Министерство образования Республики Беларусь
Учреждение образования
«Белорусский государственный университет
информатики и радиоэлектроники»

Кафедра электронных вычислительных машин

В. А. Прытков, А. А. Уваров, В. А. Супонев

ТИПОВЫЕ МЕХАНИЗМЫ СИНХРОНИЗАЦИИ ПРОЦЕССОВ

Учебно-методическое пособие
по дисциплине «Системное программное обеспечение ЭВМ»
для студентов специальности I-40 02 01
«Вычислительные машины, системы и сети»

Минск 2007

УДК 004.451
ББК 32.973.26 – 018.2
П 85

Р е ц е н з е н т
ведущий научный сотрудник
лаборатории № 222 ОИПИ НАН Беларуси,
канд. техн. наук А. А. Дудкин

Прытков, В. А.

П 85 Типовые механизмы синхронизации процессов : учеб.-метод. пособие по дисц. «Системное программное обеспечение ЭВМ» для студ. спец. I-40 02 01 «Вычислительные машины, системы и сети» / В. А. Прытков, А. А. Уваров, В. А. Супонев. – Мн. : БГУИР, 2007. – 48 с. : ил.

ISBN 978-985-488-153-9

Учебно-методическое пособие посвящено вопросам синхронизации параллельных взаимодействующих процессов. Описаны основные проблемы, возникающие при этом, и пути их решения, рассмотрены типовые механизмы синхронизации, в том числе семафоры, мьютексы и мониторы. Приводится описание их реализации в популярных на сегодняшний день ОС Windows 2000/XP/Server 2003 и UNIX на примере Alt Linux 2.4, рассматривается решение некоторых типовых задач синхронизации с использованием различных механизмов. Предназначено для студентов всех форм обучения.

УДК 004.451
ББК 32.973.26 – 018.2

ISBN 978-985-488-153-9

Содержание

Введение

Большинство современных операционных систем (ОС) являются многозадачными, поддерживающими выполнение параллельных процессов. Для организации взаимодействия между такими процессами как при обмене информацией, так и при разделении доступа при конкуренции за ресурсы каждая ОС предоставляет определенные средства, которые являются реализацией тех либо иных механизмов синхронизации.

В результате от специалиста в области вычислительной техники требуется наличие навыков в организации взаимодействия между параллельными процессами. Однако корректное использование механизмов синхронизации процессов и устранение в системе источников потенциальных конфликтов между процессами невозможны без понимания принципов и идей, заложенных в основе синхронизирующих механизмов, четкого знания, что происходит с процессами в той или иной ситуации, так же как и деталей реализации механизмов в данной конкретной ОС.

Вместе с тем на рынке технической литературы по системному программному обеспечению сложилась парадоксальная ситуация. Он изобилует большим количеством литературы, ориентированной на конечного пользователя, являющейся, с точки зрения специалиста, слишком поверхностной и в лучшем случае позволяющей начать более-менее комфортную работу в ОС, описывая ее пользовательский интерфейс.

Литературы по системному программному обеспечению, ориентированной на специалистов, на рынке очень мало, и она значительно дороже. Однако, как правило, она представляет собой либо прекрасный обзор базовых механизмов и концепций без углубления в детали их реализации в конкретных ОС, либо справочную литературу по архитектуре, сервисным функциям и системным вызовам конкретной ОС без разъяснения теоретических основ данных механизмов и без сравнения с другими возможными вариантами реализации.

Данное пособие призвано в какой-то мере восполнить недостающее звено между теорией и практикой. В нем содержится как описание типовых механизмов синхронизации процессов, так и детали конкретной реализации в ОС Windows 2000/XP/Server 2003 и UNIX на примере Alt Linux 2.4. Объем пособия не позволяет рассмотреть все возникающие при взаимодействии процессов нюансы, вместе с тем оно обладает достаточной полнотой, охватывая решения наиболее часто встречающихся задач.

Раздел 1 является вводным. В нем кратко рассмотрены базовые понятия теории параллельных процессов. Разделы 2 и 3 посвящены механизмам взаимного исключения и типовым механизмам синхронизации; раздел 4 – реализации типовых механизмов синхронизации в наиболее распространенных ОС Windows и Linux.

1. Параллельные взаимодействующие процессы

1.1. Общие сведения о процессах

Все функционирующее на компьютере программное обеспечение (ПО), иногда включая ОС, организовано в виде набора последовательных процессов. **Процесс** – это программный блок, запущенный на выполнение, в совокупности с необходимыми для решения задачи данными, включая текущие значения счетчика команд, регистров и переменных. Обычно при загрузке ОС создается несколько процессов. Одни из них являются высокоприоритетными, другие – фоновыми. В большинстве ОС процессы возникают при запуске программ на выполнение.

В однопроцессорной системе в один момент времени реально может выполняться только один процесс. Диспетчер задач периодически переключает имеющиеся в системе процессы, передавая управление. При этом упрощенно диаграмму состояний процесса можно представить в следующем виде:

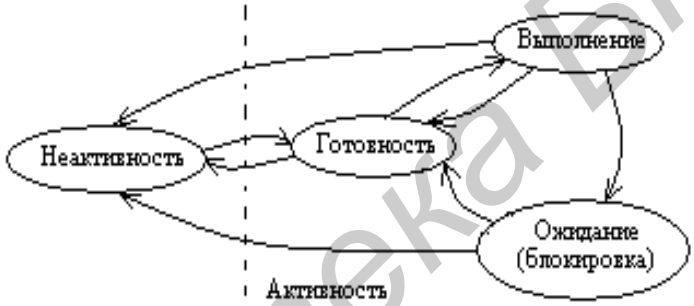

Рис. 1.1. Диаграмма состояний процесса

Процесс может находиться в **активном** и **пассивном** состоянии. В активном состоянии он участвует в конкуренции за ресурсы, в том числе и за процессорное время. Активный процесс может находиться в следующих состояниях:

- **выполнение** – все затребованные процессом ресурсы выделены, и он выполняется. В этом состоянии в однопроцессорной системе в каждый момент времени может находиться только один процесс;
- **готовность** – система может предоставить процессу все требуемые ресурсы, ни один из них не занят, за исключением процессора. Как только диспетчер предоставит управление процессу, он сможет начать (или продолжить) выполнение;
- **ожидание** – требуемые для выполнения задачи ресурсы не могут быть предоставлены процессу, они заняты другим процессом или происходит ожидание операции ввода-вывода. После освобождения ресурса или завершения операции процесс может перейти в состояние готовности.

Процесс имеет свое виртуальное адресное пространство, ему назначаются ресурсы – файлы, окна и др. Это позволяет защитить процессы друг от друга, и ОС считает процессы совершенно не связанными между собой.

В современных системах концепция процесса дополняется понятием потока (thread, нить). **Поток** можно рассматривать как достаточно независимую часть процесса, которая может выполняться отдельно, но разделяет общее адресное пространство процесса и его ресурсы. В зависимости от реализации диспетчер может выполнять переключения как процессов (в этом случае переключениями своих потоков управляет сам процесс, и системе ничего об этих потоках не известно), так и непосредственно потоков.

В данном пособии под термином «процесс» в дальнейшем подразумевается независимый объект диспетчеризации, не имеющий доступа к данным и ресурсам других подобных объектов.

Особенностью мультипрограммных ОС является то, что в их среде параллельно развивается несколько процессов. **Параллельные процессы** – это такие процессы, которые одновременно находятся в одном из активных состояний. Они могут быть независимыми либо взаимодействующими. **Независимыми** являются процессы, множество данных (переменных и файлов) которых не пересекается. Независимые процессы не влияют на работу друг друга. **Взаимодействующие** процессы совместно используют некоторые общие переменные либо иные ресурсы, и выполнение одного процесса может повлиять на выполнение другого.

Взаимодействовать могут либо конкурирующие, либо сотрудничающие процессы. Существует три основных типа взаимодействий:

- передача информации от одного процесса другому;
- конкуренция за ресурсы;
- согласование обмена данными: если процесс А поставляет данные, а процесс В использует их, то он должен ждать, пока процесс А не даст данные.

ОС должна иметь средства для организации взаимодействия между процессами.

В данном пособии рассматриваются механизмы синхронизации процессов, позволяющих организовать корректное их взаимодействие при конкуренции за ресурсы. Механизмы передачи информации между процессами – это отдельная большая тема, не входящая в предмет обсуждения в данном пособии.

1.2. Понятие состязания

Рассмотрим ситуацию, когда два процесса используют один и тот же ресурс, не синхронизируя при этом свои действия. Пусть каждый из процессов хочет вывести файл на печать. Для этого он читает индекс свободной ячейки в очереди заданий на печать, помещает туда имя файла и увеличивает индекс. Демон печати обрабатывает эту очередь. Пусть в очереди уже есть два задания, и номер свободной ячейки – 3. Процесс А читает индекс, это число 3. Тут вре-

мя, выделенное ему диспетчером задач, заканчивается, и управление переходит к процессу B. Он также читает индекс, это по-прежнему 3. Он помещает имя файла по этому индексу и увеличивает значение индекса до 4. Тут управление возвращается к процессу A. Индекс им уже был прочитан, это 3, и процесс заносит имя файла по тому же адресу, изменяя индекс на 4. В итоге имя, занесенное процессом B, затирается, и этот файл никогда не будет напечатан. С точки зрения демона, ошибок не произошло – число заданий в очереди находится в полном соответствии с индексом.

Ситуация, когда несколько процессов считывают или записывают данные одновременно и результат зависит от того, кто из них был первым, называется **состоянием состязания** (гонки), а ресурс, не допускающий одновременного использования, называется **критическим**.

1.3. Типовые задачи синхронизации

Подавляющее большинство практических задач, требующих синхронизации параллельных процессов, так или иначе сводится к нескольким типовым задачам, некоторые из которых и будут рассмотрены в данном пособии.

Задача об обедающих философах. Эта задача сформулирована Дейкстрой в 1965 году. В одном доме живет пять философов. В течение дня они предаются размышлениям либо обедают. Для обеда имеется круглый стол, на котором находятся пять тарелок со спагетти и пять вилок (рис. 1.2).

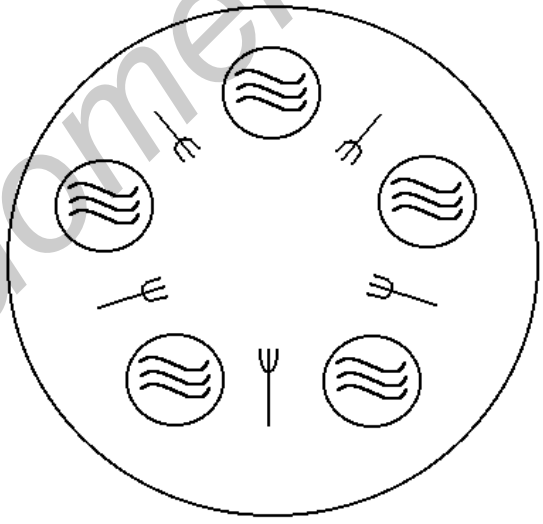

Рис. 1.2. Задача об обедающих философах

Чтобы пообедать, необходимо взять две вилки. Когда философ голоден, он подходит к столу, садится, берет одну вилку, затем вторую (справа и слева от тарелки соответственно), ест, кладет вилки на место, после чего возвращается к размышлениям. Однако если все философы подойдут к столу одновременно и одновременно возьмут правые вилки, то никто из них не сможет взять вторую вилку, поскольку все вилки будут заняты. Ситуация, когда существует такая последовательность процессов, каждый из которых владеет частью критического

ресурса, другая часть которого находится у следующего в данной последовательности процесса, а для продолжения работы им необходим ресурс целиком, называется **взаимоблокировкой.**

Здесь же просматривается и еще одна проблема. Если один философ очень медлителен, а его сосед очень быстр, то медленный философ рискует умереть от голода: он подходит к столу, берет вилку, в этот момент быстрый сосед подбегает, хватает обе вилки, начинает есть; медленный кладет вилку, чтобы повторить попытку позже; быстрый, пообедав, кладет вилки и отходит; медленный берет вилку, а в это время быстрый, снова проголодавшись, подбегает, хватает обе вилки, начинает есть; медленный кладет вилку… и так до бесконечности. Это проблема называется **голоданием**.

Необходимо организовать взаимодействие философов таким образом, чтобы исключить взаимоблокировку и голодание.

Задача «производитель-потребитель». Существует два процесса. Процесс-производитель создает некие изделия (объекты), по мере изготовления помещая их на склад (в буфер) ограниченного размера. Процесс-потребитель периодически берет изделия со склада для личных нужд.

Попробуем решить задачу следующим образом. Производитель проверяет состояние буфера, и если место свободно, помещает в него объект. Если буфер и так уже полон, процесс переходит в состояние ожидания, пока не появится место. Если же буфер был пуст, требуется сообщить потребителю, что объекты уже появились. Потребитель действует похожим образом. Он проверяет буфер, и если буфер не пуст, то берет из него объект. Если буфер пуст, то потребитель переходит в состояние ожидания, пока не появятся объекты; если же буфер был полностью заполнен, то он сообщает производителю, что место уже появилось. Очевидно, что число объектов в буфере хранится в некоторой переменной-счетчике.

```
// глобальные переменные и описания
int count=0;                  // число заполненных ячеек
#define N 100                 // длина буфера

// процесс-производитель
void producer(void)
{int item;
 while (1)
  {item=produce_item();       // сформировать новый элемент
   if (count==N) sleep();     // если буфер полон, ожидание
   insert_item(item);         // поместить элемент в буфер
   count++;                   // изменить значение счетчика
   if (count==1) wakeup(consumer);}} // если до этого буфер был
                              // пуст, разбудить потребителя
// процесс-потребитель
void consumer(void)
{int item;
 while (1)
  {if (count==0) sleep();     // если буфер пуст, ожидание
```

```
item=remove_item();       // взять элемент из буфера
count--;                  // изменить счетчик
if(count==N-1) wakeup(producer); //если буфер был заполнен,
                          // разбудить производителя
consume_item(item);}}     //использовать элемент
```

Рассмотрим вариант развития ситуации (рис. 1.3). На рисунке представлены существенные фрагменты кода, выполняемые в данный момент процессором, с разбивкой по процессам. Здесь и далее в подобных диаграммах фрагмент «//true» будет означать истинность проверки условия в контексте процесса. Это значит, что для другого процесса проверка условия может дать иные результаты в зависимости от развития ситуации. Фрагмент «###» будет означать блокировку процесса, а «=>» – его пробуждение после блокировки. В этом случае процесс переходит в состояние выполнения либо готовности в зависимости от выбора планировщика. Следует учитывать, что переключение задач планировщиком в обсуждаемом порядке является возможным, но вовсе не обязательным. Именно поэтому ошибки в синхронизации так тяжело обнаружить – они проявляются не при каждом запуске программы.

Count	Производитель	Потребитель
N N	*produce_item()* *if (count==N) //true*	
N N N-1 N-1 N-1 N-1		*if (count==0) //false* *remove_item()* *count--* *if(count==N-1) // true* *wakeup(producer)* *consume_item(item)*
N-1	*sleep() //###*	
 0 0		… *if (count==0) //true* *sleep() // ###*

Рис. 1.3. Неправильная организация взаимодействия процессов

Рассмотрим эту ситуацию. Производитель подготовил объект и пытается поместить его в буфер, который заполнен до отказа (count=N). Он читает значение счетчика, сравнивает его с максимальным, но перейти в состояние ожидания не успевает, поскольку планировщик переключает процессы, передавая управление потребителю. Тот читает счетчик, берет элемент из буфера, уменьшает счетчик и, поскольку буфер был полон, посылает сигнал производителю на выход из состояния ожидания. Однако производитель не находится в состоянии ожидания, и сигнал пропадает. Когда, наконец, планировщик вновь переключит процессы, производитель благополучно перейдет к ожиданию, несмотря на то что состояние буфера уже поменялось. В результате потребитель выберет из буфера все элементы и также перейдет в состояние ожидания. Оба про-

цесса будут ожидать – один освобождения буфера, другой – его заполнения. Аналогичные проблемы возникают при пустом буфере.

Проблема заключается в том, что производитель не успел перейти в состояние ожидания и сигнал активации пропал впустую, создавая типичную ситуацию гонок. Одно из решений проблемы – в использовании бита активации. Если сигнал активации посылается процессу, который не находился в состоянии ожидания, то бит устанавливается. Если процесс пытается уйти в состояние ожидания, проверяется этот бит. Если он установлен, то процесс не переводится в состояние ожидания, а всего лишь сбрасывает этот бит, продолжая обработку. Однако задача может быть обобщена на случай N производителей и M потребителей, и одного бита становится недостаточно.

Задача «читатели-писатели». Эта задача возникает, когда одни процессы (читатели) могут просматривать некоторые объекты (например записи таблицы в базе данных или элементы буфера), а другие (писатели) могут изменять состояние данного списка (добавлять, удалять элементы, изменять его значение и т.п.). Чтобы у читателей всегда была достоверная информация, доступ писателей требуется организовать монопольно (т.е. только один писатель в один момент времени может иметь доступ к ресурсу), а доступ читателей – совместно, для повышения производительности системы.

Решение может выглядеть следующим образом. Используется счетчик читателей, уже получивших доступ к ресурсу. Если ресурс не захвачен писателем, то читатель получает доступ и увеличивает счетчик. При окончании работы читатель счетчик уменьшает. Писатель проверяет значение счетчика и захватывает ресурс, только если счетчик равен нулю.

Даже если решить проблему состязания, используя механизмы синхронизации, о которых речь пойдет ниже, может оказаться так, что писатель будет ждать доступа бесконечно, если новый читатель успевает получить доступ, пока предыдущий читатель еще работает с ресурсом. Решение этой задачи будет приведено ниже.

Задача о парикмахерской. Это еще одна классическая задача, вызывающая гонки. Имеется парикмахерская с одним мастером, его рабочее кресло и несколько стульев для посетителей, ожидающих в очереди. Требуется организовать работу следующим образом: если посетителей нет, мастер спит в своем кресле. Как только приходит новый клиент, он будит мастера и получает обслуживание. Если клиент приходит, когда мастер занят работой, то он садится на стул в очередь. Если все стулья заняты, он просто уходит без ожидания.

1.4. Понятие о взаимном исключении

Для предотвращения состязания необходим механизм **взаимного исключения**, не позволяющий процессам обращаться к критическому ресурсу одновременно. В рассмотренной выше ситуации с принтером проблема возникла из-за того, что процесс В начал работу с ресурсом до того, как процесс А закончил

с ним работать. Кроме реализации в ОС средств для организации взаимного исключения, в ней должны быть средства для синхронизации процессов с целью обмена данными. Фрагмент кода, в котором происходит обращение к критическим ресурсам, называется **критической областью,** или критической секцией (КС). Решение задачи взаимного исключения в том, чтобы организовать такой доступ к критическому ресурсу, когда только одному процессу разрешается находиться в критической области.

Если один из процессов владеет критическим ресурсом, остальные процессы должны получить отказ и ждать освобождения ресурса. При этом если процессы выполняют операции, не приводящие к конфликтам, т.е. вне критической области, то они должны иметь возможность параллельной работы. Если процесс, имеющий доступ к критическому ресурсу, выходит из своей критической области, доступ должен быть передан другому процессу, ожидающему доступа. Ситуация взаимного исключения поясняется на рис. 1.4.

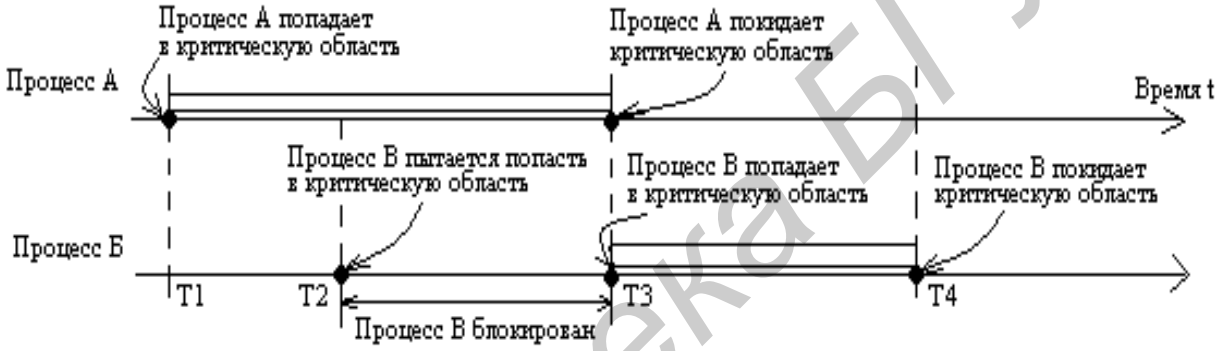

Рис. 1.4. Механизм взаимного исключения

Для корректной организации взаимодействия параллельных процессов необходимо выполнение следующих 4 условий:

- в любой момент времени только один процесс должен находиться в своей критической области (условие взаимного исключения);
- никакой процесс, находящийся вне своей критической секции, не должен влиять на выполнение других процессов, ожидающих входа в критические области, т.е. он не должен блокировать критическую область другого процесса; если несколько процессов одновременно хотят войти в критические области, то принятие решения о том, кому предоставить доступ, не должно откладываться бесконечно долго (условие прогресса);
- ни один процесс не должен ждать бесконечно долго вхождения в критическую область и, следовательно, ни один процесс не должен находиться в критической секции бесконечно долго (условие ограниченного ожидания);
- не должно быть никаких предположений о скорости или количестве процессов в системе.

Все современные системы имеют такое средство для организации взаимного исключения, как блокировка памяти, запрещающее одновременное исполнение двух или более команд, которые обращаются к одной и той же ячейке памяти, однако для полноценной поддержки взаимного исключения его недостаточно.

2. Организация взаимного исключения

2.1. Простейшие механизмы

Запрещение прерываний. Самый простой способ организации взаимного исключения состоит в запрещении прерываний при входе в критическую область. Поскольку переключение задач происходит по прерыванию, то отключение прерываний исключает передачу процессора другому процессу, что и требовалось. Однако существуют довольно серьезные доводы против такого подхода. Во-первых, при этом запрещаются и прерывания от таймера. Кроме того, в результате сбоя процесс может не вернуть систему прерываний в разблокированное состояние. Далее, в многопроцессорной системе команда блокирования прерываний повлияет только на один процессор, а ведь другие процессы могут выполняться и на другом процессоре.

Использование переменных для блокировки. Другое решение состоит в использовании специальных переменных для блокировки. Выделяется переменная, изначально равная нулю. Если процесс желает попасть в критическую область, он считывает эту переменную, и если она равна 0, то устанавливает ее в 1 и входит в критическую область. Если же переменная равна 1, то процесс ждет, пока она не установится в 0. Однако такое решение позволяет нескольким процессам одновременно войти в критические области, поскольку возникают гонки уже при доступе к переменным блокировки.

Чередование доступа. Еще одним вариантом организации взаимного исключения является чередование доступа к критическому ресурсу. Для этого имеется общая переменная, указывающая, чья очередь входить в критическую область:

```
// глобальные переменные, доступные обоим процессам
int turn;               // переменная, указывающая, чей доступ

// описания, различные для каждого из процессов
#define P_NUM 0         // 0 для процесса A, 1 - для B

// часть кода, идентичная у обоих процессов
void process (void)
{while (1)
{while(turn!=P_NUM);    // ожидание своей очереди
 critical_section();    // выполнение критической секции
 turn=1-P_NUM;          // передача очереди
 non_critical_section();}}  // выполнение некритического кода
```

Постоянная проверка значения переменной в ожидании некоторого значения называется **активным ожиданием**. При этом нерационально тратится процессорное время, поскольку процесс фактически вхолостую простаивает в цикле.

Здесь возможны и другие проблемы. Например, некритическая секция процесса B значительно длиннее, чем у процесса A. Пусть очередь доступа у процесса A. Он входит в критическую область. Процесс B в это время уже находится в некритической секции. Процесс A выполняет критическую область, передает очередь, выполняет некритическую область. Процесс B по-прежнему находится в некритической области. Процесс A переходит к ожиданию в цикле, пока процесс B не передаст очередь. Но при этом процесс B находится вне критической секции, и выполнит ее еще очень не скоро. Таким образом, нарушается одно из условий, рассмотренных ранее: процесс, находящийся вне критической области, влияет на функционирование процесса, ожидающего доступа. Фактически требуется, чтобы процессы попадали в критические секции строго поочередно, ни один из процессов не может попасть в критическую секцию дважды подряд.

Volatile. Даже если синхронизация успешно решена без помощи специальных механизмов, этого может быть недостаточно, если используются оптимизирующая компиляция программы. В этом случае измененное значение переменной может оставляться в регистре процессора, а не заноситься сразу в заданную ячейку памяти, что опять приводит к гонкам. В стандарте ANSI C описан спецификатор volatile, позволяющий гарантировать, что переменная после изменения будет возвращена в память.

2.2. Алгоритм Деккера

Датским математиком Деккером впервые был предложен алгоритм взаимного исключения, не требующий строгого чередования (рис. 2.1). По сути он объединяет два предыдущих подхода. Он основан на наличии трех переменных: req[0], req[1], turn, отвечающих за требования процессов на вхождение в критическую область, и чья очередь на вхождение при условии, если оба процесса требуют ресурс.

```
// глобальные переменные, доступные обоим процессам,
// изначально все в значении 0
int turn;              // переменная, указывающая, чей доступ
int req[2];            // запросы на вхождение в КС

// описания, различные для каждого из процессов
#define P_NUM 0        // 0 - для процесса A, 1 - для B

// часть кода, идентичная у обоих процессов
void process (void)
{while (1)
  {req[P_NUM]=1;       // запрос на вхождение в КС
L: if (req[1-P_NUM]==1)// если есть второй запрос
```

```
 {if(turn==P_NUM)   // и очередь текущего процесса
   {goto L;}        // ожидание снятия второго запроса
  else              // если же при этом чужая очередь
   {req[P_NUM]=0;   // то снятие запроса
    while (turn!=P_NUM);}} // ожидание своей очереди
else                // если других запросов не было
 {critical_section();// выполнение критической секции
  turn=1-P_NUM;     // передача очереди другому процессу
  req[P_NUM]=0;}    // снятие запроса
non_critical_section();}}
```

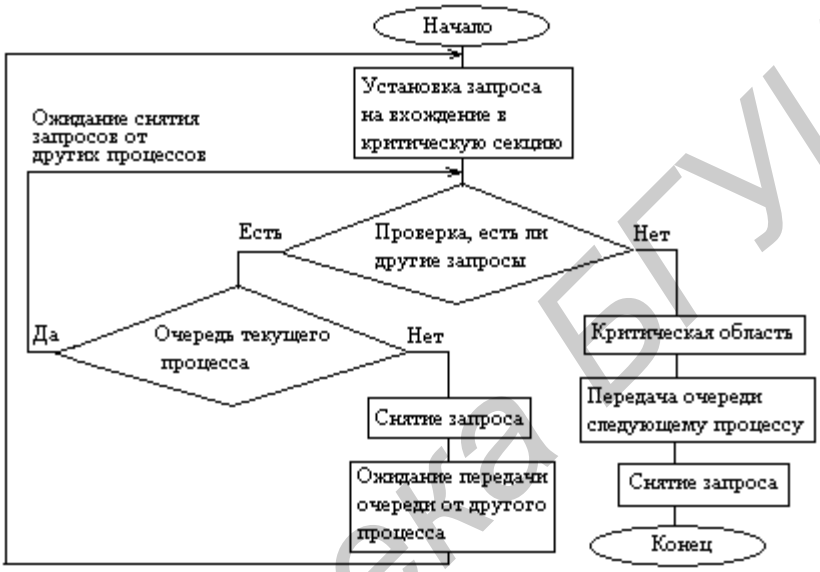

Рис. 2.1. Алгоритм Деккера

Если установлен флаг запроса от процесса А и нет флага запроса от процесса В, то выполняется критическая секция процесса А независимо от того, чья очередь, и очередь передается другому процессу. Если же установлены оба флага, то выполняется критическая секция того процесса, чья очередь. Второй процесс при этом ожидает передачи очереди. Алгоритм Деккера позволяет гарантированно решить проблему критических интервалов. Флаги запроса обеспечивают невозможность одновременного вхождения в критическую область, переменная очереди избавляет от взаимной блокировки. Однако в случае обобщения на N процессов алгоритм усложняется.

2.3. Алгоритм Петерсона

В 1981 году Петерсоном был разработан более простой алгоритм взаимного исключения, состоящий из двух независимых частей, которые могут быть оформлены отдельными процедурами. Первая вызывается перед вхождением в критическую область, вторая – по окончании ее.

```
// глобальные переменные, доступные обоим процессам,
// изначально все в значении 0
int turn;               // переменная, указывающая, чей доступ
```

```
int req[2];             // запросы на вхождение в КС
// описания, различные для каждого из процессов
#define P_NUM 0         // 0 - для процесса A, 1 - для B
// часть кода, идентичная у обоих процессов
void process(void)
{while (1)
  {req[P_NUM]=1;        // запрос на вхождение в КС
   turn=P_NUM;          // установка очереди на себя
// ожидание, если своя очередь, но есть второй запрос
   while (turn==P_NUM && req[1-P_NUM]==1);
   critical_section(); // выполнение критической секции
   req[P_NUM]=0;        // снятие запроса
   non_critical_section();}}
```

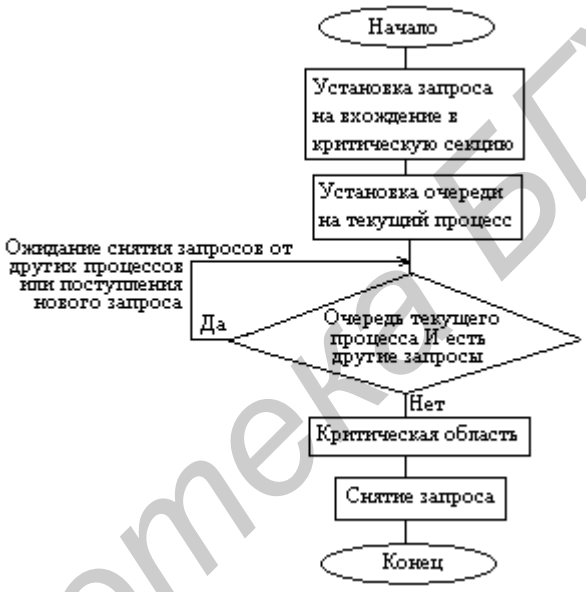

Рис. 2.2. Алгоритм Петерсона

Изначально оба процесса вне критических областей. Процесс А вызывает функцию вхождения в критическую область, устанавливает там флаг запроса в 1 и переменную очереди в 0. Поскольку процесс В не устанавливал флаг запроса, то процесс А входит в критическую область, затем после ее выполнения в функции выхода снимает запрос. Если оба процесса вызывают функцию вхождения практически одновременно, происходит следующее. Процесс А устанавливает флаг запроса и устанавливает переменную очереди на свой номер (0). В это время процесс В устанавливает свой флаг запроса и изменяет переменную очереди на свой номер (1). Проверяется условие цикла: от процесса А есть запрос и очередь установлена на текущий процесс. Процесс В переходит к активному ожиданию. Тем временем процесс А приступает к определению истинности условия цикла: запрос от процесса В есть, но очередь также установлена на него. Условие ложно, ожидания не происходит, и процесс А входит в критическую область. После ее прохождения он снимает флаг запроса. В этот момент у

процесса B условие также становится ложным, поскольку уже нет запросов от других процессов, и он входит в критическую секцию.

2.4. Аппаратная поддержка в процессорах x86

Существуют аппаратные механизмы организации взаимного исключения. Классическим примером является операция Test & Set («Проверка и установка»). В IBM360 эта команда называлась TS. Команда имеет два операнда и выполняется следующим образом. Значение второго операнда присваивается первому, после чего второй операнд устанавливается в единицу. Особенность команды в том, что она является неделимой, т.е. оба эти действия выполняются неразрывно.

Для возможности использования этой операции требуется одна общая переменная, доступная всем процессам. Переменная должна принимать значение 1, если какой-либо процесс находится в своей критической области. Кроме того, с каждым из процессов связана еще и своя персональная переменная, устанавливаемая в 1, если процесс хочет войти в критическую секцию. TS будет использоваться так: значение общей переменной будет считываться в локальную и устанавливаться в 1.

```
// глобальные переменные, доступные всем процессам
int common=0;                 // флаг занятости КС

// часть кода, идентичная у всех процессов
void process(void)
{int proc;                    // персональная переменная запроса
 while(1)
  {proc=1;                    // запрос на вхождение в КС
   while(proc==1) TS(proc, common);// ожидание разрешения на вход
   critical_section();        // выполнение критической секции
   common=0;                  // сброс флага занятости
   non_critical_section();}}// выполнение некритического кода
```

Происходит следующее. Допустим, что в критической секции процесса нет. В этом случае все переменные равны 0. Процесс A хочет получить доступ к критическому ресурсу. Он выставляет флаг. Операция TS выполняет proc=common, common=1, а поскольку common была 0, то фактически запрос снимается, флаг входа в КС устанавливается в 1, процесс выходит из цикла и входит в КС. Допустим, что в этот момент процесс B устанавливает свой флаг. Он входит в цикл, TS выполняет proc=common, common=1. Поскольку common и так уже в 1, т.е. в КС находится другой процесс, то запрос не снимается, так же как и сам флаг не изменяет значения. Процесс B ожидает в цикле. Наконец процесс A выходит из КС и снимает флаг. Процесс B в очередной раз выполняет TS, поскольку common=0, то запрос снимается, флаг входа в КС устанавливается, процесс выходит из цикла и входит в КС.

В современных 32-разрядных микропроцессорах платформы x86 есть специальные команды, являющиеся разновидностью TS: BTC, BTS, BTR. BTS (bit test & set) также имеет два операнда: **BTS Op, B**. Процессор сохраняет бит с

номером B переменной Op во флаге CF (carry - перенос), и устанавливает бит B переменной Op в 1. В качестве номера бита B может быть указано либо непосредственно число, либо 16- или 32-разрядный регистр процессора, в котором этот номер хранится. Этот индекс берется по модулю 32, т.е. использует только 5 младших бит числа, соответственно индекс находится в диапазоне 0 – 31, что позволяет выбрать любой бит в пределах 4-байтной переменной или регистра. В качестве переменной Op указывается 16- или 32-разрядная область памяти (переменная) либо 16- или 32-разрядный регистр процессора.

```
L: BTS   m, 1      ; бит 1 глобальной переменной m – флаг
                   ; нахождения в КС какого-либо процесса
   JC L            ; пока в КС есть процесс, ожидание
                   ; флаг переноса выступает в качестве
                   ; локальной переменной
   CALL critical_section
   AND m, 0fffffffdh   ; сброс флага нахождения в КС
```

Однако и у этой операции имеется тот недостаток, что ожидающий процесс находится в состоянии активного ожидания. По сути процессы, обнаружив, что доступ к критическому ресурсу закрыт, должны перейти в состояние блокировки, а как только ресурс освободится, иметь возможность выйти из нее для использования ресурса. Очевидно, что если процесс при ожидании ресурса не должен использовать процессорное время, то сам он не сможет выйти из этого состояния, и требуются соответствующие механизмы ОС.

Команда BTR Op, B выполняет полностью аналогичные действия, но бит B сбрасывается в 0. Команда BTC Op, B инвертирует значение бита B. Обе эти команды сохраняют прежнее значение бита во флаге переноса.

Существует и еще одна проблема при использовании TS или алгоритма Петерсона. Это **проблема инверсии приоритетов**. Пусть имеется 2 процесса: процесс с высоким приоритетом (H) и с низким (L), которым требуется один и тот же ресурс. Алгоритм работы планировщика таков, что всегда запускается более приоритетный процесс. Допустим, процесс L находится в критической области. В этот момент запускается процесс H, который попадает в состояние активного ожидания. Процесс L не получит процессорное время, поскольку его приоритет ниже, и поэтому не сможет завершить критическую секцию. Процесс H останется в состоянии активного ожидания, а поскольку его приоритет выше, планировщик не передаст управление процессу L. В результате высокоприоритетный процесс будет фактически заблокирован низкоприоритетным.

3. Типовые механизмы синхронизации

3.1. Семафоры и мьютексы

Понятие семафоров было введено Дейкстрой. Семафор S – это переменная специального типа, доступная параллельным процессам для проведения над ней только двух неделимых операций: закрытия P(S) и открытия V(S). Посколь-

ку эти операции неделимы (т.е. выполняются атомарно и не могут быть прерваны в процессе своего выполнения), то они исключают друг друга (не может оказаться так, что один процесс выполняет одну из них, а другой – другую), т.е. в один момент времени в однопроцессорной системе может выполняться только одна из них.

Семафорный механизм работает по следующей схеме: вначале исследуется состояние критического ресурса (операция P), определяемое значением семафора. В зависимости от результата происходит или предоставление ресурса или ожидание доступа в очереди в режиме «пассивного ожидания». В состав механизма включаются специальные средства формирования и обслуживания очереди ожидающих процессов ОС. В силу неделимости операций, даже если некоторые процессы одновременно захотят использовать критический ресурс, доступ получит только один, а второй будет помещен в очередь. Процессам из очереди не предоставляется процессорное время, пока ресурс занят. При выходе из критической области процесс вызывает операцию V, позволяющую предоставить доступ к ресурсу процессу, ожидающему в очереди.

Допустимыми значениями семафоров являются целые числа. Семафор называется **двоичным**, если максимальное значение, которое он может принять, это 1. Если больше, то семафор – N-ричный. Рассмотрим пример реализации. Допустим, семафор S инициализируется ОС в значение 1. Собственно процессы будут иметь следующий вид:

```
void process(void)
{while(1)
  {P(&S);                 // прохождение семафора
   critical_section(); // операция успешна или процесс из очереди
   V(&S);                 //восстановление семафора
   non_critical_section();}}
```

Операции P и V могут быть реализованы следующим образом:

```
void P (int *S)
{(*S)--;                   // закрытие семафора и доступа
 if (*S<0) block_process();}// если семафор уже был закрыт,
                           // поместить процесс в очередь блокировки
void V (int *S)
{if (*S<0) activate_process();// перевести 1 процесс из очереди
                           // блокированных в очередь готовых
 (*S)++;}                  // открыть семафор
```

Пусть оба процесса пытаются выполнить P(S), и это успешно удается процессу B. Он устанавливает семафор в 0 и переходит к выполнению КС. Тем временем процесс A пытается выполнить P(S). Он устанавливает семафор в –1, и помещается ОС в очередь ожидания. Процесс B выполняет КС и вызывает V(S). Поскольку семафор<0, т.е. есть очередь, то процесс A выводится из очереди ожидающих и помещается планировщиком в очередь готовых к выполнению процессов. Процесс B восстанавливает значение семафора до 0. В итоге противоречий нет: семафор закрыт, в очереди нет процессов. Когда планиров-

щик передаст управление процессу A, тот выйдет из процедуры P(S) и перейдет к выполнению критической секции. Когда процесс A окончит работу, вызвав V(S), значение семафора восстановится до 1, открывая его.

Возможны и другие реализации семафоров. Неделимость примитивов P и V обеспечивается в однопроцессорной системе простым запретом прерываний на время их выполнения. В многопроцессорных системах это проблему не решит, поскольку доступ к семафору по-прежнему будут иметь несколько процессов одновременно.

Одним из вариантов организации работы с семафором являются **мьютексы** mutex (mutual exclusion – взаимное исключение). Они представляют собой простейшие двоичные семафоры, реализованные во многих ОС, которые могут находиться только в одном из двух состояний – открыт или закрыт. Соответственно для реализации требуется всего 1 бит, хотя обычно используют переменную типа int, используя только два ее значения – 0 и 1.

Значение мьютекса устанавливается 2 процедурами. Если процесс хочет войти в КС, он вызывает процедуру закрытия мьютекса, например mutex_lock. Если мьютекс открыт, то запрос выполняется, и вызывающий процесс может попасть в КС. Если же мьютекс закрыт, то процесс блокируется, пока другой процесс, находящийся в КС, не выйдет из нее, открыв мьютекс соответствующей процедурой, например mutex_unlock. Пример реализации мьютекса, у которого 0 – открытое состояние, 1 – заблокированное:

```
mutex_lock:
   bts mutex, 1   ; закрыть мьютекс
   jnc OK         ; если мьютекс был открыт, конец процедуры
   call thread_yield; иначе переключение процесса
   jmp mutex_lock ;новая попытка
OK:ret;

mutex_unlock:
   move mutex, 0  ;открыть мьютекс
   ret;
```

Здесь при вызове mutex_lock тестируется значение бита глобальной переменной mutex. Если после выполнения операции флаг переноса равен 0 (мьютекс был открыт), то процедура завершается. Если же мьютекс был закрыт (CF=1), то выполняется системный вызов thread_yield, передающий управление другому потоку. В результате активного ожидания не происходит в отличие от прямого использования операции TS. При следующем вызове потока процедура проверки мьютекса начнется заново.

Однако существует еще одна проблема, связанная с синхронизацией. В рассмотренных ранее случаях (алгоритмы Деккера и Петерсона, семафоры) предполагалось, что имеется общая глобальная переменная. Однако процессы имеют каждый свое адресное пространство. Современные ОС, как правило, решают эту задачу двумя вариантами:

- совместно используемые переменные, например, семафоры, хранятся в ядре с доступом через системные запросы;
- процессам разрешается использовать некоторую общую область памяти.

3.2. Решения некоторых типовых задач

Задача «производитель-потребитель». Эта задача уже рассматривалась выше, и было показано, какие проблемы возникают при ее решении. Рассмотрим решение с помощью семафоров.

```
// глобальные переменные и описания
#define N 100           // размер буфера
typedef int semaphore;
semaphore mutex=1;      // контроль доступа в КС
semaphore empty=N;      // число пустых ячеек
semaphore full=0;       // число заполненных ячеек

void producer(void)
{int item;
 while (1)
  {item=produce_item();// создание данных
   P(&empty);          // уменьшение счетчика пустых ячеек
   P(&mutex);          // вход в критическую область
   insert_item(item);  // размещение элемента в буфер
   V(&mutex);          // выход из КС
   V(&full);}}         // увеличение счетчика полных ячеек

void consumer (void)
{int item;
  while (1)
  {P(&full);           // уменьшить число занятых ячеек
   P(&mutex);          // вход в КС
   item=remove_item(); // взять данные из буфера
   V(&mutex);          // выход из КС
   V(&empty);          // увеличить число пустых ячеек
   consume_item(item);}}// обработать элемент
```

Empty	Full	Mutex	Производитель	Потребитель
N	*0*	*1*		*P(&full) // ###*
N	*-1*	*1*	*P(&empty)*	
N-1	*-1*	*1*	*P(&mutex)*	
N-1	*-1*	*0*	*KC*	
N-1	*-1*	*0*	*V(&mutex)*	
N-1	*-1*	*1*	*V(&full)*	=>
N-1	*0*	*1*		*P(&mutex)*
N-1	*0*	*0*		*KC*
N-1	*0*	*0*	*P(&empty)*	
N-2	*0*	*0*	*P(&mutex) // ###*	
N-2	*0*	*-1*	=>	*V(&mutex)*

N-2	*0*	*0*	*КС*	
N-2	*0*	*0*		*V(&empty)*
N-1	*0*	*0*	*...*	

Рис. 3.1. Синхронизация с использованием семафоров

Используется три семафора: для пустых сегментов, для полных и для исключения одновременного доступа к буферу. Первые два используются не для разграничения доступа, а для синхронизации процессов.

Рассмотрим ситуацию, когда буфер пуст (рис. 3.1). В этом случае потребитель будет сразу заблокирован на семафоре. Производитель начнет работу, выполнит КС, освободит мьютекс и откроет семафор full. В зависимости от конкретной реализации механизма семафоров, численные их значения могут отличаться от приводимых на рисунке, например, ОС Windows, так же как и Linux, не допускает отрицательных значений семафора, а подсчет заблокированных процессов выполняет системными средствами, скрытыми от пользователя. Потребитель входит в КС, после чего планировщик передаст управление производителю. Производитель пройдет семафор empty, но перейдет к ожиданию на мьютексе. Как только потребитель освободит мьютекс, планировщик активизирует ожидающий на мьютексе процесс, без изменения значения мьютекса.

Задача «читатели-писатели». Рассмотрим, каким образом неправильное решение может привести к бесконечному ожиданию писателей.

```
typedef int semaphore;
semaphore mutex = 1;   // контроль доступа к переменной rc
semaphore db=1;        // контроль доступа к базе данных
int rc=0;    // кол-во процессов читающих или желающих читать

void reader(void)      // процесс читатель
{while(1)
  {P(&mutex);          // закрыть доступ к счетчику читателей
   rc++;               // увеличить его
   if (rc==1) P(&db);  // если первый, закрыть доступ к БД
   V(&mutex);          // открыть доступ к счетчику
   read_database();    // чтение БД
   P(&mutex);          // закрыть доступ к счетчику
   rc=rc-1;            // уменьшить его
   if (rc==0) V(&db);  // если нет других читателей (процесс
                       // последний), открыть доступ к БД
   V(&mutex);          // открыть доступ к счетчику
   use_read_data();}}  // использование данных

void writer(void)      // процессы писатели
{while(1)
  {make_data();        // подготовка данных к записи
   P(&db);             // закрыть доступ
   write_database();   // записать данные
   V(&db); }}          // открыть доступ
```

Первый читатель закрывает доступ к БД. Остальные всего лишь увеличивают счетчик читателей. Когда базу покинет последний читатель, он открывает к ней доступ. Если один читатель уже работает с базой, второй тоже получит к ней доступ и т.д. Если в этот момент доступ запросит писатель, он перейдет к ожиданию, поскольку доступ закрыт. Доступ писатель получит только тогда, когда базу покинет последний читатель.

Mutex	Db	Rc	Читатель 1	Читатель 2	Писатель
1	*1*	*0*		*P(&mutex)*	
0	*1*	*0*	*P(&mutex) // ###*		
-1	*1*	*0*		*rc++*	
-1	*1*	*1*		*if (rc==1) P(&db) // (true)*	
-1	*0*	*1*			*P(&db) // ###*
-1	*-1*	*1*	*=>*	*V(&mutex)*	
0	*-1*	*1*	*rc++*		
0	*-1*	*2*	*if (rc==1) // (false)*		
0	*-1*	*2*	*V(&mutex)*		
1	*-1*	*2*	*KC*		
1	*-1*	*2*	*P(&mutex)*		
0	*-1*	*2*	*rc--*		
0	*-1*	*1*	*if(rc==0) // (false)*		
0	*-1*	*1*	*V(&mutex)*		
1	*-1*	*1*		*KC*	
				...	
0	*-1*	*0*		*if(rc==0) V(&db) // (true)*	*=>*
0	*0*	*0*			*KC*
0	*0*	*0*		*V(&mutex)*	
1	*0*	*0*			*V(&db)*
1	*1*	*0*	*...*		

Рис. 3.2. Ожидание писателя

Пусть одновременно начали работу два читателя (Ч1 и Ч2) и один писатель (П). Ч2 успевает первым получить доступ к счетчику rc, наращивает его. Ч1 ожидает доступа к rc. Ч2 получает доступ к БД, открывает доступ к счетчику. Ч1 разблокируется. Если в этот момент диспетчер включает писателя, он блокируется на доступе к БД. Ч1 получает доступ к счетчику, увеличивает его. Поскольку rc=2, блокировки на БД не происходит, хотя доступ к БД закрыт. Ч1 заканчивает чтение, уменьшает счетчик, однако БД не открывает, поскольку rc=1. Ч2 читает данные, уменьшает счетчик, открывает доступ к БД. Писатель продолжает работу. Если бы П активировался, когда оба читателя не дошли до блокировки базы, он бы начал работу, захватив db. Ч2 был бы заблокирован на db, Ч1 соответственно на mutex. После освобождения db писателем Ч2 продолжил бы работу, захватив доступ к БД, далее разблокировал бы mutex, и Ч1 также смог бы продолжить работу. Однако такое решение позволяет получить дос-

туп вновь прибывающим читателям, даже если в системе есть ожидающий писатель. В результате он может ожидать бесконечно.

Требуется иное решение. Если писатель ждет доступа, вновь прибывающие читатели не получают его, а становятся в очередь ожидания. Получив доступ, когда чтение закончат находящиеся в базе читатели, писатель откроет доступ к базе ожидающим читателям.

```
typedef int semaphore;
semaphore mutex = 1;   // контроль доступа к переменной rc
semaphore db=1;        // контроль доступа к базе данных
semaphore blkrd=1;// контроль доступа, если работает писатель
int rc=0;    // кол-во процессов читающих или желающих читать

void reader(void)      // процесс читатель
 {while(1)
  {P(&blkrd);          // если есть писатель, доступ закрыт
   P(&mutex);          // закрыть доступ к счетчику читателей
   rc++;               // увеличить его
   if (rc==1) P (&db); // если 1 процесс, закрыть доступ к БД
   V(&blkrd);
   V(&mutex);          // открыть доступ к счетчику
   read_database();    // чтение БД
   P(&mutex);          // закрыть доступ к счетчику
   rc=rc-1;            // уменьшить его
   if (rc==0) V(&db);  // если нет других читателей (процесс
                       // последний), открыть доступ к БД
   V(&mutex);          // открыть доступ к счетчику
   use_read_data();}}  // использование данных

void writer(void)      // процессы писатели
 {while(1)
  {make_data();        // подготовка данных к записи
   P(&blkrd);          // блокировка доступа читателям
   P(&db);             // закрыть доступ
   write_database();   // записать данные
   V(&blkrd);
   V(&db);}}           // открыть доступ
```

Пусть в системе три читателя и один писатель (рис. 3.3). Ч1 и Ч2 получают доступ к БД, диспетчер передает управление П1, который блокирует доступ новых читателей, и пытается получить доступ к базе. Поскольку она занята, он переходит в ожидание. Ч3 пытается получить доступ к счетчику, однако блокируется на мьютексе blkrd. Ч2 заканчивает работу с базой, но поскольку в работе остается Ч1, доступ к БД не изменяется. П1 по-прежнему блокирован и Ч3 также. Ч1 заканчивает работу с БД, и поскольку он последний из читателей, работающих с БД, он открывает доступ к БД. П1 получает доступ, производит запись, открывает доступ читателям. Ч3 проходит семафор blkrd и получает доступ к БД.

Однако если в системе несколько писателей, то первый получивший доступ блокирует остальных так же, как и читателей на blkrd (П2 на рис. 3.3). После освобождения этого мьютекса, решение, кто из процессов будет запущен на

выполнение, принимает планировщик. В зависимости от дисциплины диспетчеризации вместо писателя доступ может получить читатель. В результате до момента обслуживания писателя может быть обслужено произвольное число читателей, как и при предыдущем решении. Одним из вариантов решения проблемы служит предоставление процессам-писателям большего приоритета, чем читателям.

Mutex Db Blkrd Rc	Читатель 1	Читатель 2	Читатель 3	Писатель 1	Писатель 2
1 1 1 0 *1 0 1 2*	*...* *KC*	*KC*			
		...			
1 0 1 2 *1 0 0 2*				*P(&blkrd)* *P(&db) // ###*	
1 -1 0 2			*P(&blkrd) // ###*		
1 -1 -1 2					*P(&blkrd)// ###*
0 -1 -2 0 *0 -1 -2 0*	*...* *if(rc==0) // (true)* *V(&db)*			*=>*	
0 0 -2 0 *0 0 -2 0*			*=>*	*KC* *V(&blkrd)*	*// ??? =>*
0 0 -1 0			*P(&mutex) ...*		

Рис. 3.3. Вариант организации взаимодействия

Другое решение позволяет получить доступ вначале всем ожидающим писателям, только после этого читатели будут допущены к базе. Если доступ к базе имеют читатели, а писатель ожидает его, то вновь прибывшие читатели будут обслуживаться после этого писателя:

```
typedef int semaphore;
semaphore mutex = 1;   // контроль доступа к переменной rc
int rc=0;    // кол-во процессов читающих или желающих читать
semaphore wmutex = 1;  // контроль доступа к переменной wc
int wc=0;    // кол-во процессов пишущих или желающих писать
semaphore db=1;        // контроль доступа к базе данных
semaphore blkrd=1;     // контроль доступа если работает писатель

void reader(void)      // процесс читатель
{while(1)
 {P(&mutex);           // закрыть доступ к счетчику читателей
   P(&blkrd);          // если есть писатель, доступ закрыт
   rc++;               // увеличить счетчик читателей
   if (rc==1) P (&db); // если 1 процесс, закрыть доступ к БД
   V(&blkrd);
   V(&mutex);          // открыть доступ к счетчику
   read_database();    // чтение БД
   P(&mutex);          // закрыть доступ к счетчику
   rc=rc--;            // уменьшить его
   if (rc==0) V(&db);  // если нет других читателей (процесс
```

```
                              // последний), открыть доступ к БД
    V(&mutex);                // открыть доступ к счетчику
    use_read_data();}}        // использование данных
  void writer(void)           // процессы писатели
   {while(1)
    {make_data();             // подготовка данных к записи
     P(&wmutex);              // блокировка доступа к счетчику
     wc++;                    //
     if(wc==1) P(&blkrd);// если писатель вошел в систему, новым
                              // читателям доступ закрыт
     V(&wmutex);
     P(&db);                  // закрыть доступ
     write_database();        // записать данные
     V(&db);                  // открыть доступ
     P(&wmutex);              // доступ к счетчику
     wc--;
     if(wc==0) V(&blkrd);//если больше писателей нет, открыть
                              // доступ читателям
     V(&wmutex); }}           // открыть доступ
```

Mutex	Db	Blkrd	Rc	Wmutex	Wc	Читатель 1	Чит. 2	Читатель 3	Писатель 1	Писатель 2
1	*1*	*1*	*0*	*1*	*0*	*...*				
1	*0*	*1*	*2*	*1*	*0*	*KC*	*KC*			
							...			
1	*0*	*1*	*2*	*1*	*0*				*P(&wmutex)*	
1	*0*	*1*	*2*	*0*	*0*				*wc++*	
1	*0*	*1*	*2*	*0*	*1*				*if (wc==1)// true*	
1	*0*	*1*	*2*	*0*	*1*				*P(&blkrd)*	
1	*0*	*0*	*2*	*0*	*1*				*V(&wmutex)*	
1	*0*	*0*	*2*	*1*	*1*				*P(&db) // ###*	
1	*-1*	*0*	*2*	*1*	*1*			*P(&mutex)*		
0	*-1*	*0*	*2*	*1*	*1*			*P(&blkrd) //###*		
0	*-1*	*-1*	*2*	*1*	*1*					*P(&wmutex)*
0	*-1*	*-1*	*2*	*0*	*1*					*wc++*
0	*-1*	*-1*	*2*	*0*	*2*					*if (wc==1) // false*
0	*-1*	*-1*	*2*	*0*	*2*					*V(&wmutex)*
0	*-1*	*-1*	*2*	*1*	*2*					*P(&db) // ###*
						...				
0	*-2*	*-1*	*0*	*1*	*2*	*if(rc==0)//true*				
0	*-2*	*-1*	*0*	*1*	*2*	*V(&db)*			*=>*	
0	*-1*	*-1*	*0*	*1*	*2*				*KC*	
0	*-1*	*-1*	*0*	*1*	*2*				*V(&db)*	*=>*
0	*0*	*-1*	*0*	*1*	*2*					*KC*
										...
0	*1*	*-1*	*0*	*0*	*0*				*if (wc==0)// true*	
0	*1*	*-1*	*0*	*0*	*0*			*=>*	*V(&blkrd)*	
0	*1*	*0*	*0*	*0*	*0*			*rc++*		
0	*1*	*0*	*1*	*0*	*0*	*...*				

Рис. 3.4. Корректное решение задачи

Оба приведенных решения несколько снижают производительность. В результате возможны многократные переключения процессов при прохождении фрагмента кода из серии семафоров. Несмотря на это, последний вариант обеспечивает наиболее корректное решение задачи.

Следует заметить, что в современных ОС существует поддержка выполнения нескольких операций над набором объектов синхронизации неделимым образом. Это упрощает взаимодействие, однако может сделать ряд алгоритмов, основанных на последовательных операциях над несколькими семафорами, неработоспособными.

3.3. Мониторы

Использование семафоров тоже имеет свои ограничения. Если семафоры в листинге поменять местами, то может произойти взаимоблокировка, т.е. программы не застрахованы от случайных ошибок программиста. Для организации взаимодействия в 1974 г. Хоар и Хансен предложили примитив синхронизации более высокого уровня – **монитор**. Монитор – это набор процедур, переменных и других структур данных, объединенных в особый модуль или пакет с целью организации работы с каким-либо критическим ресурсом однотипным образом и обеспечения взаимного исключения.

Например, какой-либо ресурс должен разделяться между процессами. Соответственно для получения ресурса процесс должен обратиться к некоторому планировщику, который с помощью внутренних переменных отслеживает, занят ресурс или свободен, и выполняет последующие необходимые действия. Процедуры этого планировщика доступны всем процессам, однако планировщик может обслуживать только один процесс в один момент времени. Такой планировщик и является монитором.

Процессы могут вызывать процедуры монитора, однако у внешних процедур нет прямого доступа к внутренним структурам монитора. Важным свойством монитора является следующее: при обращении к монитору в любой момент времени активным может быть только один процесс. Мониторы являются структурным компонентом языка программирования, и компилятор знает, что вызовы процедур монитора следует кодировать иначе, чем вызовы других процедур.

Для реализации взаимного исключения обычно используются мьютекс или бинарный семафор. Однако этого еще недостаточно. Как правило, первые несколько команд процедуры монитора проверяют, есть ли уже в мониторе выполняющийся процесс. Если да, то вызывающий процесс должен перейти к ожиданию. Необходим способ блокировки процессов, которые не могут продолжаться далее. Решение в том, чтобы ввести переменные состояния и две операции wait и signal. Если процедура монитора обнаруживает, что работу нельзя продолжать, то она выполняет операцию wait на какой-либо переменной

состояния. Это приводит к блокировке вызывающего процесса и позволяет другому процессу войти в монитор. Другой процесс может активизировать заблокированный, выполнив операцию signal на той переменной состояния, на которой процесс был заблокирован.

Далее необходимо сделать так, чтобы в мониторе не было двух процессов одновременно. Хоар предложил запуск разбуженного процесса и остановку второго. Хансен предложил, что процесс, выполнивший signal, должен немедленно покинуть монитор, т.е. операция signal должна выполняться в самом конце монитора. Если операция signal выполнена на переменной, с которой связаны несколько заблокированных процессов, планировщик выбирает только один из них. Существует и третье решение: позволить процессу, выполнившему signal, продолжать работу и запустить ожидающий процесс только после того, как первый процесс покинет монитор.

Переменные состояния не являются счетчиками. Они не накапливают значения. Это значит, что в случае выполнения операции signal на переменной состояния, с которой не связано ни одного блокированного процесса, сигнал будет утерян, т.е. операция wait должна выполняться раньше, чем signal.

Механизм мониторов поддерживается Java. Ключевое слово **synchronized** гарантирует, что если хотя бы один поток начал выполнение этого метода, ни один другой поток не сможет выполнять другой синхронизированный метод данного класса, т.е. идет поддержка механизма мониторов на уровне языка, что отсутствует в C. Решение задачи «производитель-потребитель» с помощью мониторов на Java (приводится по [13]) выглядит следующим образом:

```
public class ProducerConsumer
{static final int N=100;              // размер буфера
 static producer p=new producer();    // поток производитель
 static consumer c=new consumer();    // поток потребитель
 static pc_monitor mon=new pc_monitor();// монитора

public static void main (String args[])
{p.start();                  // запуск потоков производителя
 c.start();}                 // и потребителя

static class producer extends Thread  // класс производителя
{public void run()           // процедура потока
  {int item;
   while (true)
    {item=produce_item();    // произвести элемент
     mon.insert(item);}}     // добавить его в буфер

private int produce_item(){…}//получение элемента
}

static class consumer extends Thread  // класс потребителя
{public void run()           // процедура потока
 int item;
 while (true)
  {item = mon.remove();      // взять элемент из буфера
   consume_item(item);}}     // потребить элемент
```

```
private void comsume_item(int item) {…}// использование
}                              // элемента

static class pc_monitor                 // класс монитора
{private int buffer[]=new int[N];       // содержит буфер
 private int count=0, lo=0, hi=0;       // счетчик и индексы

public synchronized void insert (int val)// добавление
                               // элемента в буфер
{if (count==N) go_to_sleep();    // если буфер полон, ожидание
 buffer[hi]=val;                 // поместить элемент
 hi=(hi+1)%N;                    // увеличить индекс для записи
 count=count+1;                  // число элементов в буфере
 if (count==1) notify();}        // если буфер был пуст,
                                 // разбудить процесс
public synchronized int remove() // процедура взятия
                                 // объекта из буфера
{int val;
 if (count==0) go_to_sleep();// если буфер пуст, ожидание
 val=buffer[lo];             // взятие элемента
 lo=(lo+1)%N;                // увеличение индекса для взятия
 count=count-1;              // число элементов в буфере
 if (count==N-1) notify();   // если буфер был полон,
                             // разбудить процесс
 return val;}

private void go_to_sleep()  // процедура ожидания
{try
{wait();}                    // ожидание с обработкой исключения
  catch (InterruptedException exc) {};}}}
```

Внешний класс ProducerConsumer создает и запускает два потока. Класс монитора содержит два синхронизированных потока, используемых для текущего помещения элементов в буфер и извлечения их оттуда. Поскольку эти методы синхронизированы, то состояние состязания исключается автоматически средствами языка программирования. Переменная count отслеживает количество элементов в буфере, lo – содержит индекс, откуда надо извлекать элемент, hi – куда помещать. Если lo = hi, то в буфере или нет элементов, или он заполнен полностью, что можно отследить с помощью count. Синхронизированные методы в Java несколько отличаются от классических мониторов тем, что у них отсутствует переменная состояния. Взамен предлагаются аналогичные процедуры wait и notify.

Доступ к разделяемым переменным всегда ограничен телом монитора, что автоматически исключает критические интервалы. Монитор является пассивным объектом в том смысле, что это не процесс, его процедуры выполняются только по требованию процессов. Мониторы обладают следующими свойствами:

- высокая гибкость при реализации синхронизирующих операций;

- локализация разделяемых переменных в теле монитора позволяет вынести специфические синхронизирующие конструкции из параллельных процессов;
- процессы имеют возможность совместно использовать модули, имеющие критические секции;
- если несколько процессов разделяют некоторый ресурс и работают с ним совершенно одинаково, то в мониторе для этого требуется только одна процедура. При других решениях соответствующий код должен быть в теле всех процессов.

3.4. Сообщения

Выше уже упоминалось, что помимо синхронизации действий ОС предоставляет процессам механизмы для обмена данными. К самым распространенным их них относятся сообщения и каналы (pipe). Однако это отдельная большая тема, выходящая за рамки данного пособия. Тем не менее, поскольку в подобных механизмах используется неявная (скрытая) синхронизация, авторы сочли возможным кратко остановиться на данных механизмах и рассмотреть их возможности в качестве механизма синхронизации.

Процессы в ОС могут обмениваться сообщениями. Для хранения посланного, но еще не полученного сообщения необходимо место, называемое почтовым ящиком, или буфером сообщения. Если процесс хочет общаться с другим процессом, то он просит систему выделить ему почтовый ящик, который свяжет эти два процесса. Для отправления сообщения процесс просто помещает его в почтовый ящик, откуда второй процесс может взять его в любое время. Второй процесс должен знать о существовании ящика, и для получения сообщения выполнить к нему обращение. Если объем данных велик, целесообразно не помещать их в ящик, а оставлять всего лишь информацию, где их можно найти. Если почтовый ящик не связан жестко с конкретными процессами, то сообщение должно содержать идентификаторы и процесса-отправителя, и процесса-получателя.

Почтовый ящик состоит из заголовка, где содержится информация о ящике, и из нескольких буферов (ячеек) для сообщений. В простейшем случае сообщения передаются только в одном направлении. Процесс может посылать сообщения, пока есть свободные ячейки. Если они все заполнены, то процесс может или ожидать, или выполнять другие операции. Аналогично и процесс-получатель может получать сообщения, пока есть заполненные ячейки. Можно организовать более сложные ящики. Например, двунаправленные. Такой ящик позволяет подтверждать прием сообщений. Чтобы гарантировать доставку подтверждения в случае, когда все ячейки заняты, подтверждение помещается туда же, где лежало исходное сообщение. В эту ячейку не может быть помещено новое сообщение до тех пор, пока не будет получено подтверждение.

Как правило, используется буфер из определенного количества элементов, тип которых задается при создании ящика. Для реализации механизма дос-

таточно двух примитивов: **send** – отправить и **receive** – принять. Примитивы имеют два параметра, один из которых – собственно сообщение или его адрес, второй параметр указывает или идентификатор взаимодействующего процесса, или идентификатор почтового ящика.

Еще один вариант организации сообщений – не использовать буферизацию. В этом случае, если send выполняется раньше receive, посылающий процесс блокируется до выполнения receive, когда сообщение может быть напрямую скопировано от производителя к потребителю без промежуточной буферизации и наоборот. Этот метод называется **рандеву**, он легче реализуется, чем схема с буферизацией, однако взаимодействующие процессы должны быть жестко синхронизированы.

Примитивы send и receive имеют скрытый механизм взаимоисключения, а в большинстве систем – и блокировки при чтении из пустого ящика или записи в заполненный. Однако несмотря на простоту использования, это решение менее производительно.

Рассмотрим решение задачи производитель-потребитель с помощью сообщений:

```
#define N 100                  // ячеек в буфере
void producer (void)           // производитель
{int item;
 message m;
 while (1)
  {item=produce_item();        // создать элемент
   receive(consumer, &m);      // получить пустое сообщение
   build_message(&m, item);    // сообщение для отправки
   send(consumer, &m);}}       // послать сообщение

void consumer (void)           // потребитель
{int item, i;
 message m;
 for (i=0; i<N; i++) send(producer, &m);
                               // послать серию пустых сообщений
 while (1)
  {receive(producer, &m);      // получить сообщение
   item=extract_item(&m);      // извлечь элемент из сообщения
   send(producer,&m);          // послать пустое сообщение
   consume_item(item);}}       // использовать элемент
```

Здесь потребитель сначала отправляет N пустых сообщений производителю – по одному на каждую свободную ячейку буфера. Производитель, получив сообщение о наличии пустой ячейки, помещает элемент в буфер, отправляя сообщение потребителю. Тот, в свою очередь получив сообщение, извлекает объект, и вновь отправляет сообщение производителю о наличии свободной ячейки.

При использовании сообщений возникает ряд проблем, аналогичных при передаче пакетов в сети: сообщение может потеряться, что требует использования механизма подтверждения, а если подтверждения не было, требуется по-

вторная отправка пакета. Подтверждение также может потеряться, следовательно, необходим идентификатор, позволяющий отличать копии от оригинала и т.д.

3.5. Барьеры

Этот механизм синхронизации предназначен для группы процессов. Некоторые приложения делятся на фазы, и существует правило, что процесс не может перейти к следующей фазе, пока к этому не готовы все остальные процессы. Для этого в конце каждой фазы располагается барьер. Процесс, доходя до барьера, блокируется, пока все процессы не дойдут до него.

Так, на рис. 3.5 процесс D через какое-то время выполняет оставшиеся вычисления, и запускает примитив barrier, являющийся обычно библиотечной функцией. Поскольку только один процесс у барьера, он переходит в ожидание. Затем аналогично процесс B, потом C и, наконец, A. Как только последний из процессов выполнил вызов, блокировка снимается для всех ожидающих процессов, и они переходят за барьер.

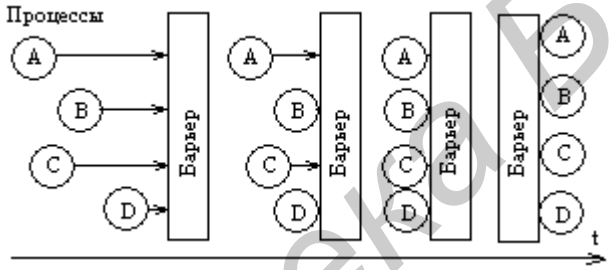

Рис. 3.5. Барьеры

Следует отметить, что многие механизмы синхронизации позволяют успешно имитировать друг друга. Рассмотрим, как с помощью сообщений можно организовать барьер.

Пусть в системе те же 4 процесса A, B, C, D и каждому из них известны уникальные идентификаторы оставшихся трех процессов; для простоты будем считать, что эти ID для процессов равны 0, 1, 2, 3 соответственно:

```
#define PROC_CNT 4          // число процессов

void main (void)
{int i;
 …                          // какие-то действия до барьера
 for(i=0; i<PROC_CNT; i++)  // каждому из процессов
   send(i);                 // посылается сообщение
 for(i=0; i<PROC_CNT; i++)  // от каждого из процессов
   receive(i);              // принимается сообщение
  …}                        // действия после барьера
```

4. Поддержка в современных ОС

4.1. Объекты синхронизации в Windows 2000/XP/Server 2003

В Windows поддерживаются следующие объекты синхронизации: критические секции, мьютексы, семафоры и события. На уровне ядра доступны и другие механизмы, в том числе для организации взаимодействия в многопроцессорных системах, но обсуждение этих возможностей выходит за рамки данного пособия.

Критические секции. Объекты этого типа создаются и удаляются функциями

void **InitializeCriticalSection** (LPCRITICAL_SECTION lpCriticalSection)
void **DeleteCriticalSection** (LPCRITICAL_SECTION lpCriticalSection)

В качестве параметра передается указатель на переменную типа `CRITICAL_SECTION`. Эти объекты не имеют дескрипторов и соответственно не могут использоваться совместно разными процессами, только потоками одного и того же процесса. При входе и выходе из критического участка кода поток вызывает соответственно функции

void **EnterCriticalSection** (LPCRITICAL_SECTION lpCriticalSection)
void **LeaveCriticalSection** (LPCRITICAL_SECTION lpCriticalSection)

Внутри области кода, защищенной одним и тем же объектом КС, может находиться только один поток. Если другой в это время вызывает EnterCriticalSection, он блокируется. Объекты КС не являются объектами ядра и поддерживаются в пространстве пользователя, что позволяет увеличить производительность. Это простейший механизм синхронизации потоков одного и того же процесса.

Мьютексы. В отличие от объектов КС мьютексы могут иметь имена и дескрипторы, реализуются в пространстве ядра, и их можно использовать для синхронизации потоков разных процессов. Дополнительно мьютексы позволяют задавать конечный период ожидания в состоянии блокировки. Мьютекс создается функцией

HANDLE **CreateMutex** (LPSECURITY_ATTRIBUTES lpsa,
BOOL bInitialOwner, LPCTSTR lpName)

Здесь `lpsa` – указатель на структуру атрибутов защиты, определяет возможность наследования указателя на объект мьютекса дочерними процессами. Если `NULL` – не наследуется. `BInitialOwner` определяет, требуется ли захват мьютекса сразу после создания. `TRUE` – да, `FALSE` – нет. `LpName` – указатель на нуль-строку, содержащую имя создаваемого объекта. Функция возвращает указатель типа `HANDLE` на созданный объект. Если создание не удалось, возвращается `NULL`. Созданный мьютекс имеет два состояния – сигнальное и не-

сигнальное. В сигнальном состоянии он может быть захвачен, в противном случае при попытке захвата поток блокируется. Для получения указателя на уже имеющийся объект используется функция

HANDLE **OpenMutex** (DWORD dwDesiredAccess, BOOL bInheritHandle, LPCTSTR lpName)

Первый параметр определяет режим использования. `MUTEX_ALL_ACCESS` определяет разрешение всех возможных вариантов работы с объектом. Второй параметр определяет возможность наследования указателя на объект мьютекса. Третий – указатель на имя. Для удаления ссылки на мьютекс используется функция

BOOL **CloseHandle** (HANDLE hObject)

Ссылка удаляется и автоматически при закрытии потока. Мьютекс уничтожается системой автоматически, если с ним не связано ни одного указателя. Для захвата мьютекса используется одна из функций ожидания сигнального состояния. Если объект находится в этом состоянии, происходит захват, и он переводится в несигнальное состояние. Другой поток будет вынужден ждать. Рассмотрим некоторые функции этого множества:

DWORD **WaitForSingleObject** (HANDLE hHandle,
DWORD dwMilliseconds)

Здесь `hHandle` – указатель на объект, сигнальное состояние которого ожидается, второй параметр – максимально допустимое время ожидания в миллисекундах. Если интервал задан 0 – тестируется текущее состояние и происходит немедленный возврат. Если указано `INFINITE` – интервал тайм-аута не используется. В случае ошибки возвращается значение `WAIT_FAILED`. Иначе – `WAIT_ABANDONED`, если поток, который создал мьютекс, уже завершился без освобождения объекта с помощью Release. Вызвавший функцию процесс становится владельцем, а мьютекс переходит в несигнальное состояние. Возвращается `WAIT_OBJECT_0`, если состояние объекта сигнальное, и `WAIT_TIMEOUT`, если завершился тайм-аут. Функция позволяет работать не только с мьютексами, но и с другими объектами, в том числе семафорами, событиями, процессами, потоками, таймерами и др.

DWORD **WaitForMultipleObjects** (
DWORD nCount, число указателей на объекты
CONST HANDLE *lpHandles,массив указателей
BOOL bWaitAll, если TRUE, то ждет всех объектов в сигнальном состоянии, FALSE – любого из них
DWORD dwMilliseconds) тайм-аут

Возвращаемые значения аналогичны, однако если `bWaitAll=FALSE`, то возвращаемое значение минус `WAIT_OBJECT_0` (либо `WAIT_ABANDONED`)

указывает номер указателя в массиве на тот объект, который находится в сигнальном состоянии. В случае если требуется ждать сигнального состояния всех объектов, то операция выполняется атомарным образом, т.е. значения объектов, находящихся в сигнальном состоянии, не изменяются до тех пор, пока есть объекты, находящиеся в несигнальном состоянии.

DWORD **MsgWaitForMultipleObjects**(DWORD nCount,
LPHANDLE pHandles, BOOL fWaitAll, DWORD dwMilliseconds,
DWORD dwWakeMask) тип сообщений в очереди, по которым ожидание также должно закончиться

Функция аналогична, но позволяет дополнительно обрабатывать сообщения из очереди, например для обработки клавиатуры или мыши. Параметры идентичны. `WakeMask` представляет собой комбинацию следующих значений:

- `QS_ALLINPUT` – любое сообщение в очереди;
- `QS_HOTKEY` – сообщение `WM_HOTKEY`;
- `QS_INPUT` – сообщение ввода;
- `QS_KEY` – сообщения `WM_KEYUP, WM_KEYDOWN, WM_SYSKEYUP, WM_SYSKEYDOWN`;
- `QS_MOUSE` – сообщения `WM_MOUSEMOVE` или нажатия кнопок мыши (`WM_LBUTTONUP, WM_RBUTTONDOWN`, и др.);
- `QS_MOUSEBUTTON` – нажатие кнопок мыши (`WM_LBUTTONUP, WM_RBUTTONDOWN`, и др.);
- `QS_MOUSEMOVE` – сообщение `WM_MOUSEMOVE`;
- `QS_PAINT` – сообщение `WM_PAINT`;
- `QS_POSTMESSAGE` – сообщение иное, чем в данном списке;
- `QS_SENDMESSAGE` – сообщение, присланное другим потоком или приложением;
- `QS_TIMER` – сообщение `WM_TIMER`.

Возвращаемые значения аналогичны предыдущему случаю, дополнительно `WAIT_OBJECT_0+nCount` указывает, что появилось сообщение в очереди.

DWORD **SignalObjectAndWait**(
HANDLE hObjectToSignal, объект, устанавливаемый в сигнальное состояние
HANDLE hObjectToWaitOn, объект, сигнальное состояние которого ожидается
DWORD dwMilliseconds, тайм-аут
BOOL bAlertaible)

Эта функция позволяет атомарным образом один объект установить в сигнальное состояние, после чего дождаться сигнального состояния второго объекта. При создании потока с ним связывается очередь асинхронных вызовов процедур (APC queue), в которую ОС либо пользователь могут помещать запросы на выполнение функций в контексте данного потока, причем эти запросы не могут быть обслужены немедленно, поскольку поток может быть занят. Параметр `bAlertaible` указывает, заканчивать ли ожидание, если пришел запрос

на асинхронный вызов процедуры. Возвращаемые значения аналогичны WaitForSingleObject.

BOOL **ReleaseMutex** (HANDLE hMutex)

Освобождает захваченный мьютекс. Передается указатель на объект, возвращает `TRUE`, если успешно, `FALSE`, если ложно.

Семафоры. Механизм семафоров поддерживает счетчики, если значение счетчика больше 0, он в сигнальном состоянии. Отрицательные значения не допускаются. Создание семафоров производится функцией

```
HANDLE CreateSemaphore (
LPSECURITY_ATTRIBUTES lpSemaphoreAttributes,   права наследования
LONG lInitialCount,            начальное значение
LONG lMaximumCount, максимально допустимое значение
LPCTSTR lpName)                указатель на имя
```

Удаление ссылки производится уже рассмотренной функцией CloseHandle или автоматически при завершении потока. Объект автоматически удаляется, если нет связанных с ним указателей. Получение указателя на существующий объект выполняется функцией

```
HANDLE OpenSemaphore (DWORD dwDesiredAccess,
BOOL bInheritHandle, LPCTSTR lpName)
```

Параметры полностью идентичны функции OpenMutex. Ожидание сигнального состояния производится с помощью рассмотренных WaitForSingleObject, WaitForMultipleObjects и др. Снятие захвата (операция V) производится функцией

```
BOOL ReleaseSemaphore (
HANDLE hSemaphore,             указатель на объект
LONG lReleaseCount,            величина, на которую надо увеличить
                               значение семафора, больше 0
LPLONG lpPreviousCount)        указатель на переменную, куда будет
                               возвращено предыдущее значение семафора
```

События. Объект события позволяет сигнализировать другим процессам о наступлении какого-либо события. Существуют сбрасываемые программно (вручную) события, когда сигнал может быть передан одновременно всем потокам, и сбрасываемые автоматически после освобождения одного из потоков. Создание объекта выполняется функцией

```
HANDLE CreateEvent (
LPSECURITY_ATTRIBUTES lpEventAttributes, атрибут наследования
BOOL bManualReset,             TRUE для ручного сброса
BOOL bInitialState,            начальное состояние, TRUE – сигнальное
LPCTSTR lpName)                указатель на имя
```

Ручной сброс выполняется функцией BOOL **ResetEvent** (HANDLE hEvent). **OpenEvent** аналогично уже рассмотренным вариантам возвращает указатель на существующий объект события. BOOL **SetEvent** (HANDLE hEvent) устанавливает событие в сигнальное состояние. BOOL **PulseEvent** (HANDLE hEvent) освобождает все потоки, ожидающие сбрасываемого вручную события, и событие сразу же сбрасывается.

Разделяемая память. Для работы с разделяемой памятью используются функции **OpenFileMapping, MapViewOfFile, UnmapViewOfFile, CloseHandle**. Разделяемая память организуется либо через отображение на память файла, либо непосредственно в памяти. Далее этот объект отображается на адресное пространство потока, в результате с файлом можно работать, как с обычным массивом данных в памяти.

Почтовые ящики (буферы сообщений). В Windows также поддерживается модель буферов сообщений (почтовых ящиков). Они являются однонаправленными. Сервер (считывающий процесс) создает дескриптор почтового ящика функцией **CreateMailSlot**, далее он ожидает сообщения с помощью **ReadFile**. Клиент (отправляющий процесс) открывает ящик с помощью **CreateFile** и передает сообщение с помощью **WriteFile**. Если ни один сервер не ожидает сообщения, открытие ящика завершится ошибкой. Сообщение может быть прочитано всеми серверами.

4.2. Решение задачи «производитель-потребитель» в ОС Windows

Рассмотрим решение задачи «производитель-потребитель» с помощью семафоров на базе платформы Windows.

```
// producer.c
#include <windows.h>
#include <stdio.h>
#include <conio.h>
#define BUF_CNT 6           // размер буфера с данными
#define MUTEX "produser_mutex"// имя объекта-мьютекса
#define SEMEMPTY "produser_empty"// семафор-счетчик пустых
#define SEMFULL "produser_full"  // семафор-счетчик полных
#define SHAREDBUF "produser_buffer"   // имя общего буфера
#define IMUTEX 1            // индексы в массиве дескрипторов
#define IEMPTY 0            // для мьютекса, счетчика пустых и
#define IFULL 2             // полных элементов в буфере

void main(void)
{char c;
 HANDLE sem[3];   // указатели на дескрипторы
 HANDLE vbuffer;  // буфер в разделяемой памяти
 LPVOID buffer;   // его отображение на пространство процесса
 LONG pos;        // номер позиции для записи

// создание мьютекса или получение указателя на уже созданный
 if ((sem[IMUTEX]=CreateMutex(NULL,FALSE,MUTEX))!=NULL)
// создание семафоров-счетчиков или получение указателей на них
```

```
  {if((sem[IEMPTY]=CreateSemaphore(NULL,BUF_CNT,BUF_CNT,
                                                    SEMEMPTY))!=NULL)
  {if((sem[IFULL]=CreateSemaphore(NULL,0,BUF_CNT,SEMFULL))!=NULL)
// создание разделяемой области или получение указателя
  {if((vbuffer=CreateFileMapping((HANDLE)0xFFFFFFFF,NULL,
                       PAGE_READWRITE,0,BUF_CNT,SHAREDBUF))!=NULL)
// отображение разделяемой памяти на адресное прост-во процесса
  {if((buffer=MapViewOfFile(vbuffer,FILE_MAP_ALL_ACCESS,0,0,
                                                    BUF_CNT))!=NULL)
  {printf("produser: enter symbol from keyboard. Q - quit\n");
   printf("and press any key after enter in CS for continue.\n");
   while (1)
// создание данных
    {c=getch();
     if (c=='Q') break;
// попытка записи в буфер
     printf("%c: Wait on empty sem & CS_mutex...",c);
     if (WaitForMultipleObjects(2,&sem[IEMPTY],TRUE,
                                     INFINITE)==WAIT_FAILED) break;
// критическая область
     printf("OK! Enter...");
     getch();
// увеличение семафора полных ячеек для получения номера записи
     if(ReleaseSemaphore(sem[IFULL],1,&pos)==0) break;
// поместить элемент в буфер
     *((char*)buffer+pos=c;
     printf("OK! Writing. \n");
// выход из КС, освобождение мьютекса
     if(ReleaseMutex(sem[IMUTEX])==0) break;
}
// отключение отображения общей памяти на пространство процесса
   UnmapViewOfFile(buffer);}
// уменьшение числа ссылок на соответствующий системный объект
   CloseHandle(vbuffer);}
  CloseHandle(sem[IFULL]);}
 CloseHandle(sem[IEMPTY]);}
 CloseHandle(sem[IMUTEX]);}
}
```

```
// consumer.c
#include <windows.h>
#include <stdio.h>
#include <conio.h>
#define BUF_CNT 6                // размер буфера с данными
#define MUTEX "produser_mutex"   // имя объекта-мьютекса
#define SEMEMPTY "produser_empty"// семафор-счетчик пустых
#define SEMFULL "produser_full"  // семафор-счетчик полных
#define SHAREDBUF "produser_buffer"  // имя общего буфера
#define IMUTEX 1            // индексы в массиве дескрипторов
#define IEMPTY 0            // для мьютекса, счетчика пустых и
#define IFULL 2             // полных элементов в буфере

void main(void)
{char c;
```

```
 HANDLE sem[3];   // указатели на дескрипторы
 HANDLE vbuffer;  // буфер в разделяемой памяти
 LPVOID buffer;   // отображение буфера на простр-во процесса
 int i;
// создание мьютекса или получение указателя на уже созданный
 if ((sem[IMUTEX]=CreateMutex(NULL,FALSE,MUTEX))!=NULL)
// создание семафоров-счетчиков или получение указателей на них
  {if((sem[IEMPTY]=CreateSemaphore(NULL,BUF_CNT,BUF_CNT,
                                                  SEMEMPTY))!=NULL)
   {if((sem[IFULL]=CreateSemaphore(NULL,0,BUF_CNT,SEMFULL))!=NULL)
// создание разделяемой области или получение указателя
    {if((vbuffer=CreateFileMapping((HANDLE)0xFFFFFFFF,NULL,
                       PAGE_READWRITE,0,BUF_CNT,SHAREDBUF))!=NULL)
// отображение разделяемой памяти на адресное прост-во процесса
    {if((buffer=MapViewOfFile(vbuffer,FILE_MAP_ALL_ACCESS,0,0,
                                                  BUF_CNT))!=NULL)
    {printf("consumer: press any key for read buffer. Q - quit\n");
    printf("and press any key after enter in CS for continue.\n");
    while (1)
     {c=getch();
      if (c=='Q') break;
// попытка чтения буфера
      printf("Wait on full_sem & CS_mutex...");
      if (WaitForMultipleObjects(2,&sem[IMUTEX],TRUE,
                                       INFINITE)==WAIT_FAILED) break;
// критическая область
      printf("OK! Enter...");
      c=getch();
      c=*((char*)buffer);
      for(i=0;i<BUF_CNT-1;i++)
        *((char*)buffer+i)=*((char*)buffer+i+1);
      printf("OK! Read symbol %c\n", c);
// выход из КС, освобождение мьютекса
      if(ReleaseMutex(sem[IMUTEX])==0) break;
// увеличение семафора-счетчика пустых ячеек
      if(ReleaseSemaphore(sem[IEMPTY],1,NULL)==0) break;
}
// отключение отображения общей памяти на пространство процесса
    UnmapViewOfFile(buffer);}
// уменьшение числа ссылок на соответствующий системный объект
    CloseHandle(vbuffer);}
   CloseHandle(sem[IFULL]);}
  CloseHandle(sem[IEMPTY]);}
 CloseHandle(sem[IMUTEX]);}
}
```

Используется буфер длиной BUF_CNT. Любой первый запущенный процесс создает требуемые объекты синхронизации и объекты разделяемой памяти для реализации общего буфера. Второй процесс получает дескрипторы уже созданных объектов. У потребителя при входе в КС используется мьютекс и семафор-счетчик заполненных ячеек, у производителя – мьютекс и семафор-счетчик пустых ячеек.

Чтобы можно было использовать функцию WaitForMultipleObjects вместо двух отдельных WaitForSingleObject, массив дескрипторов sem организуется в особом порядке – мьютекс в середине массива. Однако такие хитрости не всегда возможны, поэтому часто приходится использовать серию WaitForSingleObject либо на основе имеющихся дескрипторов строить новый массив, в который войдут только требуемые дескрипторы.

Далее, поскольку функции WaitFor… не позволяют возвращать значение семафора, требуется еще одно ухищрение. Вызов ReleaseSemaphore у производителя выполняется внутри критической секции, чтобы получить номер ячейки для записи. В данном варианте это возможно, т.к. функция WaitForMultipleObjects атомарна, и если производитель вошел в КС, то потребитель будет заблокирован на соответствующем вызове. Даже если буфер был пуст, т.е. потребитель ждал его заполнения, и производитель увеличивает семафор-счетчик внутри КС, потребитель по-прежнему будет заблокирован, поскольку производителем все еще захвачен мьютекс, т.е. гарантируется корректная работа. Однако и такое решение возможно далеко не во всех случаях.

4.3. Механизмы синхронизации UNIX на примере Alt Linux 2.4

Поддержка механизмов семафоров в UNIX. POSIX библиотека pthreads поддерживает мьютексы следующими функциями для создания, удаления, захвата и освобождения мьютекса: **pthread_mutex_init**, **pthread_mutex_destroy**, **pthread_mutex_lock**, **pthread_mutex_unlock**. В системах на основе System V, в том числе и различных дистрибутивах Linux реализована наиболее общая версия семафоров. Создание массива семафоров производится функцией

int **semget** (key_t key, int nsems, int semflg)

Здесь `key` – 32-разрядная переменная-ключ, идентифицирующая набор семафоров, nsems – число семафоров в наборе, `semflg` – режим доступа. Значение `IPC_CREAT` используется для создания нового набора, если он уже существует, проверяются права доступа. Если необходимо, чтобы возвращалась ошибка, если набор уже существует, должна использоваться совместно еще одна константа `IPC_EXCL`. Для подключения к уже существующему набору эти флаги не нужны. Младшие 9 бит флага отвечают за права доступа для root'а, владельца, группы. Возвращает идентификатор массива семафоров или –1 в случае ошибки. При создании набора создается одна общая для набора структура:

struct **semid_ds**{

struct ipc_rerm sem_perm; поля этой структуры содержат информацию о владельце, группе и правах доступа

struct sem* sem_base; указатель на массив объектов-семафоров

ushort sem_nsems; число семафоров в наборе

time_t sem_otime; время последней операции (P или V)

time_t sem_ctime; время последнего изменения
…}

При создании инициализируются поля структуры `sem_perm`, `sem_otime` устанавливается в 0, sem_ctime – в текущее время. Для каждого семафора создается структура следующего вида:

struct **sem**{
ushort semval; текущее значение семафора
pid_t sempid; ID процесса, вызвавшего последнюю операцию
ushort semncnt; кол-во процессов, ожидающих увеличения семафора
ushort semzcnt;} кол-во процессов, ожидающих нулевого семафора

Ключ `key` можно задавать явно, однако это не гарантирует уникальности ключа – такой уже может существовать в системе. Большую гарантию (однако не 100 %) дает использование функции

key_t **ftok**(char* pathname, int proj_id)

Первый параметр задает путь доступа и имя существующего файла, в качестве которого может выступать и корневой каталог («/»), и текущий каталог(«./»). Второй параметр – заданный пользователем ненулевой идентификатор. Хотя это переменная типа `int`, для генерации реально используется только 8 младших бит. Гарантируется уникальность ключа для отличающихся пар и идентичность для одинаковых. В случае ошибки возвращает –1, и ключ-идентификатор набора – в случае успеха.

Операции над семафорами в массиве выполняются вызовом

int **semop** (int semid, struct sembuf *sops, unsigned nsops)

Здесь `semid` – идентификатор набора, `sosp` – указатель на массив структур `sembuf`, определяющих, что выполняется с набором, `nsosp` – число элементов в этом массиве. В результате один вызов обрабатывает несколько семафоров. Ядро гарантирует атомарность этих операций, т.е. никакой другой процесс не начнет обработку над тем же набором, пока текущий процесс не завершит ее. Структура **sembuf** имеет следующий вид:

struct **sembuf** {
unsigned short sem_num;
short sem_op;
short sem_flg}

Здесь `sem_mun` – номер семафора в наборе. Если `sem_op = 0`, то происходит блокировка процесса до тех пор, пока семафор не станет равным нулю; если `sem_op>0`, то `sem_op` добавляется к текущему значению семафора, в результате может быть разбужен процесс, ожидающий увеличения семафора. Если `sem_op<0`, то происходит блокировка процесса до тех пор, пока значение семафора не станет равным или больше по абсолютному значению, чем

sem_op, после чего от текущего значения семафора вычитается sem_op. Если флаг sem_flg = IPC_NOWAIT, вместо блокировки происходит возврат с возвращением ошибки. Если процесс завершает работу, не освободив захваченный семафор, то ожидающий процесс будет заблокирован бесконечно. Чтобы этого не происходило, указывается флаг SEM_UNDO. В этом случае ядро запоминает произведенные процессом операции над семафором и автоматически прокручивает их в обратном порядке при завершении работы процесса. Из системы семафоры удаляются принудительно. В противном случае они будут существовать в системе, даже если ни один процесс с ними не связан. Однако это позволяет поддерживать семафоры, не зависящие от жизненного цикла процессов. Для управления объектами набора используется вызов

int **semctl** (int semid, int semnum, int cmd, …)

У вызова три или четыре параметра. Первый – идентификатор набора, второй – номер семафора в наборе, если операция выполняется над одним семафором набора, третий – команда, четвертый параметр arg – объединение следующего вида:

union **semun** {
int val;
struct semid_ds *buf;
unsigned short *array;
struct seminfo *__buf;}

Возможны следующие команды:

- IPC_STAT – копирует данные о наборе в структуру, определяемую arg.buf. Параметр semnum игнорируется;
- IPC_SET – устанавливает поля структуры semid_ds о пользователе, группе и режиме доступа из arg.buf. При этом модифицируется поле sem_ctime. Параметр semnum игнорируется;
- IPC_RMID – немедленно удаляет набор семафоров, активируя все заблокированные на нем процессы. Параметр semnum игнорируется;
- GETALL возвращает semval для всех семафоров набора в arg.array. Параметр semnum игнорируется;
- GETVAL возвращает semval для семафора semnum набора semid;
- GETNCNT возвращает значение semncnt (число процессов, ожидающих увеличения) для семафора semnum;
- GETZCNT возвращает значение semzcnt (число процессов, ожидающих 0) для семафора semnum;
- GETPID возвращает sempid (т.е. pid процесса, последним выполнявшим операцию) для семафора semnum;
- SETALL устанавливает semval для всех семафоров набора из arg.array. Модифицирует sem_ctime набора. Изменение влияет на

пробуждение ожидающих процессов. Параметр semnum игнорируется. Отрицательные значения arg.array[i] не допускаются;

- SETVAL устанавливает semval для семафора semnum. Модифицирует sem_ctime набора. Изменение влияет на пробуждение ожидающих процессов. Отрицательные значения arg.val не допускаются.

Для любой команды у процесса должны быть необходимые привилегии. Как правило, для команд, изменяющих информацию об объектах, требуются права суперпользователя, либо необходимо быть владельцем или создателем набора. В случае ошибки возвращает –1, 0 в случае успеха (за исключением команд GETNCNT, GETZCNT, GETVAL, GETPID).

Разделяемая память. UNIX также поддерживает разделяемые страницы памяти. Создание разделяемой области выполняется вызовом

int **shmget** (key_t key, int size, int shmflg)

Здесь size – размер задаваемой области, остальные параметры аналогичны рассмотренным для семафоров. Подключение области к виртуальному адресному пространству процесса производится вызовом

void * **shmat** (int shmid, const void *shmaddr, int shmflg)

Здесь shmid – идентификатор области, shmaddr – адрес виртуального пространства, куда необходимо произвести отображение; если указать NULL, то система вправе подключить область по любому адресу. Если флаг shmflg – SHM_RDONLY, то область подключается как «только для чтения». Возвращает адрес, на который происходит отображение. В случае ошибки возвращает –1. Отключение отображения в виртуальное адресное пространство процесса выполняет вызов

int **shmdt** (const void *shmaddr)

Здесь shmaddr – адрес отображения, возвращенный ранее shmat. Для управления соответствующими структурами используется вызов

int **shmctl** (int shmid, int cmd, struct shmid_ds *buf)

Здесь shmid – идентификатор области, cmd – команда, buf – указатель на структуру с информацией об области. Команда IPC_RMID позволяет освободить занятые структуры, при этом область удаляется лишь в том случае, если все отображения будут отключены процессами. При ошибке возвращает –1, при успешном завершении 0.

Сообщения. В UNIX для поддержки механизма сообщений используются следующие вызовы:

int **msgget**(key_t key, int msgflg) – для создания очереди сообщений, связанных с данным ключом;

int **msgsnd** (int msqid, struct msgbuf* msgp, size_t msgsz, int msgflg) – для отправки сообщения;

ssize_t **msgrcv** (int msqid, struct msgbuf *msgp, size_t msgsz, long msgtyp, int msgflg) – для получения сообщения;

int **msgctl** (int msqid, int cmd, struct msqid_ds *buf) – для управления структурами.

4.4. Решение задачи «производитель-потребитель» в ОС UNIX

Рассмотрим решение задачи «производитель-потребитель» на примере Alt Linux:

```
// produser.c
#include <sys/sem.h>
#include <sys/shm.h>
#include <stdio.h>

#define BUF_CNT 6 // длина буфера
#define SEM_CNT 3 // число семафоров в наборе
#define SEMKEY 1975    // ключ для набора семафоров
#define MEMKEY 1976    // ключ для разделяемой памяти
#define MUTEX    0     // индекс мьютекса в наборе
#define SEMEMPTY 1     // индекс семафора – счетчика пустых
#define SEMFULL  2     // индекс семафора – счетчика занятых
#define IDNAME "/"     // строковый идентификатор для ключа

int main(int argv, char* argc[])
{char c;
 int i;
 int ids;              // идентификатор набора семафоров
 int idm;              // идентификатор разделяемой памяти
 void *buf;            // указатель на отображенную область
 unsigned short seminit[SEM_CNT];
                       // инициализирующие значения семафоров
 struct semid_ds sem_i;     // информационная структура набора
 struct sembuf oper[2];     // массив для выполнения операций P и V
 key_t semkey, memkey; // системные ключи наб. семафоров и памяти

// создание набора семафоров
 if ((semkey=ftok(IDNAME,SEMKEY))!=-1)
 if ((ids=semget(semkey,SEM_CNT,IPC_CREAT|0x1f))!=-1)
// получить информацию о наборе
 {if (semctl(ids, 0, IPC_STAT, &sem_i)!=-1)
  {i=1;
// если это первый вход, то необходима инициализация
    if(sem_i.sem_otime==0)
// подготовка массива
     {seminit[MUTEX]=1;
      seminit[SEMEMPTY]=BUF_CNT;
      seminit[SEMFULL]=0;
// инициализация
      if (semctl(ids, 0, SETALL, seminit)==-1) i=0; }
    if (i==1)
// создание общей памяти
```

```
      {if ((memkey=ftok(IDNAME,MEMKEY))!=-1)
       if ((idm=shmget(memkey,BUF_CNT,IPC_CREAT|0x1f))!=-1)
// создание отображения в произвольную область
       {if((int)(buf=shmat(idm,NULL,SHM_RND))!=-1)
        {printf("produser: enter symbol from keyboard. Q - quit\n");
         printf("and press enter after enter in CS for continue.\n");
         while (1)
// рабочий цикл
          {c=getchar();
           if (c=='Q') break;
// создание структур для организации P() на наборе, уменьшение
// числа пустых ячеек и захват мьютекса
           printf("%c: Wait on empty_sem & mutex_CS...\n",c);
           oper[0].sem_num=SEMEMPTY;
           oper[0].sem_op=-1;
           oper[0].sem_flg=SEM_UNDO;
           oper[1].sem_num=MUTEX;
           oper[1].sem_op=-1;
           oper[1].sem_flg=SEM_UNDO;
           if(semop(ids, oper, 2)==-1) break;
// критическая область
           printf("OK! Enter...\n");
           getchar(); getchar();
// получить текущее значение семафора – счетчика заполненных
// ячеек – это адрес записи в разделяемую память
           if((i=semctl(ids, SEMFULL, GETVAL))==-1) break;
// запись
           *((char*)buf+i)=c;
           printf("OK! Writing.\n");
// создание структур для организации V() на наборе, увеличение
// числа полных ячеек и освобождение мьютекса
           oper[0].sem_num=MUTEX;
           oper[0].sem_op=1;
           oper[0].sem_flg=SEM_UNDO;
           oper[1].sem_num=SEMFULL;
           oper[1].sem_op=1;
           oper[1].sem_flg=SEM_UNDO;
           if(semop(ids, oper, 2)==-1) break;
          }
// отключение отображения общей памяти на пространство процесса
         shmdt(buf);}
// удаление разделяемой памяти
        shmctl(idm, IPC_RMID, NULL);}
       } }
// удаление набора семафоров
  semctl(ids,0,IPC_RMID);}
}

// consumer.c
#include <sys/sem.h>
#include <sys/shm.h>
#include <stdio.h>

#define BUF_CNT 6 // длина буфера
```

```
#define SEM_CNT 3 // число семафоров в наборе
#define SEMKEY 1975    // ключ для набора семафоров
#define MEMKEY 1976    // ключ для разделяемой памяти
#define MUTEX    0     // индекс мьютекса в наборе
#define SEMEMPTY 1     // индекс семафора – счетчика пустых
#define SEMFULL  2     // индекс семафора – счетчика занятых
#define IDNAME "/"     // строковый идентификатор для ключа

int main(int argv, char* argc[])
{char c;
 int i;
 int ids;         // идентификатор набора семафоров
 int idm;         // идентификатор разделяемой памяти
 void *buf;       // указатель на отображенную область
 unsigned short seminit[SEM_CNT];
                  // инициализирующие значения семафоров
 struct semid_ds sem_i;// информационная структура набора
 struct sembuf oper[2];// массив структур для операций P и V
 key_t semkey, memkey; // системные ключи наб. семафоров и памяти

// создание набора семафоров
 if ((semkey=ftok(IDNAME,SEMKEY))!=-1)
 if ((ids=semget(semkey,SEM_CNT,IPC_CREAT|0x1f))!=-1)
// получить информацию о наборе
  {if (semctl(ids, 0, IPC_STAT, &sem_i)!=-1)
   {i=1;
// если это первый вход, то необходима инициализация
    if(sem_i.sem_otime==0)
// подготовка массива
     {seminit[MUTEX]=1;
      seminit[SEMEMPTY]=BUF_CNT;
      seminit[SEMFULL]=0;
// инициализация
      if (semctl(ids, 0, SETALL, seminit)==-1) i=0; }
    if (i==1)
// создание общей памяти
     {if ((memkey=ftok(IDNAME,MEMKEY))!=-1)
      if ((idm=shmget(memkey,BUF_CNT,IPC_CREAT|0x1f))!=-1)
// создание отображения в произвольную область
      {if((int)(buf=shmat(idm,NULL,SHM_RND))!=-1)
       {printf("consumer: press enter for read buffer. Q - quit\n ");
        printf("and press enter after enter in CS for continue.\n");
        while (1)
// рабочий цикл
          {c=getchar();
          if (c=='Q') break;
// создание структур для организации P() на наборе, уменьшение
// числа полных ячеек и захват мьютекса
         printf("Wait on full_sem & mutex_CS...\n");
         oper[0].sem_num=SEMFULL;
         oper[0].sem_op=-1;
         oper[0].sem_flg=SEM_UNDO;
         oper[1].sem_num=MUTEX;
         oper[1].sem_op=-1;
```

```
          oper[1].sem_flg=SEM_UNDO;
          if(semop(ids, oper, 2)==-1) break;
// критическая область
          printf("OK! Enter...\n");
          getchar();
// чтение
          c=*((char*)buf);
          for(i=1;i<BUF_CNT;i++)
            *((char*)buf+i-1)=*((char*)buf+i);
          printf("OK! Read symbol %c\n", c);
// создание структур для организации V() на наборе, увеличение
// числа пустых ячеек и освобождение мьютекса
          oper[0].sem_num=MUTEX;
          oper[0].sem_op=1;
          oper[0].sem_flg=SEM_UNDO;
          oper[1].sem_num=SEMEMPTY;
          oper[1].sem_op=1;
          oper[1].sem_flg=SEM_UNDO;
          if(semop(ids, oper, 2)==-1) break;
         }
// отключение отображения общей памяти на пространство процесса
       shmdt(buf);}
// удаление разделяемой памяти
     shmctl(idm, IPC_RMID, NULL);}
  } }
// удаление набора семафоров
 semctl(ids,0,IPC_RMID);}
}
```

В отличие от Window в UNIX имеются широкие возможности по чтению информации и модификации. Не требуется никаких ухищрений для нормальной организации взаимодействия. Однако при завершении одного процесса остальные могут аварийно завершиться, поскольку в программе процесс при выходе освобождает все занятые структуры. При запуске они создаются вновь, с прежним ключом, однако это уже другие структуры, и работать с ними по старому адресу невозможно. Данная проблема возникает из-за того, что объекты не могут быть удалены автоматически, требуя программного удаления. Решение видится в том, чтобы в случае аварийной ситуации вновь пытаться создать требуемые структуры. Еще один вариант – создание счетчика активных процессов. При завершении процесс уменьшает счетчик, а последний завершающийся процесс освобождает структуры. Еще один вариант – использование внешнего процесса, отвечающего за создание и инициализацию структур, так же как и за их освобождение.

4.5. Сравнение реализации механизмов в ОС Windows и UNIX

Сравним основные возможности механизмов синхронизации, предоставляемых ОС Windows и ОС UNIX, сведя их для краткости в таблицу:

Сравнение механизмов синхронизации в ОС Windows и UNIX

Критерий	Windows	UNIX
Разнообразие механизмов синхронизации	Несколько различающихся типов	Один универсальный механизм
Создание объектов синхронизации	Отдельно для каждого объекта	Для набора объектов
Удаление объектов синхронизации	Автоматическое при отсутствии дескрипторов	Ручное
Операция P(S)	Только –1	На любую величину
Операция V(S)	На любую величину	На любую величину
Атомарное выполнение нескольких операций	Только для P(S)	Любое сочетание из набора
Получение значения семафора	Только при V(S)	Отдельная команда
Инициализация объектов	При создании	Отдельная команда
Управление набором	Отсутствует	Поддерживается
Отрицательные значения семафоров	Не поддерживаются	Не поддерживаются

Литература

1. Бек, Л. Введение в системное программирование. – М. : Мир, 1988.
2. Вахалия, Ю. UNIX изнутри. – СПб. : Питер, 2003.
3. Гордеев, А. В. Операционные системы : учебник для вузов. – СПб. : Питер, 2005.
4. Грегори, К. Использование VisualC++ 6. – М. ; СПб. ; К. : ИД «Вильямс», 2000.
5. Грибанов, В. П. и др. Операционные системы : учеб. пособие. – М. : Финансы и статистика, 1990.
6. Дейтел, Г. Введение в операционные системы: в 2 т. – М. : Мир, 1987.
7. Карпов, В. Е., Коньков, К. А. Основы операционных систем. Курс лекций: учеб. пособие. – М. : Интернет-университет информационных технологий, 2004.
8. Робачевский, А. М., Немнюгин, С. А., Стесик, О. Л. Операционная система UNIX. – СПб. : БХВ-Петербург, 2005.
9. Рочкинд, М. Программирование для UNIX. – СПб. : БХВ-Петербург, 2005.
10. Руссинович, М., Соломон, Д. Внутреннее устройство Microsoft Windows:

Windows Server 2003, Windows XP и Windows 2000. – М. : ИТД «Русская редакция»; СПб. : Питер, 2005.
11. Солдатов, В. П. Программирование драйверов Windows. – М. : ООО «Бином-Пресс», 2006.
12. Столлингс, В. Операционные системы. – М. : ИД «Вильямс», 2004.
13. Таненбаум, Э. Современные операционные системы. – СПб. : Питер, 2004.
14. Харт, Д. Системное программирование в среде Windows. – М. : ИД «Вильямс», 2005.
15. Хоар, Ч. Взаимодействующие последовательные процессы. – М. : Мир, 1989.
16. Юров, В. И. Assembler. Учебник для вузов. – СПб. : Питер, 2005.

Св. план 2007, поз. 71

Учебное издание

Прытков Валерий Александрович
Уваров Андрей Александрович
Супонев Виктор Алексеевич

ТИПОВЫЕ МЕХАНИЗМЫ СИНХРОНИЗАЦИИ ПРОЦЕССОВ

Учебно-методическое пособие
по дисциплине «Системное программное обеспечение ЭВМ»
для студентов специальности I-40 02 01
«Вычислительные машины, системы и сети»

Редактор Т. П. Андрейченко
Корректор Е. Н. Батурчик

Подписано в печать 05.02.2007.	Формат 60×84 1/16.	Бумага офсетная.
Усл. печ. л. 2,91.	Печать ризографическая.	Гарнитура «Таймс».
Уч. - изд. л. 2,8.	Тираж 100 экз.	Заказ 631.

Издатель и полиграфическое исполнение: Учреждение образования
«Белорусский государственный университет информатики и радиоэлектроники»
ЛИ №02330/0056964 от 01.04.2004. ЛП №02330/0131666 от 30.04.2004.
220013, Минск, П. Бровки, 6