Синхронизация процессов

Потребность в синхронизации процессов возникает только в мультипрограммных ОС и связана с совместным использованием аппаратных и информационных ресурсов вычислительной системы. Выполнение процессов в таких ОС в общем случае имеет асинхронный характер, т.е. процессы выполняются независимо в том плане, что практически невозможно с полной определенностью сказать, на каком этапе выполнения будет находиться определенный процесс в определенный момент времени.

Суть синхронизации процессов состоит в согласовании их скоростей путем приостановки процесса до наступления некоторого события и последующей его активизации при наступлении этого события.

Синхронизация лежит в основе любого *взаимодействия процессов,* которое может быть связано:

- с обменом данными (процесс-получатель должен обращаться за данными только после их записи процессом-отправителем);
- с разделением ресурсов (например, если активному процессу требуется доступ к последовательному порту, занятому другим процессом, то активный процесс должен быть приостановлен до освобождения ресурса);
- с синхронизацией процесса с внешними событиями (например, с нажатием комбинации клавиш).

Сложность проблемы синхронизации состоит в нерегулярности возникающих ситуаций при взаимодействии процессов. Пренебрежение вопросами синхронизации может привести к неправильной работе процессов или даже к краху системы. Примерами таких ситуаций являются *гонки* и *тупики*.

Гонками называются ситуации, когда в отсутствие синхронизации два (или более) процесса обрабатывают разделяемые данные и конечный результат зависит от соотношения скоростей процессов.

Тупики – это взаимные блокировки процессов, могущие возникать вследствие недостаточно корректного решения задачи синхронизации и состоящие в том, что ряд процессов удерживает ресурсы, запрашиваемые другими процессами, и в то же время запрашивает ресурсы, удерживаемые другими.

Гонки рассмотрим на примере приложения, ведущего базу данных о клиентах некоторого предприятия (рис. 2.12). Сведения о каждом клиенте представляют собой запись, содержащую поле описания заказа клиента и поле оплаты заказа. Таким образом, запись изменяется в двух случаях: когда клиент делает заказ и когда он его оплачивает.

Пусть приложение оформлено как единый процесс, имеющий два потока, A и B. Пусть

поток А заносит в базу данные о заказах, а поток В – данные об оплате. Оба потока совместно работают над общим файлом базы данных по следующему общему алгоритму.

1. Считать из файла базы данных в буфер запись о клиенте.
2. Изменить запись (поток А заносит данные о заказе, поток В – об оплате).
3. Вернуть измененную запись в файл.

Предположим, что клиент, которому в базе уже соответствует запись, сделал заказ и сразу оплатил его, т.е. данные о заказе и об оплате должны поступить в базу практически одновременно. Далее возможен следующий вариант развития событий.

Пусть в некоторый момент поток А обновляет данные о заказе в записи о клиенте, выполняет шаги А1 и А2, но выполнить шаг А3 (занести содержимое буфера в запись базы) не успевает вследствие завершения кванта времени.

Потоку В требуется внести сведения об оплате заказа этого же клиента. Предположим, что, когда подходит очередь потока В, он успевает сделать шаги В1 и В2, а затем прерывается. При этом в его буфере оказывается запись, где данные о заказе относятся к старому заказу, а данные об оплате – к новому.

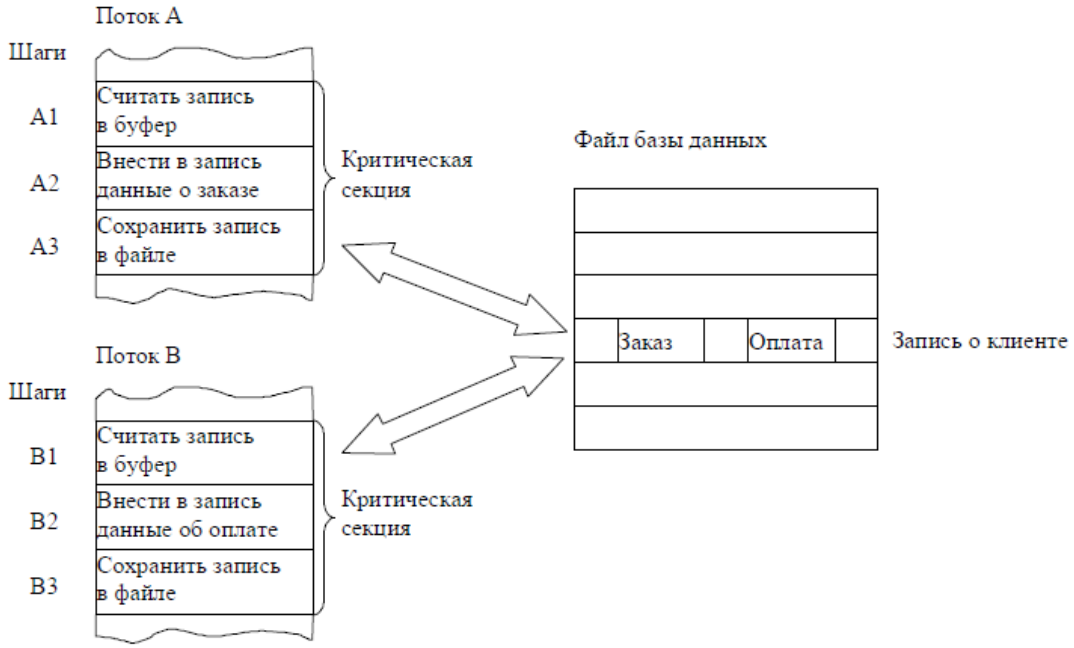
Далее поток А получает управление, выполняет шаг А3 – запись в базу содержимого своего буфера – и завершается. Вслед за ним то же проделывает поток В.

Смена содержимого буферов и базы приведена на рис. 2.12. В итоге данные о новом заказе оказались потеряны.

Приведенный вариант не является неизбежным. В силу нерегулярности возникающих ситуаций при взаимодействии процессов (в данном случае потоков) возможно и другие варианты развития событий (см. рис. 2.13). Все определяется взаимными скоростями потоков и моментами их прерывания.

Критическая секция

Важным понятием синхронизации процессов является понятие *критической секции* – части



программы, в которой осуществляется доступ к разделяемым данным.

Рисунок 2.11 – Возникновение гонок при доступе к разделяемым данным

Шаги	Буфер А	Буфер В	Запись о клиенте
			Заказ old, оплата old
А1	Заказ old, оплата old		
А2	Заказ new, оплата old		Заказ old, оплата old
В1		Заказ old, оплата old	
В2		Заказ old, оплата new	Заказ old, оплата old
А3	Заказ new, оплата old		Заказ new, оплата old
В3		Заказ old, оплата new	Заказ old, оплата new

Рисунок 2.12 – Пошаговое выполнение и результаты потоков в ситуации гонок

Эти данные, соответственно, называются критическими данными; при несогласованном их изменении могут возникнуть нежелательные эффекты. В приведенном выше

примере гонок такими данными являются записи файла базы данных. Критические секции каждого из потоков отмечены на рис. 2.11. В общем случае в разных потоках критическая секция состоит из разных последовательностей команд.

Чтобы исключить эффект гонок по отношению к некоторому ресурсу, необходимо обеспечить, чтобы в каждый момент в критической секции, связанной с этим ресурсом, находился максимум один процесс. Этот прием называют взаимным исключением.

Средства синхронизации процессов и потоков

Для синхронизации процессов, порождаемых прикладными программами, программист может использовать как собственные средства и приемы, так и средства операционной системы, предоставляемые в форме системных вызовов. Последние являются во многих случаях более эффективными или единственно возможными. Например, потоки разных процессов могут взаимодействовать только через посредство операционной системы. Рассмотрим ряд средств синхронизации, начиная с простейших.

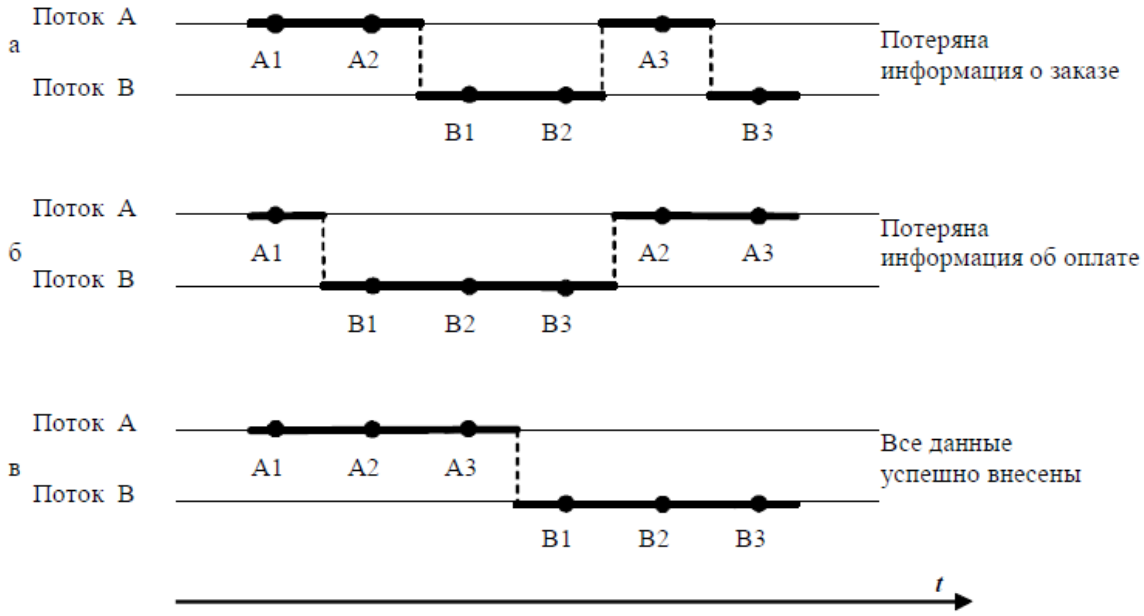


Рисунок 2.13 – Влияние относительных скоростей потоков на результат решения задачи

Простейший способ обеспечить взаимное исключение – позволить процессу, находящемуся в критической секции, **запрещать все системные вызовы.**

Недостатки метода. Опасно доверять управление системой пользовательскому процессу; он может надолго занять процессор, а при крахе процесса в критической области крах потерпит вся система, потому что системные вызовы никогда не будут разрешены. Использование блокирующих переменных. С каждым разделяемым ресурсом D связывается двоичная переменная F(D), которая принимает следующие значения:

$$F(D) = \begin{cases} 1, \text{ если ресурс свободен (то есть ни один процесс не находится в данный момент в критической секции, связанной с данным процессом);} \\ 0, \text{ если ресурс занят.} \end{cases}$$

На рис. 2.14 показан фрагмент алгоритма процесса, использующего блокирующую переменную.

Перед входом в критическую секцию процесс проверяет, свободен ли ресурс D.

Если он занят, (F(D) = 0), то проверка циклически повторяется. Если же ресурс свободен (F(D) = 1), то значение переменной F(D) устанавливается в 0 и процесс входит в критическую секцию. После того как процесс выполнит все действия с разделяемым ресурсом D, значение переменной F(D) снова устанавливается равным 1.

Если все процессы написаны с использованием вышеописанных соглашений, то взаимное исключение гарантируется. При этом процессы могут быть прерваны операционной системой в любой момент и в любом месте, в том числе в критической секции.

Ограничение на прерывания единственное: нельзя прерывать процесс между выполнением операций проверки и установки блокирующей переменной, т.е. операция проверки и установки блокирующей переменной должна быть неделимой. Поясним это.

Пусть в результате проверки переменной процесс определил, что ресурс свободен, но сразу после этого, не успев установить переменную в 0, был прерван. За время его приостановки другой процесс занял ресурс, вошел в свою критическую секцию, но также был прерван, не завершив работы с разделяемым ресурсом. В итоге первый процесс полагает ресурс свободным, тогда как второй его занял. При возврате управления первому процессу он, считая ресурс свободным, установил признак занятости и начал выполнять свою критическую секцию. Таким образом, был нарушен принцип взаимного исключения, что потенциально может привести к нежелательным последствиям. Во избежание таких ситуаций в системе команд машины желательно иметь единую операцию “проверка-установка логической переменной” (например, это возможно реализовать посредством команд BTC, BTR и BTS процессоров x86) или же реализовывать системными средствами соответствующие программные примитивы (базовые функции ОС), которые бы запрещали прерывания на протяжении всей операции проверки и установки.

Недостатки метода. В течение времени, когда один процесс находится в критической секции, другой процесс, которому требуется тот же ресурс, будет выполнять рутинные действия по опросу блокирующей переменной, бесполезно тратя процессорное время.

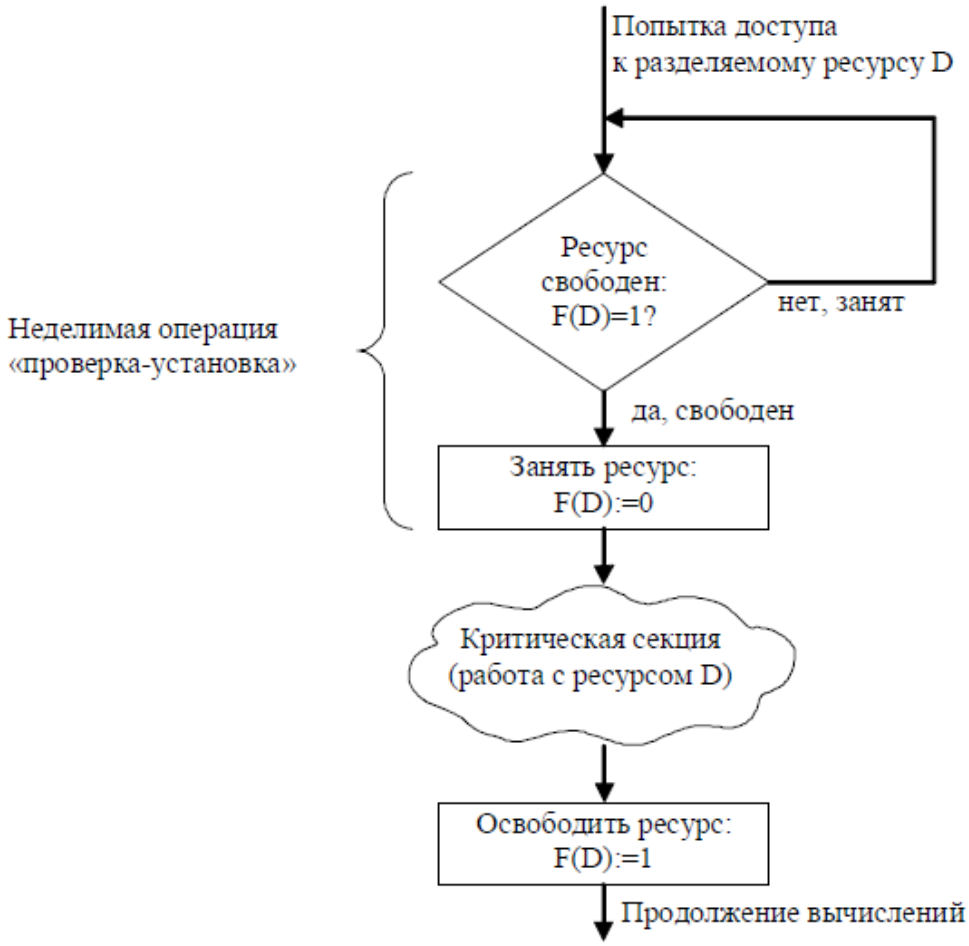


Рисунок 2.14 – Реализация критических секций с использованием блокирующих переменных

Специальные системные вызовы для работы с критическими секциями позволяют устранить такие ситуации простоя. В разных операционных системах соответствующие функции реализуется по-разному, но действия их и использование аналогичны. Если ресурс занят, то нуждающийся в нем процесс не выполняет циклический опрос, а вызывает системную функцию, переводящую его (процесс) в состояние ожидания освобождения ресурса. Процесс, который использует ресурс, после выхода из критической секции выполняет системную функцию, переводящую первый процесс, ожидающий ресурса, в состояние готовности.

На рисунке 2.15 показана реализация взаимного исключения при синхронизации потоков с помощью таких функций в ОС Windows NT.

Каждый из потоков, претендующих на доступ к разделяемому ресурсу D, должен содержать два системных вызова – для входа в критическую секцию с занятием ресурса, освобожденного другим потоком, и выхода из нее с освобождением ресурса для другого потока.

Поток, претендующий на доступ к критическим данным, для входа в критическую секцию выполняет системный вызов EnterCriticalSection. В рамках этого вызова выполняется

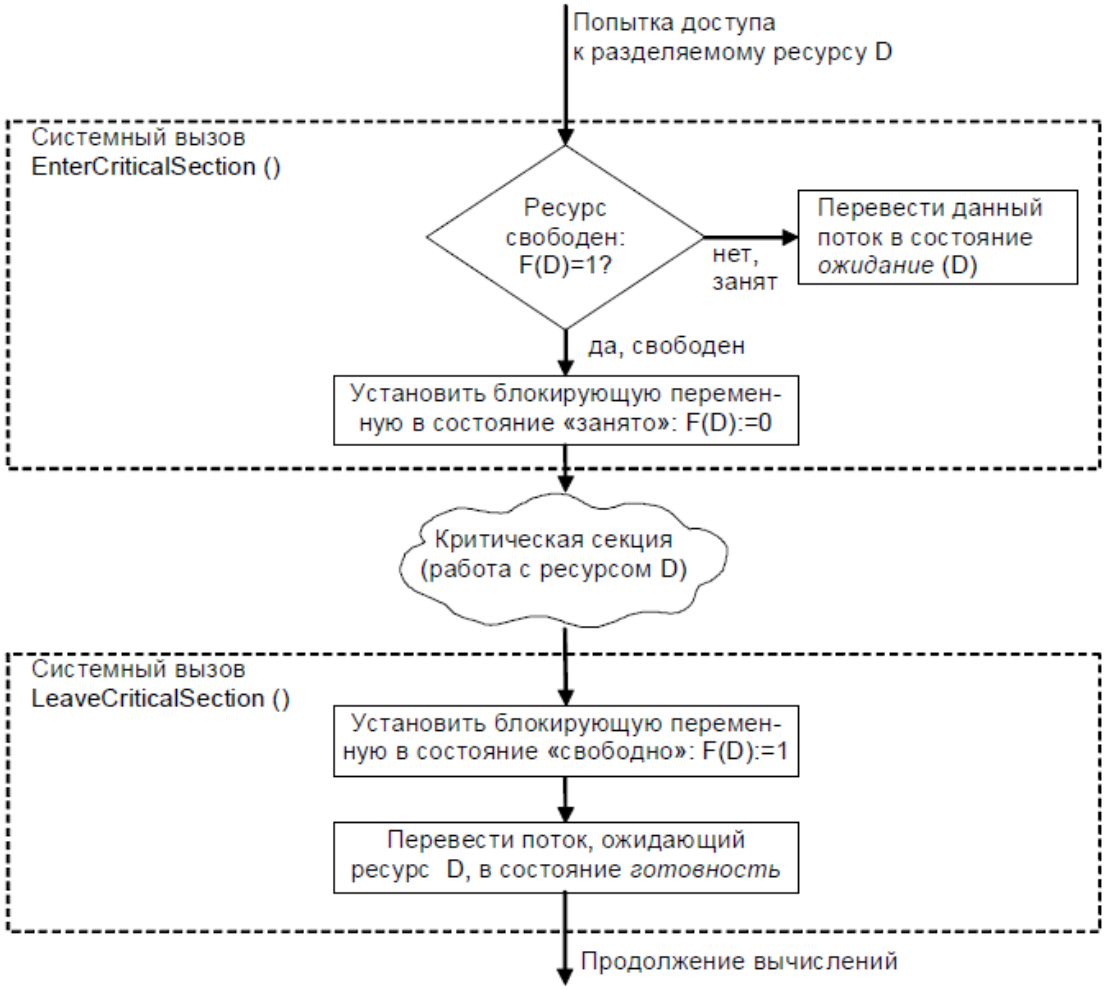


Рисунок 2.15 – Реализация взаимного исключения с использованием системных функций проверка блокирующей переменной.

В случае занятости ресурса поток переводится в состояние ожидания и делается отметка о том, что он должен быть активизирован по освобождении ресурса (поток ставится в очередь ожидающих освобождения ресурса). Если ресурс свободен, он занимается (F(D):=0), делается отметка о его принадлежности данному потоку и поток продолжает работу.

Поток, который использует ресурс, после выхода из критической секции должен выполнить системный вызов LeaveCriticalSection. В результате отмечается, что ресурс свободен (F(D):=1), и первый поток из очереди ожидающих ресурс переводится в состояние готовности.

Специфика метода. Если объем работы в критической секции небольшой и вероятность в скором доступе к ресурсу велика, то экономнее окажется метод блокирующих переменных (за счет затрат на вызов функций). Семафоры Дейкстры – обобщение метода блокирующих

переменных. Вводятся два новых примитива. В абстрактной форме эти примитивы, традиционно обозначаемые P и V, оперируют над целыми неотрицательными переменными, называемыми семафорами. Пусть S – такой семафор. Операции определяются следующим образом.

V(S): переменная S увеличивается на 1 одним неделимым действием; выборка,

инкремент и запоминание не могут быть прерваны, и к S нет доступа другим процессами во время выполнения этой операции.

P(S): уменьшение S на 1, если это возможно. Если S=0, то невозможно уменьшить S и остаться в области целых неотрицательных значений. В этом случае процесс, вызывающий P-операцию, ждет, пока это уменьшение станет возможным. Успешная проверка и уменьшение также является неделимой операцией. В частном случае, когда семафор S может принимать только значения 0 и 1, он превращается в блокирующую переменную. Операция P заключает в себе потенциальную возможность перехода процесса, который ее выполняет, в состояние ожидания, в то время как V-операция может при некоторых обстоятельствах активизировать другой процесс, приостановленный операцией P.

Рассмотрим использование семафоров на классическом примере взаимодействия двух выполняющихся в режиме мультипрограммирования процессов, один из которых пишет данные в буферный пул, а другой считывает их из буферного пула (рис.2.16). Пусть буферный пул состоит из N буферов, каждый из которых может содержать одну запись. Процесс-писатель должен приостанавливаться, когда все буферы оказываются занятыми, и активизироваться при освобождении хотя бы одного буфера. Напротив, процесс-читатель приостанавливается, когда все буферы пусты, и активизируется при появлении хотя бы одной записи. Введем два семафора: e – число пустых и f – число заполненных буферов. До

начала работы e=N, f=0. Предположим, что запись в буфер и считывание из буфера являются критическими секциями. Введем двоичный семафор b, который будем использовать для обеспечения взаимного исключения (т.е. этот семафор в данном качестве будет служить блокирующей переменной). Оба процесса после проверки доступности буферов должны выполнить проверку доступности критической секции.

Проблема тупиков.

Приведенный пример позволяет проиллюстрировать проблему синхронизации, называемую тупиками, дедлоками (deadlocks) или клинчами (clinch) и состоящую во

взаимных блокировках процессов (ряд процессов удерживает ресурсы, запрашиваемые другими процессами, и в то же время запрашивает ресурсы, удерживаемые другими).

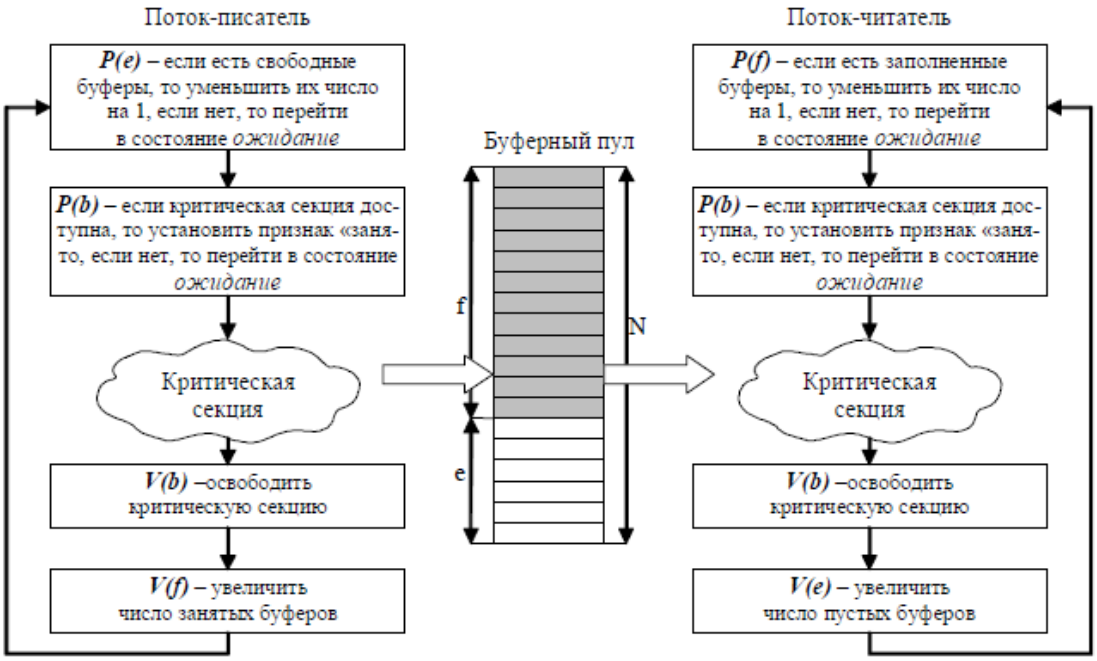


Рисунок 2.16 – Использование семафоров для синхронизации потоков

Если переставить местами операции P(e) и P(b) в программе-писателе, то при некотором стечении обстоятельств рассматриваемые два процесса могут заблокировать друг друга.

Пусть процесс-писатель первым войдет в критическую секцию и обнаружит отсутствие свободных буферов. Картина примет следующий вид.

Писатель ждет освобождения буфера, а читатель не может этого сделать, так как эти действия выполняются в ставшей недоступной критической секции.

В рассмотренном примере тупик был образован двумя процессами, но взаимно блокировать друг друга могут и большее число процессов.

Решение проблемы тупиков требует решения следующих задач:

· предотвращение тупиков,

· распознавание тупиков,

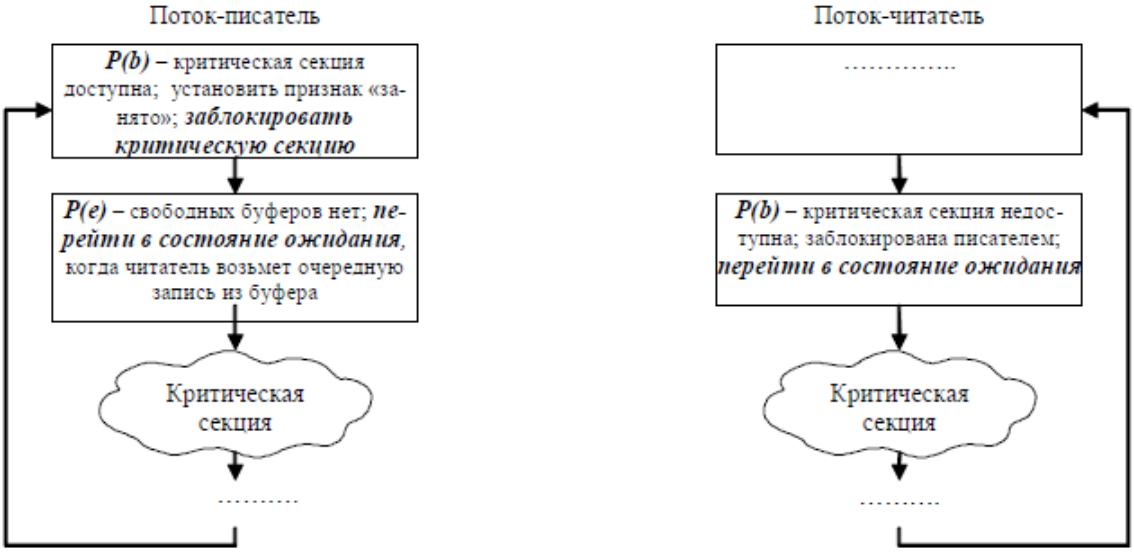
· восстановление системы после тупиков.

Тупики могут быть предотвращены на стадии написания программ, то есть программы

должны быть написаны таким образом, чтобы тупик не мог возникнуть ни при каком соотношении взаимных скоростей процессов. Второй подход к предотвращению тупиков

называется динамическим и заключается в использовании определенных правил при назначении ресурсов процессам, например, ресурсы могут выделяться в определенной последовательности, общей для всех процессов.

В некоторых случаях, когда тупиковая ситуация образована многими процессами, использующими много ресурсов, распознавание тупика является нетривиальной задачей. Существуют формальные, программно-реализованные методы распознавания тупиков, основанные на ведении таблиц распределения ресурсов и таблиц запросов к занятым ресурсам. Анализ этих таблиц позволяет обнаружить взаимные блокировки.



Если же тупиковая ситуация возникла, то не обязательно снимать с выполнения все заблокированные процессы. Можно снять только часть из них, при этом освобождаются ресурсы, ожидаемые остальными процессами; можно совершить “откат” некоторых процессов до так называемой контрольной точки, в которой запоминается вся информация, необходимая для восстановления выполнения программы с данного места. Контрольные точки расставляются в программе в местах, после которых возможно возникновение тупика.

Из всего вышесказанного ясно, что использовать семафоры нужно очень осторожно, так как одна незначительная ошибка может привести к останову системы (или по крайней мере нескольких процессов).

Для того, чтобы облегчить написание корректных программ, было предложено высокоуровневое средство синхронизации, называемое монитором. Монитор – это набор процедур, переменных и структур данных. Процессы могут вызывать процедуры монитора,

но не имеют доступа к внутренним данным монитора. Мониторы имеют важное свойство, которое делает их полезными для достижения взаимного исключения: только один процесс может быть активным по отношению к монитору. Компилятор обрабатывает вызовы процедур монитора особым образом. Обычно, когда процесс вызывает процедуру монитора, то первые несколько инструкций этой процедуры проверяют, не активен ли какой-либо другой процесс по отношению к этому монитору. Если да, то вызывающий процесс приостанавливается, пока другой процесс не освободит монитор. Таким образом, исключение входа нескольких процессов в монитор реализуется не программистом, а компилятором, что делает ошибки менее вероятными.

Универсальные объекты синхронизации

Мьютексы. Обычно используются для управления доступом к данным. Мьютекс – синхронизирующий объект как для процессов, так и для потоков. В каждый момент времени только один процесс (поток) имеет право обладания этим объектом. Суть его использования такова.

Пусть два (или более) процесса (потока) имеют необходимость в доступе к некоторому разделяемому ресурсу. Для организации взаимного исключения один из этих процессов должен посредством функции CreateMutex создать мьютекс с некоторым именем, предназначенный для пользования этим ресурсом. Остальные процессы должны знать это

имя для обращения к мьютексу.

Существует набор функций, позволяющий запросить право владения этим объектом. Одни функции этого набора прерывают работу вызвавшего их потока до момента освобождения мьютекса, другие проверяют возможность получения права обладания и при наличии такой возможности сразу получают это право, иначе возвращают управление вызвавшему их процессу.. Первым занимает мьютекс либо его создатель, либо процесс, первым вызвавший одну из функций ожидания. По завершении работы с разделяемым ресурсом мьютекс освобождается.

Фактически мьютекс – это реализация монитора в семействе WinNT.

События. Обычно используются для оповещения процессов о завершении некоторых действий. В каждой ОС реализованы свои механизмы передачи сообщений и обработки событий.

Взаимодействие процессов и потоков включает в себя, помимо синхронизации, также и передачу данных между ними. Для передачи данных обычно используются средства ОС, специально ориентированные на межпроцессные взаимодействия.

В ОС семейства WinNT для передачи данных чаще всего употребляются сообщения (при этом «полезная информация» переносится как самим сообщением, так и его параметрами – wparam и lparam). Специально зарезервированное сообщение wm_copydata позволяет передавать большие объемы данных, используя механизм отображаемых в память файлов (memory-mapped files). Этот механизм можно использовать и напрямую, оперируя функциями winAPI.

В UNIX для передачи данных используются конвейеры (при последовательном запуске программ), каналы, сигналы (эквивалент сообщений windows), а также средства, базирующиеся на протоколе TCP/IP. Последние позволяют организовать в том числе и межмашинное взаимодействие.

Проблемы взаимодействия процессов

1.1.1. Изоляция процессов и их взаимодействие

Одна из важнейших целей, которые ставятся при разработке многозадачных систем, заключается в том, чтобы разные процессы, одновременно работающие в системе, были как можно лучше изолированы друг от друга. Это означает, что процессы (в идеале) не должны ничего знать даже о существовании друг друга. Для каждого процесса ОС предоставляет ***виртуальную машину***, т.е. полный набор ресурсов, имитирующий выполнение процесса на отдельном компьютере.

Изоляция процессов, во-первых, является необходимым условием надежности и безопасности многозадачной системы. Один процесс не должен иметь возможности вмешаться в работу другого или получить доступ к его данным, ни по случайной ошибке, ни намеренно.

Во-вторых, проектирование и отладка программ чрезвычайно усложнились бы, если бы программист должен был учитывать непредсказуемое влияние других процессов.

С другой стороны, есть ситуации, когда взаимодействие необходимо. Процессы могут совместно обрабатывать общие данные, обмениваться сообщениями, ждать ответа и т.п. Система должна предоставлять в распоряжение процессов средства взаимодействия. Это не противоречит тому, что выше было сказано об изоляции процессов. Чтобы взаимодействие не привело к полному хаосу, оно должно выполняться только с помощью тех хорошо продуманных средств, которые предоставляет процессам ОС. За пределами этих средств действует изоляция процессов.

Это как граница государств – пересекать ее в произвольных местах запрещено, но должна быть хорошо обустроенная система пропускных пунктов, где легко проконтролировать соблюдение правил пересечения границы.

Понятие взаимодействия процессов включает в себя несколько видов взаимодействия, основными из которых являются:

- синхронизация процессов, т.е., упрощенно говоря, ожидание одним процессом каких-либо событий, связанных с работой других процессов;
- обмен данными между процессами.

Набор средств, предназначенных для взаимодействия процессов, часто обозначают аббревиатурой IPC (InterProcess Communication). Состав функций и методов обусловлен многолетним опытом как программистов-практиков, так и теоретиков, рассматривающих проблемы взаимодействия параллельных процессов.

Следует еще раз отметить, что проблемы взаимодействия процессов не зависят от способа реализации параллельной работы процессов в конкретной ОС. Большая часть этих проблем имеет место в равной степени и для квазипараллельной реализации, и для истинной параллельности с использованием нескольких процессоров. Средства решения проблем, правда, могут зависеть от реализации.

1.1.2. Проблема взаимного исключения процессов

Серьезная проблема возникает в ситуации, когда два (или более) процесса одновременно пытаются работать с общими для них данными, причем хотя бы один процесс изменяет значение этих данных.

Проблему часто поясняют на таком, немного условном, примере. Пусть имеется система резервирования авиабилетов, в которой одновременно работают два процесса. Процесс A обеспечивает продажу билетов, процесс B – возврат билетов. Не углубляясь в детали, будем считать, что оба процесса работают с переменной **N** – числом оставшихся билетов, причем соответствующие фрагменты программ на псевдокоде выглядят примерно так:

Процесс A:	***Процесс B:***
. . .	**. . .**
R1 := N;	**R2 := N;**
R1 := R1 - 1;	**R2 := R2 + 1;**
N := R1;	**N := R2;**
. . .	**. . .**

Представим себе теперь, что при квазипараллельной реализации процессов в ходе выполнения этих трех операторов происходит переключение процессов. В результате, в зависимости от

непредсказуемых случайностей, порядок выполнения операторов может оказаться различным, например:

1) **R1 := N; R2 := N; R2 := R2 + 1; N := R2; R1 := R1 - 1; N := R1;**
2) **R2 := N; R2 := R2 + 1; R1 := N; R1 := R1 - 1; N := R1; N := R2;**
3) **R1 := N; R1 := R1 - 1; N := R1; R2 := N; R2 := R2 + 1; N := R2;**

Ну и что? А то, что в случае 1 значение **N** в результате окажется уменьшенным на 1, в случае 2 – увеличенным на 1, и только в случае 3 значение **N**, как и положено, не изменится.

Можно привести менее экзотические примеры.

- Если два процесса одновременно пытаются отредактировать одну и ту же запись базы данных, то в результате разные поля одной записи могут оказаться несогласованными.
- Если один процесс добавляет сообщение в очередь, а другой в это время пытается взять сообщение из очереди для обработки, то он может прочесть не полностью сформированное сообщение.
- Если два процесса одновременно пытаются обратиться к диску, каждый со своим запросом, то что именно каждый из них в результате прочтет или запишет на диск – сказать трудно.

Ситуация понятна: нельзя разрешать двум процессам одновременно обращаться к одним и тем же данным, если при этом происходит изменение этих данных.

То, что мы рассматривали квазипараллельную реализацию процессов, не столь существенно. Для процессов, работающих на разных процессорах, но одновременно обращающихся к одним и тем же данным, ситуация примерно та же.

Задолго до создания многозадачных систем разработчики средств автоматики столкнулись с неприятным эффектом зависимости результата операции от случайного и непредсказуемого соотношения скоростей распространения разных сигналов в электронных схемах. Этот эффект они назвали «гонками». Мы здесь ведем речь, в сущности, о том же.
Для более четкого описания ситуации было введено понятие ***критической секции***.

Критической секцией процесса по отношению к некоторому ресурсу называется такой участок программы процесса, при прохождении которого необходимо, чтобы никакой другой процесс не находился в *своей* критической секции по отношению к *тому же* ресурсу.

В примере с билетами приведенные три оператора в каждом из процессов составляют критическую секцию этого процесса по отношению к общей переменной **N**. Алгоритм работы каждого процесса в отдельности правилен, но правильная работа двух процессов в совокупности может быть гарантирована, только если они не сунутся одновременно каждый в свою критическую секцию.

А как им это запретить?

На первый взгляд кажется, что эта проблема (ее называют проблемой ***взаимного исключения*** процессов) решается просто. Например, можно ввести булеву переменную **Free**, доступную обоим процессам и имеющую смысл «критическая область свободна». Каждый процесс перед входом в свою критическую область должен ожидать, пока эта переменная не станет истинной, как показано ниже:

Процесс A:	***Процесс B:***
. . .	**. . .**
while not Free do	**while not Free do**
;	**;**
Free := false;	**Free := false;**
(критическая секция A)	**(критическая секция B)**
Free := true;	**Free := true;**
. . .	**. . .**

В обоих процессах цикл **while** не делает ничего, кроме ожидания, пока другой процесс выйдет из своей критической секции.

А не нужно ли было что-нибудь сделать с переменной **Free** еще до запуска процессов A и B? Прежде всего, отметим, что предложенное решение использует такую неприятную вещь, как активное ожидание: процессорное время растрачивается на многократную проверку переменной **Free**. Но это полбеды.

Беда в том, что такое решение ничего не решает. Если реализовать его на практике, то «неприятности» станут реже, но не исчезнут. В первом, бесхитростном варианте программы угрожаемыми участками были критические секции обоих процессов. Теперь же уязвимый участок сузился до одной точки, отмеченной в программе каждого процесса штриховой линией. Это точка между проверкой переменной **Free** и изменением этой переменной. Если переключение процессов произойдет, когда вытесняемый процесс будет находиться именно в этой точке, то сначала в критическую секцию войдет (с полным правом на это) другой процесс, а потом, когда управление вернется к первому процессу, он без дополнительной проверки тоже войдет в свою критическую секцию.

Разработчики первых программных систем, использующих взаимодействие параллельных процессов, не сразу осознали сложность проблемы взаимного исключения. Были испробованы различные программные решения, прежде чем удалось найти такое, которое удовлетворяло трем естественным условиям:

- в любой момент времени не более, чем один процесс может находиться в критической секции;

- если критическая секция свободна, то процесс может беспрепятственно войти в нее;
- все процессы равноправны.

Попробуйте сами найти такое решение. В книгах /**Ошибка! Источник ссылки не найден.**/ и /**Ошибка! Источник ссылки не найден.**/ можно найти несколько вариантов решения с анализом их ошибок.

В конце концов правильные алгоритмы решена задачи взаимного исключения предложили сначала Деккер (его алгоритм довольно запутанный, что часто случается с первыми решениями сложных задач), затем Питерсон, чей алгоритм проще и понятнее.

Для любознательных приводим решение Питерсона. В нем используются булевы переменные **flagA, flagB**, изначально равные **false**, и переменная перечисляемого типа **turn: A..B**.

Процесс A:	***Процесс B:***
. . .	**. . .**
flagA := true;	**flagB := true;**
turn := B;	**turn := A;**
while flagB and turn = B do	**while flagA and turn = A do**
;	**;**
(критическая секция A)	**(критическая секция B)**
flagA := false;	**flagB := false;**
. . .	**. . .**

Приведенный алгоритм действительно решает проблему, однако у него есть два существенных недостатка. Во-первых, если в конкуренции за критическую секцию участвуют не два процесса, а три или более, то программа становится очень громоздкой. Во-вторых, решение Питерсона основано на использовании активного ожидания.

Еще одним направлением в реализации взаимного исключения стало включение специальных машинных команд в наборы команд новых процессоров. Поскольку опасность, как мы видели, связана с разделением по времени операций проверки и присваивания, то были предложены команды, выполняющие одновременно проверку и присваивание. Такие команды есть, например, у процессоров Pentium. С их помощью действительно можно проще реализовать взаимное исключение, но для этого все равно требуется активное ожидание.

1.1.3. Двоичные семафоры Дейкстры

Совершенно иным образом подошел к проблеме взаимного исключения великий голландский ученый Э.Дейкстра (E.Dijkstra, 1966). Он предложил использовать новый вид программных объектов – ***семафоры***. Здесь мы рассмотрим их простейший вариант – ***двоичные семафоры***, они же ***мьютексы*** (mutex, от слов MUTual EXclusion – взаимное исключение).

Двоичным семафором называется переменная S, которая может принимать значения 0 и 1 и для которой определены только две операции.

- P(S) – операция занятия (закрытия) семафора. Она ожидает, пока значение S не станет равным 1, и, как только это случится, присваивает S значение 0 и завершает свое выполнение. *Очень важно*: операция P *по определению* неделима, т.е. между проверкой и присваиванием не может вклиниться другой процесс, который бы изменил значение S.
- V(S) – операция освобождения (открытия) семафора. Она просто присваивает S значение 0.

Чем переменная-семафор отличается от обычной булевой переменной? Тем, что для нее недопустимы никакие иные операции, кроме P и V. Нельзя написать в программе **S:=1** или **if(S)then ...** , если **S** определена как семафор.

Чем операция P отличается от варианта с проверкой и присваиванием, который мы выше признали неудовлетворительным? Неделимостью. Но это «по определению», а как на практике добиться этой неделимости? Это отдельный, вполне решаемый вопрос.

Заслуга Дейкстры как раз в том, что он разделил проблему взаимного исключения на две независимые проблемы разных уровней:

- на уровне реализации: как обеспечить работу семафоров в соответствии с их определением;
- на уровне взаимодействия процессов: как написать корректно работающую программу, если в распоряжении программиста имеются семафоры.

Решать эти две задачи по отдельности легче, чем обе вместе, при этом решать их обычно должны разные люди: первую – разработчики ОС, а вторую – разработчики прикладной программы.

Рассмотрим сначала реализацию. Очевидно, функции P и V удобнее и надежнее один раз реализовать в ОС, чем каждый раз по-новому – в прикладных программах. (Названия этих функций могут в конкретных системах быть и иными, более выразительными.)

Системная функция P(S) должна проверить, свободен ли семафор S. Если свободен (S = 1), то система занимает его (S := 0) и на этом функция завершается. Если же семафор занят, то система блокирует процесс, вызвавший функцию P, и запоминает, что этот процесс блокирован по ожиданию освобождения семафора S. Таким образом, при реализации семафоров удается избежать активного ожидания.

Неделимость операции обеспечивается тем, что во время выполнения системой функции P переключение процессов запрещено. В крайнем случае, ОС имеет возможность для этого на короткое время запретить прерывания.

Системная функция V(S) – это, конечно, не просто присваивание S := 1. Кроме этого, система должна проверить, нет ли среди спящих процессов такого, который ожидает освобождения семафора S. Если такой процесс найдется, система разблокирует его, а переменная S в этом случае сохраняет значение 0 (семафор снова занят, теперь уже другим процессом).

Может ли случиться так, что несколько спящих процессов ждут освобождения одного и того же семафора? Да, так вполне может быть. Какой из этих процессов должен быть разбужен системой? С точки зрения корректности работы и соответствия определениям функций P и V – любой, но только один. С точки зрения эффективности работы – вероятно, надо разбудить самый приоритетный процесс, а в случае равенства приоритетов… ну, видимо, тот, который спит дольше.

Теперь, когда мы разобрались с реализацией семафоров, можно о ней забыть[1] и помнить только, что семафоры существуют и могут быть использованы при необходимости.

Рассмотрим теперь вторую половину задачи – использование семафоров для управления взаимодействием процессов. Как можно реализовать корректную работу процессов с критическими секциями, если использовать двоичный семафор? Да очень просто.

Процесс A:	***Процесс B:***
. . .	**. . .**
P(S);	**P(S);**
(критическая секция A)	**(критическая секция B)**
V(S);	**V(S);**
. . .	**. . .**

И все. Сложности ушли в реализацию семафоров. Надо только проследить, чтобы до начала работы процессов семафор S был открыт.

1.1.4. Средства взаимодействия процессов

Можно доказать, что использование двоичных семафоров позволяет корректно решить любые проблемы синхронизации процессов. Но вовсе не обязательно это решение окажется простым и удобным. В некоторых случаях использование семафоров должно все же сопровождаться нежелательным активным ожиданием.

За десятилетия, прошедшие после изобретения семафоров, были предложены различные средства синхронизации, более приспособленные для различных типовых задач. Рассмотрим некоторые из них.

1.1.4.1. Целочисленные семафоры

В упомянутой работе Дейкстры, помимо двоичных семафоров, принимающих значения 0 и 1, был рассмотрен также более общий тип семафоров со значениями на интервале от 0 до некоторого N.

[1] Но к экзамену вспомнить!

Функция P(S) уменьшает положительное значение семафора на 1, а при нулевом значении переходит в ожидание, как и в случае двоичного семафора. Функция V(S) увеличивает значение семафора на 1, но не более N.

Область применения целочисленных семафоров несколько иная, чем у двоичных. Целочисленные семафоры применяются в задачах выделения ресурсов из ограниченного запаса. Величина N характеризует общее количество имеющихся единиц ресурса, а текущее значение переменной – количество свободных единиц. При запросе ресурса процесс вызывает функцию V(S), при освобождении – P(S).

Для целочисленных семафоров иногда удобно использовать модифицированную функцию V(S, k), вторым параметром которой является число одновременно запрашиваемых единиц ресурса. Такая функция блокирует процесс, если значение семафора меньше k.

1.1.4.2. Семафоры с множественным ожиданием

Возможна ситуация, когда процесс может выбрать один из нескольких путей дальнейшей работы, но на каждом пути он может быть заблокирован закрытым семафором. Разумно было бы ждать освобождения любого из семафоров и только тогда выбрать свободный путь. Но как это сделать? Вызвав P(S) для одного из семафоров, процесс обречен ждать освобождения именно этого семафора, а не любого из имеющихся.

Житейская ситуация: покупатель в супермаркете, выбирающий, к какой из касс занять очередь. Хорошо бы угадать очередь, которая пройдет быстрее…
Функция ***множественного ожидания*** P(S_1, S_2, … S_n) позволяет указать в качестве параметров несколько двоичных семафоров (или массив семафоров). Если хотя бы один из семафоров свободен, функция занимает его, в противном случае она ждет освобождения любого из семафоров.

Другой, не менее полезный вариант множественного ожидания, это ожидание момента, когда *все* указанные семафоры окажутся свободны. Это означает, что процесс может работать дальше только в том случае, если одновременно выполнены несколько условий, каждое из которых задано в виде двоичного семафора.

1.1.4.3. Сигналы

Сигнал – это нечто, что может быть послано процессу системой или другим процессом. С сигналом не связано никакой информации, кроме номера (кода), указывающего, какой именно тип сигнала посылается. При получении сигнала процесс прерывает свою текущую работу и переходит на выполнение функции, определенной как обработчик сигналов данного типа.

Таким образом, сигналы сильно похожи на прерывания, но только высокоуровневые, управляемые системой, а не аппаратурой.

Механизм сигналов позволяет решить, например, проблему критической секции иным способом, чем семафоры.

Подумайте самостоятельно, как это можно сделать.

1.1.4.4. Сообщения

Сообщения также посылаются процессу системой или другим процессом, однако отличаются от сигналов в двух отношениях.

Во-первых, сообщения не прерывают работу процесса-получателя. Вместо этого они становятся в очередь сообщений. Процесс должен сам вызвать функцию приема сообщения. Если очередь пуста, эта функция блокирует процесс до получения какого-нибудь сообщения.

Во-вторых, с сообщением, в отличие от сигнала, может быть связана информация, передаваемая получателю. Таким образом, сообщения – это средство не только синхронизации, но и обмена данными между процессами.

1.1.4.5. Общая память

Поговорим теперь еще об обмене данными. Самым простым и естественным способом такого обмена представляется возможность совместного доступа двух или более процессов к общей области памяти. Но поскольку обычно ОС стремится, наоборот, надежно разделить память разных процессов, то для выделения обшей памяти нужны специальные системные средства.

Общая память служит только средством обмена данными, но никак не решает проблем синхронизации. Участки программы, где происходит работа с общей памятью, часто следует рассматривать как критические секции и защищать семафорами.

1.1.4.6. Программные каналы

Другое часто используемое средство обмена данными – программный канал (pipe; иногда переводится как «трубопровод»). В этом случае для выполнения обмена используются не команды чтения/записи в память, а функции чтения/записи в файл. Программный канал «притворяется файлом», для работы с ним используются те же операции, что для последовательного доступа к файлу: открытие, чтение, запись, закрытие. Однако источником читаемых данных служит не файл на диске, а процесс, выполняющий запись «в другой конец трубы». Данные, записанные одним процессом, но пока не прочитанные другим, хранятся в системном буфере. Если же процесс пытается прочесть данные, которые пока не записаны другим процессом, то процесс-читатель блокируется до получения данных.

1.1.5. Проблема тупиков

Согласно определению из /**Ошибка! Источник ссылки не найден.**/, тупик – это состояние, в котором «*некоторые процессы заблокированы в результате таких запросов на ресурсы, которые*

никогда *не могут быть удовлетворены, если не будут предприняты* чрезвычайные *системные меры*».

Как это прикажете понимать?

Прежде всего, давайте отметим, что процессу, действующему в одиночку, не под силам загнать приличную ОС в тупик. Требования процесса не будут удовлетворены, только если они превышают то, что есть у системы. Скажем, процесс требует 500 Мб оперативной памяти, когда у системы есть всего-то 256 Мб. Ну, так в этом случае процесс будет не блокирован, а беспощадно убит системой.

Иное дело, если в деле замешаны два или более процессов. Согласно другому определению, данному в /**Ошибка! Источник ссылки не найден.**/, «*Группа процессов находится в тупиковой ситуации, если каждый процесс из группы ожидает события, которое может вызвать только другой процесс из той же группы*».

Рассмотрим такой пример. Пусть каждый из процессов A и B собирается работать с двумя файлами, F1 и F2, причем не намерен разделять эти файлы с другим процессом. Программы же процессов слегка различаются, а именно:

Процесс A:	***Процесс B:***
. . .	**. . .**
Открыть(F1);	**Открыть(F2);**
Открыть(F2);	**Открыть(F1);**
(работа процесса A с файлами);	**(работа процесса B с файлами);**
Закрыть(F1);	**Закрыть(F1);**
Закрыть(F2);	**Закрыть(F2);**
. . .	**. . .**

В этой ситуации все может пройти благополучно. Пусть, например, процесс A успеет открыть оба файла к тому моменту, когда процесс B попытается открыть F2. Эта попытка не увенчается успехом: процесс B либо будет заблокирован до освобождения файлов, либо получит сообщение об ошибке при открытии файла и, если процесс умный, через какое-то время попытается еще раз. В конце концов оба процесса получат требуемые ресурсы (в данном случае открытые файлы), хотя и не оба сразу.

Совсем иное будет дело, если A успеет открыть только F1, после чего B откроет F2. Тут-то и получится тупик. Процесс A хочет открыть файл F2, но не сможет этого сделать раньше, чем B закроет этот файл. Но B не закроет F2 до того, как сумеет открыть файл F1, который занят процессом A. Каждый из процессов захватил один из ресурсов и не собирается его отдавать раньше, чем получит другой. Ситуация «двух баранов на мосту».

Подчеркнем: тупик – это не просто блокировка процесса, когда необходимый ему ресурс занят. Занят – ну и что, со временем, авось, освободится. Тупик – это ***взаимная блокировка***, из которой нет выхода.

Еще пример. Пусть в системе имеется 100 Мб памяти, доступной для процессов. Процесс А при своем старте занимает 40 Мб, но позднее на короткое время требует еще 30 Мб, после чего завершается, освобождая всю память. Процесс В ведет себя точно таким же образом.

Каждый из процессов по отдельности не требует у системы ничего невозможного. В то же время понятно: если оба процесса сумеют стартовать и начать параллельную работу, то ни один из них никогда не получит дополнительные 30 Мб. Тупик.

Совсем грубый пример. Процесс А на каком-то этапе работы ждет сообщения от процесса В, после чего собирается послать ответное сообщение. В то же время процесс В ждет сообщения от А, чтобы потом ответить на него. Тупик неизбежен. В данном случае, в отличие от предыдущих, возникновение тупика связано с явной ошибкой в логике программы.

Может ли помочь в борьбе с тупиками использование семафоров и других подобных средств? Вряд ли, разве что в редких случаях. Семафор – это, скорее, дополнительный тормоз для процесса. Вещь полезная, но не способствующая продвижению.

Все известные способы борьбы с тупиками можно разделить на три группы:

- исключение возможности тупиков путем анализа исходного текста программ;
- предотвращение возникновения тупиков при работе ОС;
- ликвидация возникших тупиков.

Что касается анализа текста – это, безусловно, нужная вещь, хотя и непростая. Определить по тексту программ процессов, могут ли они зайти в тупик – сложная задача. К тому же, если и могут, то совсем не обязательно зайдут, все может зависеть от конкретных исходных данных и от временны*х* соотношений. Но главное – для анализа исходного текста программ нужно иметь в своем распоряжении этот текст. Реально ли это? Только в некоторых ситуациях. Например, при разработке встроенной системы исходные тексты всех прикладных программ обычно доступны разработчику ОС. Конечно, в этом случае анализ на возможность тупиков просто необходим. Другой пример – разработка сложного многопроцессного приложения, когда разработчик должен хотя бы выявить возможность взаимной блокировки между «своими» процессами.

Если тексты недоступны, то можно попытаться предотвращать тупики уже в ходе работы программ, отслеживая их запросы на ресурсы и блокируя либо прекращая те процессы, которые, по-видимому, «лезут в тупик».

Самый грубый из подобных подходов заключается в том, что каждый процесс должен захватывать все необходимые ему ресурсы сразу, при старте, после чего он может либо удерживать ресурсы в

течение всего времени работы, либо освобождать ресурсы, которые больше не нужны. Такой подход, безусловно, в корне исключает возможность тупиков. К сожалению, на практике он почти исключает и многозадачную работу: многие ресурсы необходимы для работы большинства процессов (например, диски, принтер, системные файлы), поэтому пока один процесс удерживает все ресурсы, другие не смогут даже стартовать.

Чуть получше ***алгоритм нумерованных ресурсов***. Он заключается в том, что все ресурсы, имеющиеся в системе, нумеруются целыми числами в произвольном порядке (хотя, вероятно, для повышения эффективности лучше всего пронумеровать их в порядке возрастания дефицитности ресурса). Далее применяется простое правило: запрос процесса на выделение ему ресурса с номером K удовлетворяется только в том случае, если процесс в данный момент не владеет никаким другим ресурсом с номером $N \geq K$. Другими словами, запрос ресурсов следует выполнять только в порядке возрастания номеров. Нетрудно показать, что это правило является достаточным условием отсутствия тупиков. Но это условие слишком ограничивающее, оно отсекает много ситуаций, когда тупик на самом деле не возник бы.

Наиболее изящен ***алгоритм банкира***, предложенный тем же Дейкстрой. В нем предполагается, что каждый процесс при старте должен объявить системе, на какие ресурсы и в каком максимальном количестве он может в будущем претендовать. Далее, вводится понятие ***безопасного*** (в отношении тупиков) ***состояния системы***. Текущее состояние, в котором имеется набор процессов, каждый из которых владеет некоторыми ресурсами, считается безопасным, если имеющихся в наличии свободных ресурсов достаточно для того, чтобы ОС смогла в определенной последовательности удовлетворить максимальные запросы каждого процесса.

Это определение проще понять, если рассмотреть, как проверяется безопасность заданного состояния. Согласно алгоритму, среди имеющихся процессов ищется такой, чьи максимальные заявленные запросы система может удовлетворить, используя имеющиеся свободные ресурсы. Если хотя бы один такой процесс найден, то он вычеркивается из общего списка и все ресурсы, которые он уже успел занять, считаются свободными (т.е. считается, что система смогла завершить этот процесс). Далее, при увеличившихся свободных ресурсах, ищется следующий процесс, чьи запросы система может удовлетворить, и т.д. Если в результате удается вычеркнуть все процессы, то анализируемое состояние системы является безопасным.

Сам же алгоритм банкира можно теперь сформулировать очень просто: любой запрос процесса на выделение ему дополнительных ресурсов должен удовлетворяться только в том случае, если состояние, в которое перейдет система после этого выделения, будет безопасным.

Название алгоритма навеяно, видимо, образом осторожного банкира, который выдает кредит только в том случае, если после этого сможет выполнить все свои обязательства даже в худшем случае. Явно не российский банкир.

Хотя данный алгоритм красив и логичен, у него есть, по меньшей мере, два недостатка. Во-первых, в современных ОС не принято требовать от процессов, чтобы те заранее объявляли свои будущие потребности в ресурсах. Во-вторых, выполнять проверку безопасности состояния при каждом запросе любого из процессов – слишком трудоемкое удовольствие.

Рассмотрим, наконец, третий подход – ликвидацию уже возникших тупиков, без попыток предотвратить их возникновение. В книге этот подход назван «алгоритмом страуса».

Здесь, прежде всего, возникает вопрос: как убедиться, что система действительно в тупике? Внешним признаком этого является длительное отсутствие какой-либо активности двух или более процессов. Но это недостоверный признак, процессы могли просто надолго задуматься над каким-нибудь трудоемким вычислением. Есть алгоритмы, которые анализируют текущее состояние процессов и ресурсов, наличие заблокированных запросов, и на этой основе ставят диагноз тупика. В принципе, такой алгоритм мог бы быть встроен в ОС. Однако в литературе нет сведений о том, чтобы это было осуществлено на практике. Обычно полагаются на волевое решение оператора или администратора системы.

Но пусть даже точно известно, что тупик есть. Как можно его устранить? Как «растащить баранов с моста»? Как правило, для этого применяется радикальное решение: принудительно прекратить один из тупиковых процессов (сбросить одного барана в реку). Если не помогло – утопить следующего барана. И т.д.

В литературе рассматриваются и более гуманные, но сложные способы. В принципе, можно регулярно запоминать состояние всех или только наиболее ответственных процессов в определенных контрольных точках или через определенные периоды времени. Тогда, вместо полного прекращения процесса, его можно вернуть к одной из последних контрольных точек и придержать там, пока другой баран не перейдет через реку.

Завершая рассмотрение проблемы тупиков, следует признать, что в настоящее время ее практическое значение значительно меньше, чем хотелось бы теоретикам. Во-первых, совсем не легко загнать в тупик современную ОС, работающую на компьютере с огромными ресурсами. В примере мы рассмотрели тупик, возникший из-за 100 Мб памяти, но если учесть еще несколько гигабайт, которые можно использовать в файле подкачки, то запросы процессов должны быть уж очень велики, чтобы привести к тупику. А, скажем, такое устройство, как принтер, в современных ОС вообще не может стать причиной тупика, т.к. система не отдает его во владение ни одному процессу даже на время.

Во-вторых, хотя тупик в принципе остается возможным, пользователь вряд ли даже заметит его. Скорее, он скажет «Опять Windows зависла!» и перезагрузит систему.

Предотвращение тупиков остается действительно важным, например, при разработке встроенной системы, которая должна длительное время безотказно работать без вмешательства человека.