается взаимное исключение при выполнении примитивов P и V?
5. Изложите, как могут быть реализованы семафорные примитивы для мультипроцессорной системы?
6. Что такое мьютекс?
7. Изложите алгоритм решения задачи «поставщик-потребитель» при использовании семафоров Дейкстры.
8. Изложите алгоритм решения задачи «читатели-писатели» при использовании семафоров Дейкстры.
9. Что такое «монитор Хоара»? Приведите пример такого монитора.
10. Что представляют собой почтовые ящики?
11. Что представляют собой конвейеры (программные каналы)?
12. Что представляют собой очереди сообщений? Чем отличаются очереди сообщений от почтовых ящиков?