# ТЕМА 2.4. СИНХРОНИЗАЦИЯ И ВЗАИМОБЛОКИРОВКА РЕСУРСОВ

В данной теме рассматриваются следующие вопросы:

- Состязательная ситуация.
- Понятия критического ресурса и области.
- Механизмы синхронизации: критические секции, мьютексы, семафоры, события, барьеры, атомарные операции.
- Проблема тупиков.
- Взаимоблокировка ресурсов в многозадачных системах.
- Решение задачи взаимоблокировки ресурсов.

Лекции – 2 часа, лабораторные занятия – 2 часа, самостоятельная работа – 2 часа.

Экзаменационные вопросы по теме:

- Синхронизация и взаимоблокировка ресурсов. Механизмы синхронизации.
- Взаимоблокировка ресурсов в многозадачных системах. Решение задачи взаимоблокировки ресурсов.

### 2.4.1. Состязательная ситуация

В некоторых операционных системах процессы, работающие совместно, могут сообща использовать некое общее хранилище данных. Каждый из процессов может считывать из общего хранилища данных и записывать туда информацию. Это хранилище представляет собой участок в основной памяти (возможно, в структуре данных ядра) или файл общего доступа. Местоположение совместно используемой памяти не влияет на суть взаимодействия и возникающие проблемы[1].

> Ситуации, в которых два (и более) процесса считывают или записывают данные одновременно и конечный результат зависит от того, какой из них был первым, называются **состояниями состязания**.

К сожалению, с ростом параллелизма из-за все большего количества ядер состязательные ситуации встречаются все чаще.

Процессам часто бывает необходимо взаимодействовать между собой. Поэтому необходимо правильно организованное взаимодействие между процессами, по возможности не использующее прерываний.

Проблема разбивается на три пункта.

- Первый: передача информации от одного процесса другому.
- Второй связан с контролем над деятельностью процессов: как гарантировать, что два процесса не пересекутся в критических ситуациях.
- Третий касается согласования действий процессов: если процесс А должен поставлять данные, а процесс В выводить их на печать, то процесс В должен подождать и не начинать печатать, пока не поступят данные от процесса А.

Второй и третий случаи относятся и к процессам, и к потокам. В первом случае у потоков нет проблем, т.к. они используют общее адресное пространство.

Передача информации от одного процесса другому может осуществляться несколькими способами:

- Разделяемая память
- Каналы (трубы), это псевдофайл, в который один процесс пишет, а другой читает.
- Сокеты — поддерживаемый ядром механизм, скрывающий особенности среды и позволяющий единообразно взаимодействовать процессам, как на одном компьютере, так и в сети.
- Почтовые ящики (только в Windows), однонаправленные, возможность широковещательной рассылки.
- Вызов удаленной процедуры, процесс А может вызвать процедуру в процессе В, и получить обратно данные.

### 2.4.2. Понятия критического ресурса и области

Чтобы избежать состязательной ситуации, нужно найти способ взаимного исключения, то есть некий способ, обеспечивающий правило, при котором если общие данные или файл используются одним процессом, возможность их использования всеми другими процессами исключается. Описанные выше трудности произошли благодаря тому, что процесс Б стал использовать общие переменные еще до того, как процесс А завершил работу с ними. Выбор подходящих элементарных операций для достижения взаимного исключения является основной проблемой конструирования любой операционной системы.

Проблемы обхода состязательных ситуаций могут быть сформулированы также в абстрактной форме. Какую-то часть времени процесс занят внутренними вычислениями и чем-нибудь другим, не создающим состязательных ситуаций. Но иногда он вынужден обращаться к общей памяти или файлам либо совершать какие-нибудь другие значимые

действия, приводящие к состязаниям. Та часть программы, в которой используется доступ к общей памяти, называется критической областью или критической секцией.

> **Критическая область** — часть программы, в которой есть обращение к совместно используемым данным.
>
> Соответственно, **критический ресурс** — тот ресурс, к которому осуществляется одновременный доступ

Если бы удалось все выстроить таким образом, чтобы никакие два процесса не находились одновременно в своих критических областях, это позволило бы избежать состязаний.

Хотя выполнение этого требования позволяет избежать состязательных ситуаций, его недостаточно для того, чтобы параллельные процессы правильно выстраивали совместную работу и эффективно использовали общие данные. Для приемлемого решения необходимо соблюдение четырех условий:

1. Два процесса не могут одновременно находиться в своих критических областях.

2. Не должны выстраиваться никакие предположения по поводу скорости или количества центральных процессоров.

3. Никакие процессы, выполняемые за пределами своих критических областей, не могут блокироваться любым другим процессом.

4. Процессы не должны находиться в вечном ожидании входа в свои критические области.

В абстрактном смысле необходимое нам поведение показано на рис. 2.4.1.

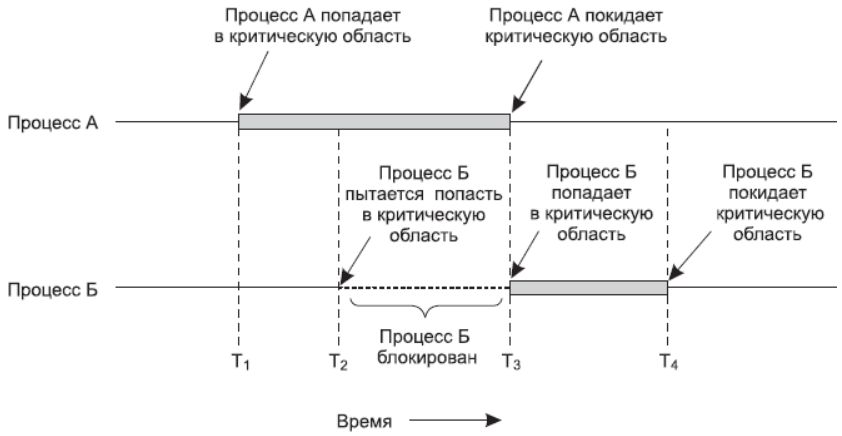


Рис. 2.4.1. Взаимное исключение использования критических областей

Мы видим, что процесс А входит в свою критическую область во время Т1. Чуть позже, когда наступает время Т2, процесс Б пытается войти в свою критическую область, но терпит неудачу, поскольку другой процесс уже находится в своей критической области, а мы допускаем это в каждый момент времени только для одного процесса. Следовательно, Б временно приостанавливается до наступления времени Т3, когда А покинет свою критическую область, позволяя Б тут же войти в свою критическую область. Со временем (в момент Т4) Б покидает свою критическую область, и мы возвращаемся в исходную ситуацию, когда ни один из процессов не находится в своей критической области.

### 2.4.3. Механизмы синхронизации: критические секции, мьютексы, семафоры, события, барьеры, атомарные операции

Для достижения режима взаимного исключения (при котором, пока один процесс занят обновлением общей памяти и находится в своей критической области, никакой другой

процесс не сможет войти в свою критическую область и создать проблему) можно предложить различные решения.

### Запрещение прерываний

В однопроцессорных системах простейшим решением является запрещение всех прерываний каждым процессом сразу после входа в критическую область и их разрешение сразу же после выхода из критической области. При запрещении прерываний не могут осуществляться никакие прерывания по таймеру. Поскольку центральный процессор переключается с одного процесса на другой в результате таймерных или каких-нибудь других прерываний, то при выключенных прерываниях он не сможет переключиться на другой процесс. Поскольку процесс запретил прерывания, он может исследовать и обновлять общую память, не опасаясь вмешательства со стороны любого другого процесса.

Но применение такого метода в пользовательском режиме вполне может привести к краху системы при некорректном поведении приложения. А на многопроцессорных системах он и вовсе бесполезен, так прерывания будут запрещены на одном процессоре, а другие процессоры смогут обратиться к критическим ресурсам. В то же время запрещение прерываний всего на несколько инструкций зачастую является очень удобным средством для самого ядра, когда оно обновляет переменные или списки.

Вывод здесь следующий: запрещение прерываний в большинстве своем является полезной технологией внутри самой операционной системы, но не подходит в качестве универсального механизма взаимных блокировок для пользовательских процессов.

### Блокирующие переменные

В качестве второй попытки рассмотрим программное решение, в котором используется одна общая (блокирующая) переменная, исходное значение которой равно нулю. Когда процессу требуется войти в свою критическую область, сначала он проверяет значение блокирующей переменной. Если оно равно 0, процесс устанавливает его в 1 и входит в критическую область. Если значение уже равно 1, процесс просто ждет, пока оно не станет равно нулю. Таким образом, нулевое значение показывает, что ни один из процессов не находится в своей критической области, а единица — что какой-то процесс находится в своей критической области.

На самом деле такое решение не защищает от состязания: существует некоторый момент времени между проверкой значения переменной и ее изменением, в который может вклиниться другой процесс.

### Строгое чередование

Третий подход к решению проблемы взаимных исключений показан на рис. 2.4.2. Этот программный фрагмент, как почти все фрагменты, приводимые в этой книге, написан на языке C.

```
while(TRUE) {
        while(turn!=0)      /*цикл*/;
        critical_region();
        turn=1;
        noncritical_region();
}
```

*а*

```
while(TRUE) {
        while(turn!=0)      /*цикл*/;
        critical_region();
        turn=0;
        noncritical_region();
}
```

*б*

Рис. 2.4.2. Предлагаемое решение проблемы критической области: а — процесс 0; б — процесс 1. В обоих случаях следует убедиться, что в коде присутствует точка с запятой, завершающая оператор while

Изначально целочисленная переменная turn, показанная на рис. 2.4.2, равна нулю и отслеживает, чья настала очередь входить в критическую область и проверять или

обновлять общую память. Сначала процесс 0 проверяет значение turn, определяет, что оно равно нулю, и входит в критическую область. Процесс 1 также определяет, что значение этой переменной равно нулю, из-за чего находится в коротком цикле, постоянно проверяя, когда turn получит значение 1. Постоянная проверка значения переменной, пока она не приобретет какое-нибудь значение, называется активным ожиданием. Как правило, этого ожидания следует избегать, поскольку оно тратит впустую время центрального процессора. Активное ожидание используется только в том случае, если есть основание полагать, что оно будет недолгим. Блокировка, использующая активное ожидание, называется спин-блокировкой.

Блокировка, использующая активное ожидание, называется **спин-блокировкой**.

Когда процесс 0 выходит из критической области, он устанавливает значение переменной turn в 1, разрешая процессу 1 войти в его критическую область. Предположим, что процесс 1 быстро выходит из своей критической области, в результате чего оба процесса находятся вне своих критических областей, а переменная turn установлена в 0. Теперь процесс 0 быстро завершает свой полный цикл, выходит из критической области и устанавливает значение turn в 1. В этот момент значение turn равно 1 и оба процесса выполняются вне своих критических областей.

Внезапно процесс 0 завершает работу вне своей критической области и возвращается к началу цикла. К сожалению, в данный момент ему не разрешено войти в его критическую область, поскольку переменная turn имеет значение 1 и процесс 1 занят работой вне своей критической области. Процесс 0 зависает в своем цикле while до тех пор, пока процесс 1 не установит значение turn в 0. Иначе говоря, когда один процесс работает существенно медленнее другого, поочередная организация вхождения в критическую область вряд ли подойдет.

Эта ситуация нарушает сформулированное ранее третье условие: процесс 0 оказывается заблокированным тем процессом, который не находится в своей критической области. То есть, хотя этот алгоритм и позволяет предотвращать любые состязательные ситуации, его нельзя рассматривать в качестве серьезного кандидата на решение проблемы, поскольку он нарушает третье условие.

### Алгоритм Петерсона

Используя сочетание идеи очередности с идеей блокирующих и предупреждающих переменных, голландский математик Деккер (T. Dekker) стал первым, кто придумал программное решение проблемы взаимного исключения, не требующее четкой очередности. В 1981 году Петерсон придумал гораздо более простой способ достижения взаимного исключения, которое перевело решение Деккера в разряд устаревших. Алгоритм Петерсона показан в листинге 2.2. Этот алгоритм состоит из двух процедур, написанных на ANSI C, а это значит, что для всех определенных и используемых функций должны быть предоставлены функции-прототипы. Но в целях экономии места мы не будем показывать прототипы ни в этом, ни в последующих примерах.

```
#define FALSE 0
#define TRUE 1
#define N 2 /* количество процессов */
int turn; /* чья очередь? */
int interested[N]; /* все исходные значения равны 0 (FALSE) */
void enter_region(int process); /* process имеет значение 0 или 1 */
{
int other; /* номер другого процесса */
other = 1 – process; /* противостоящий процесс */
interested[process] = TRUE; /* демонстрация заинтересованности */
turn = process; /* установка флажка */
while (turn == process && interested[other] == TRUE)/* цикл без инструкции
*/;
}
```

```
void leave_region(int process) /* процесс, покидающий критическую область */
{
interested[process] = FALSE; /* признак выхода из критической области */ }
```

Перед использованием общих переменных (то есть перед входом в свою критическую область) каждый процесс вызывает функцию enter_region, передавая ей в качестве аргумента свой собственный номер процесса, 0 или 1. Этот вызов заставляет процесс ждать, если потребуется, безопасного входа в критическую область. После завершения работы с общими переменными процесс, чтобы показать это и разрешить вход другому процессу, если ему это требуется, вызывает функцию leave_region.

Рассмотрим работу алгоритма. Изначально ни один из процессов не находится в критической области. Затем процесс 0 вызывает функцию enter_region. Он демонстрирует свою заинтересованность, устанавливая свой элемент массива и присваивая переменной turn значение 0. Поскольку процесс 1 заинтересованности во входе в критическую область не проявил, функция enter_region тотчас же возвращает управление. Теперь, если процесс 1 вызовет функцию enter_region, он зависнет до тех пор, пока interested[0] не получит значение FALSE, а это произойдет только в том случае, если процесс 0 вызовет функцию leave_region, чтобы выйти из критической области.

Теперь рассмотрим случай, когда оба процесса практически одновременно вызывают функцию enter_region. Оба они будут сохранять свой номер процесса в переменной turn. В расчет берется последнее сохранение, поскольку первое будет переписано и утрачено. Предположим, что процесс 1 сохранил свой номер последним и turn имеет значение 1. Когда оба процесса доберутся до оператора while, процесс 0 не выполнит его ни одного раза и войдет в свою критическую область. Процесс 1 войдет в цикл и не будет входить в свою критическую область до тех пор, пока процесс 0 не выйдет из своей критической области.

### Команда TSL (и аналогичные)

Описываемые ранее проблемы были связаны с тем, что операции чтения и изменения содержимого памяти могли быть разорваны во времени. Процессоры, предназначенные для многозадачной работы, как правило, имеют инструкции, которые выполняют эти операции атомарно, то есть неделимо.

Например, команда

TSL RX,LOCK

(TSL — Test and Set Lock, то есть проверь и установи блокировку) работает следующим образом. Она считывает содержимое слова памяти lock в регистр RX, а по адресу памяти, отведенному для lock, записывает ненулевое значение. При этом гарантируются неделимость операций чтения слова и сохранение в нем нового значения — никакой другой процесс не может получить доступ к слову в памяти, пока команда не завершит свою работу. Центральный процессор, выполняющий команду TSL, блокирует шину памяти, запрещая другим центральным процессорам доступ к памяти до тех пор, пока не будет выполнена эта команда.

Следует заметить, что блокировка шины памяти существенно отличается от запрета на прерывания. Если при выполнении чтения слова памяти с последующей записью в него запретить прерывания, ничто не помешает второму центральному процессору, подключенному к шине памяти, получить доступ к слову между чтением и записью. Получается, что запрет прерываний на процессоре 1 не оказывает абсолютно никакого воздействия на процессор 2. Перекрыть процессору 2 доступ к памяти, пока процессор 1 не завершит выполнение команды, можно только одним способом — заблокировав шину, а для этого требуется специальное оборудование (в основном для этого используется линия шины, сигнал на которой блокирует шину, исключая к ней доступ всех процессоров, кроме того, который ее заблокировал).

**Листинг 2.4.1.** Вход и выход из критической области с использованием команды TSL

```
enter region:
TSL REGISTER,LOCK  | копирование lock в регистр с присвоением ей 1
CMP REGISTER,#0    | было ли значение lock нулевым?
JNE enter_region   | если оно было ненулевым, значит, блокировка
                   | уже установлена и нужно войти в цикл
RET                | возврат управления вызывающей программе;
                   | вход в критическую область осуществлен
leave region:
MOVE LOCK,#0       | присвоение переменной lock нулевого значения
RET                | возврат управления вызывающей программе
```

Альтернативой команде TSL служит команда XCHG, осуществляющая атомарный обмен содержимого двух областей памяти, например регистра и слова памяти.

**Листинг 2.4.2.** Вход и выход из критической области с использованием команды XCHG

```
Enter_region:
MOVE REGISTER,#1   | помещение 1 в регистр
XCHG REGISTER,LOCK | обмен содержимого регистра и переменной lock
CMP REGISTER,#0    | было ли значение lock нулевым?
JNE enter_region   | если оно было ненулевым, значит, блокировка
                   | уже установлена и нужно войти в цикл
RET                | возврат управления вызывающей программе;
                   | вход в критическую область осуществлен
leave region:
MOVE LOCK,#0       | присвоение переменной lock нулевого значения
RET                | возврат управления вызывающей программе
```

### Особенность синхронизации между процессами

Синхронизация предполагает наличие общей памяти. В случае потоков проблемы нет, так как потоки разделяют адресное пространство процесса. Но у процессов разобщенные адресные пространства, поэтому нужно найти какое-то решение. Во-первых, некоторые общие структуры данных, например семафоры, могут храниться в ядре и быть доступны только через системные вызовы. Такой подход позволяет устранить проблему. Во-вторых, большинство современных операционных систем (включая UNIX и Windows) предлагают процессам способ, позволяющий использовать некоторую часть их адресного пространства совместно с другими процессами. В этом случае допускается совместное использование буферов и других структур данных. В худшем случае, когда нет доступа ни к чему другому, можно воспользоваться общим файлом.

Если два или более процесса совместно используют все свои адресные пространства или их большие части, различие между процессами и потоками немного размывается, но все равно присутствует. Два процесса, использующие общее адресное пространство, все равно имеют различные открытые файлы, аварийные таймеры и другие присущие процессам отличительные свойства, а вот для потоков в рамках одного процесса эти свойства являются общими. И никуда не деться от того обстоятельства, что несколько процессов, использующих общее адресное пространство, никогда не будут столь же эффективными, как потоки, реализованные на пользовательском уровне, поскольку к управлению процессами неизменно привлекается ядро.

### Семафоры

Ситуация изменилась в 1965 году, когда Дейкстра предложил использовать целочисленную переменную для подсчета количества активизаций, отложенных на будущее. Он предложил учредить новый тип переменной — семафор (semaphore). Значение семафора может быть равно 0, что будет свидетельствовать об отсутствии сохраненных активизаций, или иметь какое-нибудь положительное значение, если ожидается не менее одной активизации.

Дейкстра предложил использовать две операции с семафорами, которые сейчас обычно называют down и up (обобщения sleep и wakeup соответственно). Операция down выясняет, отличается ли значение семафора от 0. Если отличается, она уменьшает это значение на 1 (то есть использует одну сохраненную активизацию) и продолжает свою работу.

Если значение равно 0, процесс приостанавливается, не завершая в этот раз операцию down. И проверка значения, и его изменение, и, возможно, приостановка процесса осуществляются как единое и неделимое атомарное действие. Тем самым гарантируется, что с началом семафорной операции никакой другой процесс не может получить доступ к семафору до тех пор, пока операция не будет завершена или заблокирована. Атомарность является абсолютно необходимым условием для решения проблем синхронизации и исключения состязательных ситуаций. Атомарные действия, в которых группа взаимосвязанных операций либо выполняется без каких-либо прерываний, либо вообще не выполняется, приобрели особую важность и во многих других областях информатики.

Операция up увеличивает значение, адресуемое семафором, на 1. Если с этим семафором связаны один или более приостановленных процессов, способных завершить ранее начатые операции down, система выбирает один из них (к примеру, произвольным образом) и позволяет ему завершить его операцию down. Таким образом, после применения операции up в отношении семафора, с которым были связаны приостановленные процессы, значение семафора так и останется нулевым, но количество приостановленных процессов уменьшится на 1. Операция увеличения значения семафора на 1 и активизации одного из процессов также является неделимой. Ни один из процессов не может быть заблокирован при выполнении операции up, равно как ни один из процессов не может быть заблокирован при выполнении wakeup в предыдущей модели.

### Мьютексы

Иногда при невостребованности возможностей семафоров в качестве счетчиков используется их упрощенная версия, называемая мьютексом. Мьютексы справляются лишь с управлением взаимным исключением доступа к общим ресурсам или фрагментам кода. Простота и эффективность реализации мьютексов делает их особенно полезными для совокупности потоков, целиком реализованных в пользовательском пространстве.

Мьютекс — это совместно используемая переменная, которая может находиться в одном из двух состояний: заблокированном или незаблокированном. Следовательно, для их представления нужен только один бит, но на практике зачастую используется целое число, при этом нуль означает незаблокированное, а все остальные значения — заблокированное состояние. Для работы с мьютексами используются две процедуры. Когда потоку (или процессу) необходим доступ к критической области, он вызывает процедуру **`mutex_lock`**. Если мьютекс находится в незаблокированном состоянии (означающем доступность входа в критическую область), вызов проходит удачно и вызывающий поток может свободно войти в критическую область.

В то же время, если мьютекс уже заблокирован, вызывающий поток блокируется до тех пор, пока поток, находящийся в критической области, не завершит свою работу и не вызовет процедуру **`mutex_unlock`**. Если на мьютексе заблокировано несколько потоков, то произвольно выбирается один из них, которому разрешается воспользоваться заблокированностью других потоков.

Благодаря исключительной простоте мьютексов они легко могут быть реализованы в пользовательском пространстве при условии доступности команды TSL или XCHG.

В листинге 2.4.3 показан код процедур **`mutex_lock`** и **`mutex_unlock`**, предназначенных для

использования в совокупности потоков, работающих в пользовательском пространстве.

Решение, в котором используется команда XCHG, по сути, ничем не отличается.

**Листинг 2.4.3.** Реализация mutex_lock и mutex_unlock

```
mutex_lock:
TSL REGISTER,MUTEX  | копирование мьютекса в регистр и установка
| его в 1
CMP REGISTER,#0     | был ли мьютекс нулевым?
JZE ok              | если он был нулевым, значит, не был
                    | заблокирован,поэтому нужно вернуть
                    | управление вызывающей программе
CALL thread_yield   | мьютекс занят; пусть планировщик
                    | возобновит работу другого потока
JMP mutex lock      | повторная попытка
ok: RET             | возврат управления вызывающей программе;
                    | будет осуществлен вход в критическую область
mutex_unlock:
MOVE MUTEX,#0       | сохранение в мьютексе значения 0
RET                 | возврат управления вызывающей программе
```

В некоторых языков программирования существует высокоуровневая надстройка над мьютексами — мониторы. У мониторов имеется весьма важное свойство, позволяющее успешно справляться со взаимными исключениями: в любой момент времени в мониторе может быть активен только один процесс. Мониторы являются конструкцией языка программирования, поэтому компилятор осведомлен об их особенностях и способен обрабатывать вызовы процедур монитора не так, как вызовы всех остальных процедур. Обычно при вызове процессом процедуры монитора первые несколько команд процедуры осуществляют проверку на текущее присутствие активности других процессов внутри монитора. Если какой-нибудь другой процесс будет активен, вызывающий процесс будет приостановлен до тех пор, пока другой процесс не освободит монитор. Если монитор никаким другим процессом не используется, вызывающий процесс может в него войти.

Реализация взаимного исключения при входе в монитор возлагается на компилятор, но чаще всего для этого используется мьютекс или двоичный семафор. Поскольку обеспечением взаимного исключения занимается компилятор, а не программист, вероятность того, что будут допущены какие-то неправильные действия, становится гораздо меньше. В любом случае тот, кто создает монитор, не должен знать, как именно компилятор обеспечивает взаимное исключение. Достаточно лишь знать, что после превращения всех критических областей в процедуры монитора никакие два процесса не будут выполнять код своих критических областей в одно и то же время.

### Передача сообщений

Передача сообщений использует два примитива, **send** и **receive**, которые, подобно семафорам и в отличие от мониторов, являются системными вызовами, а не конструкциями языка. Как таковые они легко могут быть помещены в библиотечные процедуры, например:

```
send(destination, &message);
```

или

```
receive(source, &message);
```

Первый вызов отправляет сообщение заданному получателю, а второй получает сообщение из заданного источника (или из любого источника, если получателю все равно). Если доступные сообщения отсутствуют, получатель может заблокироваться до их поступления. Или же он может немедленно вернуть управление с кодом ошибки.

Чтобы застраховаться от утраты сообщений, отправитель и получатель должны договориться о том, что как только сообщение будет получено, получатель должен

отправить в ответ специальное подтверждение. Если по истечении определенного интервала времени отправитель не получит подтверждение, он отправляет сообщение повторно.

Доступно множество вариантов передачи сообщений. В способе адресации сообщений каждому процессу присваивается уникальный адрес и сообщения адресуются процессам. Можно также изобрести новую структуру данных, называемую почтовым ящиком. Этот ящик представляет собой место для буферизации конкретного количества сообщений, которое обычно указывается при его создании. При использовании почтовых ящиков в качестве параметров адреса в вызовах **send** и **receive** указываются почтовые ящики, а не процессы. Когда процесс пытается послать сообщение заполненному почтовому ящику, он приостанавливается до тех пор, пока из этого почтового ящика не будет извлечено сообщение, освобождая пространство для нового сообщения.

### Барьеры

Этот механизм синхронизации предназначен для групп процессов. Некоторые приложения разбиты на фазы и следуют правилу, согласно которому ни один из процессов не может перейти к следующей фазе, пока все процессы не будут готовы перейти к следующей фазе. Добиться выполнения этого правила можно с помощью барьеров, поставленных в конце каждой фазы. Когда процесс достигает барьера, он блокируется до тех пор, пока этого барьера не достигнут все остальные процессы. Это позволяет синхронизировать группы процессов. Действие барьера представлено на рис. 2.4.3.

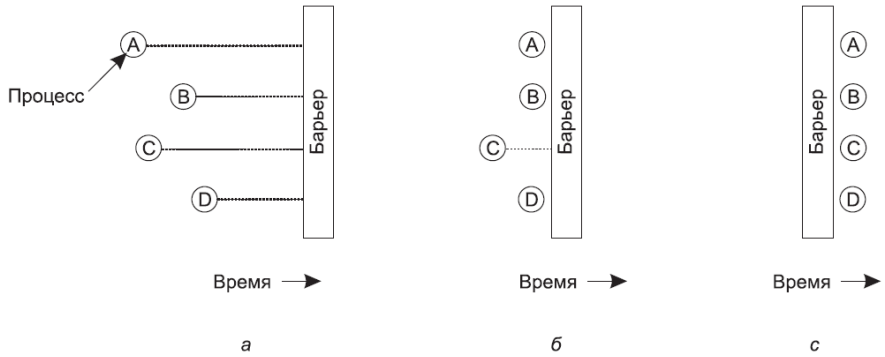


Рис. 2.4.3. Использование барьера: а — процесс достигает барьера; б — все процессы, кроме одного, заблокированы на барьере; в — последний процесс достигает барьера, и все процессы преодолевают этот барьер

### Объекты синхронизации Windows

**Объект критического раздела** обеспечивает синхронизацию, аналогичную той, которая предоставляется объектом мьютекса, за исключением того, что критический раздел может использоваться только потоками одного процесса [2]. Объекты критических разделов нельзя совместно использовать в процессах.

Объекты событий, мьютексов и семафоров также можно использовать в однопроцессном приложении, но объекты критически важных разделов обеспечивают немного более быстрый и эффективный механизм синхронизации взаимоисключения (зависящий от процессора тест и инструкция установки). Как и объект мьютекса, объект критического раздела может принадлежать только одному потоку за раз, что делает его полезным для защиты общего ресурса от одновременного доступа. В отличие от объекта мьютекса, невозможно определить, был ли отброшен критический раздел.

Чтобы использовать критические секции, сначала нужно объявить переменную типа **CRITICAL_SECTION**. Прежде чем потоки процесса смогут использовать его, инициализируйте критический раздел с помощью функции **InitializeCriticalSection** или **InitializeCriticalSectionAndSpinCount**.

Поток использует функцию **EnterCriticalSection** или **TryEnterCriticalSection** для запроса владения критически важным разделом. Функция **LeaveCriticalSection** используется для освобождения владения критически важным разделом. Если объект критического раздела в настоящее время принадлежит другому потоку, **EnterCriticalSection** бесконечно ожидает права владения.

Любой поток процесса может использовать функцию **DeleteCriticalSection** для освобождения системных ресурсов, выделенных при инициализации объекта критического раздела. После вызова этой функции объект критического раздела нельзя использовать для синхронизации.

Объект **мьютекса** — это объект синхронизации, состояние которого устанавливается в значение Signaled, если он не принадлежит ни одному потоку, и без знака, когда он принадлежит [3]. Только один поток за раз может владеть объектом мьютекса, имя которого происходит из-за того, что он полезен для координации взаимоисключающего доступа к общему ресурсу. Например, чтобы предотвратить запись двух потоков в общую память одновременно, каждый поток ожидает владения объектом мьютекса перед выполнением кода, который обращается к памяти. После записи в общую память поток освобождает объект мьютекса.

Поток использует функцию **CreateMutex** или **CreateMutexEx** для создания объекта мьютекса. Создающий поток может запросить немедленное владение объектом мьютекса, а также указать имя для объекта мьютекса. Он также может создать неименованный мьютекс.

Потоки в других процессах могут открывать дескриптор существующего именованного объекта мьютекса, указывая его имя в вызове функции **OpenMutex**. Чтобы передать дескриптор неименованного мьютекса другому процессу, используйте функцию **DuplicateHandle** или наследование дескриптора типа "родители-потомки".

Любой поток с дескриптором объекта мьютекса может использовать одну из функций ожидания для запроса владения объектом мьютекса. Если объект мьютекса принадлежит другому потоку, функция ожидания блокирует запрашивающий поток до тех пор, пока поток-владение не освободит объект мьютекса с помощью функции **ReleaseMutex**. Возвращаемое значение функции **wait** указывает, была ли функция возвращена по какой-либо причине, кроме состояния мьютекса, для которой задано значение **signaled**.

**Объект семафора** — это объект синхронизации, который поддерживает число от нуля до указанного максимального значения [4]. Число уменьшается каждый раз, когда поток завершает ожидание объекта семафора, и увеличивается каждый раз, когда поток освобождает семафор. Когда число достигает нуля, больше потоки не могут ждать, пока состояние объекта семафора станет сигналом. Состояние семафора становится сигнальным, когда это число становится больше нуля, и несигнальным, когда равно нулю.

Объект семафора полезен при управлении общим ресурсом, который может поддерживать ограниченное число пользователей. Он выступает в качестве шлюза, который ограничивает количество потоков, совместно использующих ресурс, указанным максимальным числом. Например, приложение может установить ограничение на количество создаваемых окон. Он использует семафор с максимальным числом, равным предельному значению окна, уменьшая счетчик при создании окна и при каждом закрытии окна. Приложение указывает объект семафора в вызове одной из функций ожидания перед созданием каждого окна. Если значение счетчика равно нулю, указывающее, что достигнуто предельное количество окон, функция ожидания блокирует выполнение кода создания окна.

Поток использует функцию **CreateSemaphore** или **CreateSemaphoreEx** для создания объекта семафора. В потоке создания указывается начальное число и максимальное значение

счетчика для объекта. Начальное число не должно быть ни меньше нуля, ни больше максимального значения. Создающий поток также может указать имя объекта семафора. Потоки в других процессах могут открывать дескриптор существующего объекта семафора, указывая его имя в вызове функции **OpenSemaphore**.

Начальное число семафора обычно устанавливается в максимальное значение. Затем количество уменьшается по сравнению с этим уровнем по мере использования защищенного ресурса. Кроме того, можно создать семафор с начальным числом, равным нулю, чтобы заблокировать доступ к защищенному ресурсу во время инициализации приложения. После инициализации можно использовать **ReleaseSemaphore**, чтобы увеличить число до максимального значения.

Завершив использование объекта семафора, вызовите функцию **CloseHandle**, чтобы закрыть дескриптор. Объект семафора уничтожается при закрытии последнего дескриптора. Закрытие дескриптора не влияет на число семафоров; Поэтому обязательно вызовите **ReleaseSemaphore** перед закрытием дескриптора или перед завершением процесса. В противном случае время ожидания операций ожидания будет истекает или продолжается бесконечно в зависимости от того, задано ли значение времени ожидания.

**Объект события** — это объект синхронизации, состояние которого может быть явно задано с помощью функции **SetEvent** [5]. Ниже приведены два типа объекта события.

**Событие сброса вручную**. Объект события, состояние которого остается сигнальным до тех пор, пока функция **ResetEvent** явно не будет сброшена на незначимую. Во время передачи сигнала можно освободить любое количество ожидающих потоков или потоков, которые впоследствии указывают один и тот же объект события в одной из функций ожидания.

**Событие автоматического сброса**. Объект события, состояние которого остается сигнальным до тех пор, пока не будет освобожден один поток ожидания, в этот момент система автоматически устанавливает состояние без знака. Если ожидающих потоков нет, состояние объекта события остается сигнальным. Если ожидается несколько потоков, выбирается ожидающий поток. Не предполагайте порядок "первым в очереди" (FIFO). Внешние события, такие как APC в режиме ядра, могут изменить порядок ожидания.

Объект события полезен при отправке в поток сигнала, указывающего на то, что произошло определенное событие. Например, в перекрывающихся входных и выходных данных система присваивает указанному объекту события сигнальное состояние после завершения перекрывающейся операции. Один поток может указать различные объекты событий в нескольких одновременных перекрывающихся операциях, а затем использовать одну из функций ожидания с несколькими объектами, чтобы дождаться получения сигнала о состоянии любого из объектов событий.

Поток использует функцию **CreateEvent** или **CreateEventEx** для создания объекта события. Создаваемый поток определяет начальное состояние объекта и указывает, является ли он объектом события сброса вручную или автоматическим сбросом. Создающий поток также может указать имя объекта события. Потоки в других процессах могут открывать дескриптор для существующего объекта события, указывая его имя в вызове функции **OpenEvent**.

**Барьер синхронизации** позволяет нескольким потокам ждать, пока все потоки не достигнут определенной точки выполнения, прежде чем поток продолжит работу. Барьеры синхронизации не могут быть разделены между процессами [6].

Барьеры синхронизации полезны для поэтапных вычислений, в которых потоки, выполняющие один и тот же код в параллельном режиме, должны завершить один этап, прежде чем переходить к следующему.

Чтобы создать барьер синхронизации, вызовите функцию **InitializeSynchronizationBarrier** и укажите максимальное количество потоков и количество раз, когда поток должен вращаться перед блокировкой. Затем запустите потоки, которые будут использовать барьер. После завершения работы каждый поток вызывает **EnterSynchronizationBarrier**, чтобы подождать у барьера. Функция **EnterSynchronizationBarrier** блокирует каждый поток до тех пор, пока число потоков, заблокированных в барьере, не достигнет максимального количества потоков для барьера, после чего **EnterSynchronizationBarrier** разблокирует все потоки. Функция **EnterSynchronizationBarrier** возвращает значение **TRUE** для одного из потоков, которые вошли в барьер, и возвращает значение **FALSE** для всех остальных потоков.

Чтобы снять барьер синхронизации, когда он больше не нужен, вызовите Метод **DeleteSynchronizationBarrier**. Эту функцию можно вызывать сразу после вызова **EnterSynchronizationBarrier**, так как она гарантирует, что все потоки завершали использование барьера перед освобождением.

### 2.4.4. Проблема тупиков

При работе многих приложений процессу нужен исключительный доступ не к одному, а сразу к нескольким ресурсам. Если несколько приложений будут при этом блокировать одни и те же ресурсы, при этом неизбежны конфликты, и даже неразрешимые. Например, процесс 1 заблокировал ресурс А и пытается получить исключительный доступ к ресурсу Б. В это же время процесс 2 заблокировал ресурс Б и пытается получить исключительный доступ к ресурсу А. Здесь процессы 1 и 2 заблокировали сами себя.

Основная часть взаимоблокировок связана с ресурсами, к которым некоторым процессам были предоставлены исключительные права доступа. К их числу относятся устройства, записи данных, файлы и т. д. Чтобы придать рассмотрению взаимоблокировок как можно более универсальный характер, мы будем называть объекты, к которым предоставляется доступ, **ресурсами**. Ресурсами могут быть аппаратные устройства или какая-то часть информации (например, запись базы данных). Некоторые ресурсы могут быть доступны в нескольких идентичных экземплярах. Другими словами, под ресурсом понимается все, что должно предоставляться, использоваться и через некоторое время высвобождаться, поскольку в один и тот же момент времени может использоваться только одним процессом.

Ресурсы бывают двух видов: выгружаемые и невыгружаемые.

К **выгружаемым** относятся такие ресурсы, которые могут быть безболезненно отобраны у процесса, который ими обладает. Примером такого ресурса может послужить память. При недостатке памяти в системе можно забрать память у менее приоритетного процесса и выгрузить ее в файл подкачки.

**Невыгружаемый** ресурс нельзя отобрать у его текущего владельца, не вызвав потенциально сбоя в вычислениях. Обычно это аппаратные устройства, как принтер или накопитель на магнитной ленте. Впрочем, на смартфоне, не поддерживающем свопинг или страничную организацию памяти, простой выгрузкой взаимоблокировки из-за дефицита памяти избежать не удастся.

Как правило, во взаимоблокировках фигурируют невыгружаемые ресурсы. Обычно потенциальные взаимоблокировки с участием выгружаемых ресурсов могут быть устранены путем перераспределения ресурсов от одного процесса к другому. Поэтому наше внимание будет сконцентрировано на невыгружаемых ресурсах.

В наиболее общем виде при использовании ресурса происходит следующая последовательность событий:

1. Запрос ресурса.

2. Использование ресурса.

3. Высвобождение ресурса.

Если во время запроса ресурс недоступен, запрашивающий процесс вынужден перейти к ожиданию. В некоторых операционных системах при отказе в выделении запрошенного ресурса процесс автоматически блокируется, а когда ресурс становится доступен — возобновляется. В других системах отказ в выделении запрашиваемого ресурса сопровождается кодом ошибки, и принятие решения о том, что следует делать, немного подождать или попытаться снова получить ресурс, возлагается на вызывающий процесс.

Процесс, чей запрос на выделение ресурса был только что отклонен, обычно входит в короткий цикл: запрос ресурса, затем приостановка, — после чего повторяет попытку. Хотя этот процесс не заблокирован, но по всем показателям он является фактически заблокированным, поскольку не может выполнять никакой полезной работы. При дальнейшем рассмотрении вопроса мы будем предполагать, что при отказе в выделении запрошенного ресурса процесс впадает в спячку.

### 2.4.5. Взаимоблокировка ресурсов в многозадачных системах

Взаимоблокировкам можно дать следующее формальное определение.

> **Взаимоблокировка** в группе процессов возникает в том случае, если каждый процесс из этой группы ожидает события, наступление которого зависит исключительно от другого процесса из этой же группы.

Поскольку все процессы находятся в состоянии ожидания, ни один из них не станет причиной какого-либо события, которое могло бы возобновить работу другого процесса, принадлежащего к этой группе, и ожидание всех процессов становится бесконечным. В этой модели предполагается, что у процессов есть только один поток, а прерывания, способные возобновить работу заблокированного процесса, отсутствуют. Условие отсутствия прерываний необходимо, чтобы не позволить заблокированному по иным причинам процессу возобновить свою работу, скажем, по аварийному сигналу, после чего вызвать событие, освобождающее другие имеющиеся в группе процессы.

В большинстве случаев событием, наступления которого ожидает каждый процесс, является высвобождение какого-либо ресурса, которым на данный момент владеет другой участник группы. Иными словами, каждый процесс из группы, попавшей в ситуацию взаимоблокировки, ожидает ресурса, которым обладает другой процесс из этой же группы. Ни один из процессов не может работать, ни один из них не может высвободить какой-либо ресурс, и ни один из них не может возобновить свою работу. Количество процессов и количество и вид удерживаемых и запрашиваемых ресурсов не имеет значения. Этот результат сохраняется для любого типа ресурсов, включая аппаратные и программные ресурсы. Этот вид взаимоблокировки называется **ресурсной взаимоблокировкой**. Наверное, это самый распространенный, но далеко не единственный вид.

Коффман (Coffman et al., 1971) показал, что для возникновения ресурсных взаимоблокировок должны выполняться четыре условия:

1. **Условие взаимного исключения**. Каждый ресурс либо выделен в данный момент только одному процессу, либо доступен.

2. **Условие удержания и ожидания**. Процессы, удерживающие в данный момент ранее выделенные им ресурсы, могут запрашивать новые ресурсы.

3. **Условие невыгружаемости**. Ранее выделенные ресурсы не могут быть принудительно отобраны у процесса. Они должны быть явным образом высвобождены тем процессом, который их удерживает.

4. **Условие циклического ожидания**. Должна существовать кольцевая последовательность из двух и более процессов, каждый из которых ожидает высвобождения ресурса, удерживаемого следующим членом последовательности.

Для возникновения ресурсной взаимоблокировки должны соблюдаться все четыре условия. Если одно из них не соблюдается, ресурсная взаимоблокировка невозможна.

Холт (Holt, 1972) показал, как эти четыре условия могут быть смоделированы с использованием направленных графов. У графов имеется два вида узлов: процессы, показанные окружностями, и ресурсы, показанные квадратами. Направленное ребро, которое следует от узла ресурса (квадрата) к узлу процесса (окружности), означает, что этот ресурс был ранее запрошен, получен и на данный момент удерживается этим процессом. Направленное ребро, идущее от процесса к ресурсу, означает, что процесс в данное время заблокирован в ожидании высвобождения этого ресурса.

На рис. 2.4.4, *а* ресурс R в данное время выделен процессу A. Направленное ребро, идущее от процесса к ресурсу, означает, что процесс в данное время заблокирован в ожидании высвобождения этого ресурса. На рис. 2.4.4, *б* процесс B ожидает высвобождения ресурса S. На рис. 2.4.4, *в* мы наблюдаем взаимоблокировку:

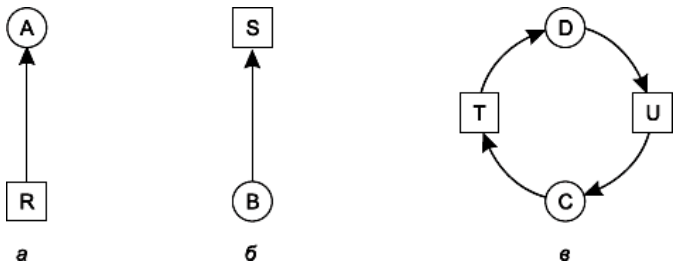


Рис. 2.4.4. Графы распределения ресурсов: а — ресурс занят; б — запрос ресурса; в — взаимоблокировка

### 2.4.6. Решение задачи взаимоблокировки ресурсов

Чаще всего для борьбы с взаимными блокировками используются четыре стратегии:

1. **Игнорирование проблемы**. Может быть, если вы проигнорируете ее, она проигнорирует вас.

2. **Обнаружение и восстановление**. Дайте взаимоблокировкам проявить себя, обнаружьте их и выполните необходимые действия.

3. **Динамическое уклонение** от них за счет тщательного распределения ресурсов.

4. **Предотвращение** за счет структурного подавления одного из четырех условий, необходимых для их возникновения.

#### Страусиный алгоритм

Самым простым подходом к решению проблемы является «страусиный алгоритм»: спрячьте голову в песок и сделайте вид, что проблема отсутствует. Люди реагируют на эту стратегию по-разному. Математики считают ее неприемлемой и говорят, что взаимоблокировки следует предотвращать любой ценой. Инженеры спрашивают, как часто ожидается возникновение проблемы, как часто система дает сбой по другим причинам и насколько серьезны последствия взаимоблокировки. Если взаимоблокировка возникает в среднем один раз в пять лет, а система раз в неделю сбоит из-за технических отказов и дефектов операционной системы, большинство инженеров не захотят платить за избавление от взаимоблокировок существенным снижением производительности или удобства использования.

### Обнаружение взаимоблокировок и восстановление работоспособности

При использовании этой технологии система не пытается предотвращать взаимоблокировки. Она позволяет им произойти, пытается обнаружить момент их возникновения, а затем предпринимает некоторые действия по восстановлению работоспособности.

В случае уникального ресурса для системы нужно построить ресурсный граф. Если этот граф содержит один и более циклов, значит, мы имеем дело с взаимоблокировкой. Любой процесс, являющийся частью цикла, заблокирован намертво. Если циклов нет, значит, система не находится в состоянии взаимоблокировки. Известно множество алгоритмов для обнаружения циклов в направленных графах.

Когда в системе существует несколько экземпляров каких-нибудь ресурсов, для обнаружения взаимоблокировки необходим другой подход. В [1] описан алгоритм, основанный на использовании матриц и предназначенный для обнаружения взаимоблокировки при работе *n* процессов. Алгоритм использует матрицы текущего распределения и запросов и векторы существующих ресурсов и доступных ресурсов.

После обнаружения взаимоблокировки нужно из нее выйти, чтобы восстановить работоспособность системы.

**Восстановление за счет приоритетного овладения ресурсом**. Возможность отобрать ресурс у процесса, позволить использовать его другому процессу, а затем вернуть его без извещения процесса во многом зависит от природы этого ресурса. Восстановление этим способом зачастую затруднено или вовсе невозможно. Выбор процесса для приостановки обусловлен тем, какой именно процесс обладает тем ресурсом, который у него можно легко отобрать.

**Восстановление путем отката**. Если разработчики системы и операторы вычислительной машины знают о том, что есть вероятность возникновения взаимоблокировки, они могут организовать периодическое создание процессами контрольных точек. Это означает, что состояние процесса записывается в файл, что позволит осуществить его последующий перезапуск. Чтобы выйти из взаимоблокировки, процесс, владеющий необходимым ресурсом, откатывается назад к точке, предшествующей получению данного ресурса, для чего он запускается из одной из своих контрольных точек. Вся работа, выполненная после этой контрольной точки, теряется.

**Восстановление путем уничтожения процессов**. Самым грубым, но и самым простым способом прервать взаимоблокировку является уничтожение одного или нескольких процессов. Можно уничтожить процесс, находящийся в цикле взаимоблокировки. Если повезет, то другие процессы смогут продолжить свою работу. Если это не поможет, то все можно повторить, пока цикл не будет разорван.

### Уклонение от взаимоблокировок

Мы предполагали, что, когда процесс запрашивает ресурсы, он просит их все сразу. Но в большинстве систем ресурсы запрашиваются по одному. Система должна уметь принимать решение, представляет выделение ресурса опасность или нет, и выделять его только в том случае, если это безопасно. И при определенных условиях взаимоблокировки можно избежать, но только если заранее будет доступна вполне определенная информация.

Основные алгоритмы уклонения от взаимоблокировок основаны на концепции безопасных состояний. Состояние считается **безопасным**, если существует какой-то порядок планирования, при котором каждый процесс может доработать до конца, даже если все процессы внезапно и срочно запросят максимальное количество ресурсов.

Следует отметить, что небезопасное состояние само по себе не является состоянием взаимоблокировки. разница между безопасным и небезопасным состоянием заключается

в том, что в безопасном состоянии система может гарантировать, что все процессы закончат свою работу, а в небезопасном состоянии такой гарантии дать нельзя.

### Алгоритм банкира

Алгоритм планирования, позволяющий избегать взаимоблокировок, был разработан Дейкстрой (Dijkstra, 1965) и известен как алгоритм банкира. Модель алгоритма основана на примере банкира маленького городка, имеющего дело с группой клиентов, которым он выдал ряд кредитов. (Много лет назад банки не давали кредиты, пока не убеждались в том, что они могут быть возвращены.) Алгоритм проверяет, ведет ли выполнение каждого запроса к небезопасному состоянию. Если да, то запрос отклоняется. Если удовлетворение запроса к ресурсу приводит к безопасному состоянию, ресурс предоставляется процессу. Чтобы понять, является ли состояние безопасным, банкир проверяет, может ли он предоставить достаточно ресурсов для удовлетворения запросов какого-нибудь клиента. Если да, то эти ссуды считаются возвращенными, после чего проверяется следующий ближайший к пределу займа клиент и т. д. Если в конечном счете все ссуды могут быть погашены, состояние является безопасным и исходный запрос можно удовлетворить.

Алгоритм банкира может быть распространен на работу с несколькими ресурсами. При этом используются матрицы выделения ресурсов и запросов.

Хотя алгоритм замечателен в теории, на практике он по существу бесполезен, поскольку нечасто можно определить заранее, каковы будут максимальные потребности процессов в ресурсах. Кроме того, количество процессов не фиксировано, оно динамически изменяется по мере входа пользователей в систему и выхода их из нее. И более того, ресурсы, считавшиеся доступными, могут внезапно пропасть. Таким образом, на практике лишь немногие системы, если таковые вообще имеются, используют алгоритм банкира для уклонения от взаимоблокировок. Но в некоторых системах для предотвращения взаимных блокировок используются эвристические правила, подобные алгоритму банкира. Например, сети могут дросселировать трафик, когда использование буфера превысит, скажем, 70 %, при оценке, что оставшихся 30 % будет достаточно для завершения обслуживания текущих пользователей и возвращения их ресурсов.

### Предотвращение взаимоблокировки

Если мы сможем гарантировать, что хотя бы одно из четырех условий возникновения взаимоблокировок никогда не будет выполнено, то взаимоблокировки станут структурно невозможными.

Сначала предпримем атаку на условие взаимного исключения. Если в системе нет ресурсов, отданных в единоличное пользование одному процессу, мы никогда не попадем в ситуацию взаимоблокировки. Данные проще всего сделать доступными только для чтения, чтобы процессы могли их использовать одновременно. Но также понятно, что если позволить двум процессам одновременно печатать данные на принтере, то это приведет к хаосу. За счет использования очереди на печать (спулинга) выдавать свои выходные данные могут сразу несколько процессов. В этой модели единственным процессом, который фактически запрашивает физический принтер, является демон принтера. Так как демон не запрашивает никакие другие ресурсы, взаимоблокировки, связанные с принтером, можно исключить.

Второе из условий, сформулированных Коффманом (Coffman et al., 1971), выглядит несколько более обещающим. Если можно будет помешать процессам, удерживающим ресурсы, войти в фазу ожидания дополнительных ресурсов, то можно будет исключить и взаимоблокировку. Один из способов достижения этой цели заключается в том, чтобы заставить все процессы запрашивать все свои ресурсы до начала выполнения своей работы. Если все доступно, то процессу будет выделено все, что ему требуется, и он

сможет доработать до завершения. Если один или несколько ресурсов заняты, ничего не будет выделяться и процесс будет просто ждать.

Проблема, непосредственно связанная с этим подходом, состоит в том, что многие процессы не знают, сколько ресурсов им понадобится, пока не начнут работу. Фактически если бы они об этом знали, то можно было бы использовать и алгоритм банкира. Вторая проблема состоит в том, что при таком подходе ресурсы не будут использоваться оптимально.

Возможна также атака и третьего условия (невыгружаемости). Если процессу выделен принтер и он распечатал лишь половину своих выходных данных, то принудительно отобрать у него принтер по причине недоступности запрошенного плоттера в лучшем случае будет слишком затруднительно, а в худшем — просто невозможно. Тем не менее, чтобы избежать подобной ситуации, некоторые ресурсы могут быть виртуализированы. Сохранение очереди на печать на диске и предоставление возможности доступа к реальному принтеру только демону принтера исключает возникновение взаимоблокировок с участием принтера, хотя и создает одну из таких потенциальных возможностей в отношении дискового пространства. Но при наличии дисков большой емкости исчерпание дискового пространства становится маловероятным.

Однако не все ресурсы могут быть виртуализированы подобным образом. К примеру, чтобы записи в базах данных или в таблицах внутри операционной системы могли использоваться, они должны быть заблокированы, и здесь закладывается потенциальная вероятность взаимоблокировки.

Циклическое ожидание можно устранить несколькими способами. Один из них заключается в простом выполнении правила, которое гласит, что процессу в любой момент времени дано право только на один ресурс. Если нужен второй ресурс, процесс обязан освободить первый. Но подобное ограничение не всегда приемлемо для процесса.

Другой способ, позволяющий избежать циклического ожидания, заключается в поддержке общей нумерации всех ресурсов. Теперь действует следующее правило: процессы могут запрашивать ресурс, когда только пожелают, но все запросы должны быть сделаны в порядке нумерации ресурсов. Процесс может запросить сначала принтер, затем накопитель на магнитной ленте, но не может сначала потребовать плоттер, а затем принтер. Если придерживаться этого правила, то у графа распределения ресурсов никогда не будет циклов.

Хотя порядковая нумерация ресурсов исключает проблему взаимоблокировок, может не представиться возможности подобрать порядок, удовлетворяющий абсолютно всех. Когда ресурсы включают в себя элементы таблицы процессов, дисковое пространство очереди печати, заблокированные записи базы данных и другие абстрактные ресурсы, количество потенциальных ресурсов и различных применений может быть настолько большим, что не сможет работать никакое упорядочение.

### Другие вопросы взаимоблокировок

Хотя и уклонение от взаимоблокировок, и предупреждение их возникновения в общем случае оказались не слишком перспективными средствами, для определенных приложений известно множество превосходных алгоритмов специального назначения.

К примеру, во многих системах управления базами данных часто встречающимися операциями являются запросы на блокирование нескольких записей с последующим обновлением всех заблокированных записей. Когда одновременно запущено несколько процессов, существует реальная опасность возникновения взаимоблокировок. Часто используемый при этом подход называется **двухфазным блокированием**. В первой фазе процесс пытается заблокировать по одной все необходимые ему записи. Если все проходит успешно, он приступает ко второй фазе, осуществляя свои обновления и

снимая блокировку. При этом в первой фазе не проводится никакой реальной работы. Если в первой фазе встретятся некоторые необходимые записи, которые уже заблокированы, процесс просто снимает все свои блокировки и начинает первую фазу заново. Но эта стратегия в общем виде неприменима. К примеру, в системах реального времени и системах управления процессами остановка процесса на полпути только из-за недоступности ресурса и его повторный запуск с самого начала просто недопустимы.

Другая разновидность взаимоблокировок может проявиться в системах обмена данными (например, сетях), в которых один и более процессов связываются путем обмена сообщениями. Общая договоренность предполагает, что процесс А отправляет сообщение-запрос процессу В, а затем блокируется до тех пор, пока В не пошлет назад ответное сообщение. Предположим, что сообщение-запрос где-то затерялось. Процесс А заблокирован в ожидании ответа. Процесс В заблокирован в ожидании запроса на какие-либо его действия. В результате возникает взаимоблокировка. Подобную ситуацию назвали **коммуникационной взаимоблокировкой**, чтобы отличить ее от более распространенной ресурсной взаимоблокировки. Коммуникационная взаимоблокировка является аномальным вариантом синхронизации совместных действий. Процесс в этом типе взаимной блокировки не может завершить обслуживание, если выполняется независимо от других процессов. Для прекращения коммуникационной взаимоблокировки обычно применяется **истечение времени ожидания** (timeout).

**Активная взаимоблокировка** (livelock) возникает, когда процесс старается проявить вежливость, отказавшись от уже приобретенной блокировки, как только замечает, что не может получить следующую блокировку, в которой нуждается. Затем он ждет, скажем, несколько миллисекунд и повторяет попытку. В принципе, в таком поведении нет ничего плохого, и оно помогает обнаружить взаимоблокировку и избавиться от нее. Но если другой процесс делает то же самое в точности в то же самое время, эти процессы попадут в ситуацию, похожую на ту, когда два человека пытаются разминуться на улице и каждый из них вежливо уходит в сторону, но из этого ничего не выходит, поскольку они пытаются одновременно уйти в одну и ту же сторону.

Проблемой, тесно связанной как с обычной, так и с активной взаимоблокировкой, является **зависание**. В динамической системе запрос ресурсов происходит постоянно. Для того чтобы принять решение, кто и когда какой ресурс получит, нужна определенная политика. Эта политика, хотя бы и разумная, может привести к тому, что некоторые процессы никогда не будут обслужены, даже если они не находятся в состоянии взаимоблокировки. Зависания можно избежать за счет использования политики распределения ресурсов «первым пришел — первым и обслужен». При таком подходе процесс, ожидающий дольше всех, обслуживается следующим. В конечном итоге любой заданный процесс со временем станет самым старшим в очереди и получит необходимый ему ресурс.

## Список использованных источников

1. Таненбаум, Э. Современные операционные системы. / Э. Таненбаум, Х. Бос. — 4-е изд. — СПб.: Питер, 2015. — 1120 с.

2. Объекты критического раздела
https://learn.microsoft.com/ru-ru/windows/win32/sync/critical-section-objects

3. Объекты мьютексов
https://learn.microsoft.com/ru-ru/windows/win32/sync/mutex-objects

4. Объекты семафора
https://learn.microsoft.com/ru-ru/windows/win32/sync/semaphore-objects

5. Объекты событий (синхронизация)
https://learn.microsoft.com/ru-ru/windows/win32/sync/event-objects

6. Барьеры синхронизации
https://learn.microsoft.com/ru-ru/windows/win32/sync/synchronization-barriers