.Цели и средства синхронизации

Существует достаточно обширный класс средств операционной системы, с помощью которых обеспечивается взаимная синхронизация процессов и потоков. Потребность в синхронизации потоков возникает только в мультипрограммной операционной системе и связана с совместным использованием аппаратных и информационных ресурсов вычислительной системы. Синхронизация необходима для исключения гонок и тупиков при обмене данными между потоками, разделении данных, при доступе к процессору и устройствам ввода-вывода.

Во многих операционных системах эти средства называются средствами межпроцессного взаимодействия — Inter Process Communications (IPC), что отражает историческую первичность понятия «процесс» по отношению к понятию «поток». Обычно к средствам IPC относят не только средства межпроцессной синхронизации, но и средства межпроцессного обмена данными.

Выполнение потока в мультипрограммной среде всегда имеет асинхронный характер. Очень сложно с полной определенностью сказать, на каком этапе выполнения будет находиться процесс в определенный момент времени. Даже в однопрограммном режиме не всегда можно точно оценить время выполнения задачи. Это время во многих случаях существенно зависит от значения исходных данных, которые влияют на количество циклов, направления разветвления программы, время выполнения операций ввода-вывода и т. п. Так как исходные данные в разные моменты запуска задачи могут быть разными, то и время выполнения отдельных этапов и задачи в целом является весьма неопределенной величиной.

Еще более неопределенным является время выполнения программы в мультипрограммной системе. Моменты прерывания потоков, время нахождения их в очередях к разделяемым ресурсам, порядок выбора потоков для выполнения — все эти события являются результатом стечения многих обстоятельств и могут быть интерпретированы как случайные. В лучшем случае можно оценить вероятностные характеристики вычислительного процесса, например вероятность его завершения за данный период времени.

Таким образом, потоки в общем случае (когда программист не предпринял специальных мер по их синхронизации) протекают независимо, асинхронно друг другу. Это справедливо как по отношению к потокам одного процесса, выполняющим общий программный код, так и по отношению к потокам разных процессов, каждый из которых выполняет собственную программу.

Любое взаимодействие процессов или потоков связано с их *синхронизацией,* которая заключается в согласовании их скоростей путем приостановки потока до наступления некоторого события и последующей его активизации при наступлении этого события. Синхронизация лежит в основе любого взаимодействия потоков, связано ли это взаимодействие с разделением ресурсов или с обменом данными. Например, поток-получатель должен обращаться за данными только после того, как они помещены в буфер потоком-отправителем. Если же поток-получатель обратился к данным до момента их поступления в буфер, то он должен быть приостановлен.

При совместном использовании аппаратных ресурсов синхронизация также совершенно необходима. Когда, например, активному потоку требуется доступ к последовательному порту, а с этим портом в монопольном режиме работает другой поток, находящийся в

данный момент в состоянии ожидания, то ОС приостанавливает активный поток и не активизирует его до тех пор, пока нужный ему порт не освободится. Часто нужна также синхронизация с событиями, внешними по отношению к вычислительной системе, например реакции на нажатие комбинации клавиш Ctrl+C.

Ежесекундно в системе происходят сотни событий, связанных с распределением и освобождением ресурсов, и ОС должна иметь надежные и производительные средства, которые бы позволяли ей синхронизировать потоки с происходящими в системе событиями.

Для синхронизации потоков прикладных программ программист может использовать как собственные средства и приемы синхронизации, так и средства операционной системы. Например, два потока одного прикладного процесса могут координировать свою работу с помощью доступной для них обоих глобальной логической переменной, которая устанавливается в единицу при осуществлении некоторого события, например выработки одним потоком данных, нужных для продолжения работы другого. Однако во многих случаях более эффективными или даже единственно возможными являются средства синхронизации, предоставляемые операционной системой в форме системных вызовов. Так, потоки, принадлежащие разным процессам, не имеют возможности вмешиваться каким-либо образом в работу друг друга. Без посредничества операционной системы они не могут приостановить друг друга или оповестить о произошедшем событии. Средства синхронизации используются операционной системой не только для синхронизации прикладных процессов, но и для ее внутренних нужд.

Обычно разработчики операционных систем предоставляют в распоряжение прикладных и системных программистов широкий спектр средств синхронизации. Эти средства могут образовывать иерархию, когда на основе более простых средств строятся более сложные, а также быть функционально специализированными, например средства для синхронизации потоков одного процесса, средства для синхронизации потоков разных процессов при обмене данными и т. д. Часто функциональные возможности разных системных вызовов синхронизации перекрываются, так что для решения одной задачи программист может воспользоваться несколькими вызовами в зависимости от своих личных предпочтений.

Необходимость синхронизации и гонки

Пренебрежение вопросами синхронизации в многопоточной системе может привести к неправильному решению задачи или даже к краху системы. Рассмотрим, например (рис. 4.16), задачу ведения базы данных клиентов некоторого предприятия. Каждому клиенту отводится отдельная запись в базе данных, в которой среди прочих полей имеются поля Заказ и Оплата. Программа, ведущая базу данных, оформлена как единый процесс, имеющий несколько потоков, в том числе поток А, который заносит в базу данных информацию о заказах, поступивших от клиентов, и поток В, который фиксирует в базе данных сведения об оплате клиентами выставленных счетов. Оба эти потока совместно работают над общим файлом базы данных, используя однотипные алгоритмы, включающие три шага.

1. Считать из файла базы данных в буфер запись о клиенте с заданным идентификатором.

2. Внести новое значение в поле Заказ (для потока А) или Оплата (для потока В).

3. Вернуть модифицированную запись в файл базы данных.

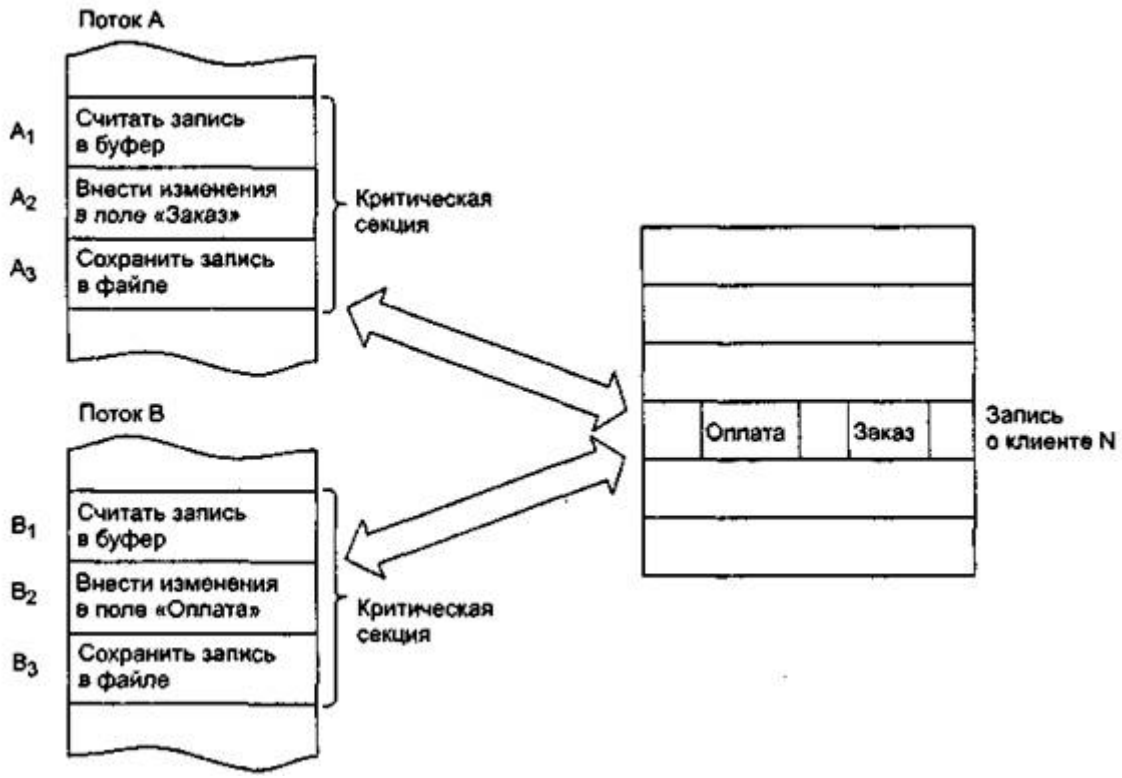

Рис. 4.16. Возникновение гонок при доступе к разделяемым данным

Обозначим соответствующие шаги для потока А как Al, А2 и А3, а для потока В как В1, В2 и В3. Предположим, что в некоторый момент поток А обновляет поле Заказ записи о клиенте N. Для этого он считывает эту запись в свой буфер (шаг А1), модифицирует значение поля Заказ (шаг А2), но внести запись в базу данных (шаг А3) не успевает, так как его выполнение прерывается, например, вследствие завершения кванта времени.

Предположим также, что потоку В также потребовалось внести сведения об оплате относительно того же клиента N. Когда подходит очередь потока В, он успевает считать запись в свой буфер (шаг В1) и выполнить обновление поля Оплата (шаг В2), а затем прерывается. Заметим, что в буфере у потока В находится запись о клиенте N, в которой поле Заказ имеет прежнее, не измененное значение.

Когда в очередной раз управление будет передано потоку А, то он, продолжая свою работу, запишет запись о клиенте N с модифицированным полем Заказ в базу данных (шаг А3). После прерывания потока А и активизации потока В последний запишет в базу данных поверх только что обновленной записи о клиенте N свой вариант записи, в которой обновлено значение поля Оплата. Таким образом, в базе данных будут зафиксированы сведения о том, что клиент N произвел оплату, но информация о его заказе окажется потерянной (рис. 4.17, а).

Сложность проблемы синхронизации кроется в нерегулярности возникающих ситуаций. Так, в предыдущем примере можно представить и другое развитие событий: могла быть потеряна информация не о заказе, а об оплате (рис. 4.17, *б)*

или, напротив, все исправления были успешно внесены (рис. 4.17, *в).* Все определяется взаимными скоростями потоков и моментами их прерывания. Поэтому отладка взаимодействующих потоков является сложной задачей. Ситуации, подобные той, когда

два или более потоков обрабатывают разделяемые данные и конечный результат зависит от соотношения скоростей потоков, называются *гонками.*

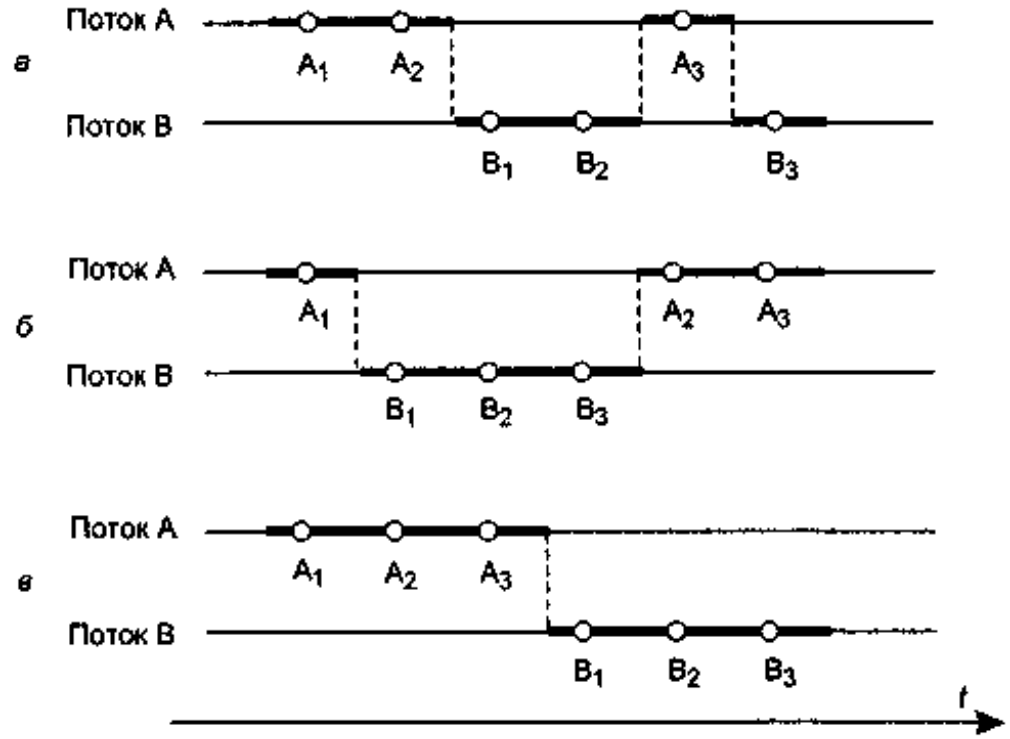

Рис. 4.17. Влияние относительных скоростей потоков на результат решения задачи

Критическая секция

Важным понятием синхронизации потоков является понятие «критической секции» программы. *Критическая секция* — это часть программы, результат выполнения которой может непредсказуемо меняться, если переменные, относящиеся к этой части программы, изменяются другими потоками в то время, когда выполнение этой части еще не завершено. Критическая секция всегда определяется по отношению к определенным *критическим данным,* при несогласованном изменении которых могут возникнуть нежелательные эффекты. В предыдущем примере такими критическими данными являлись записи файла базы данных. Во всех потоках, работающих с критическими данными, должна быть определена критическая секция. Заметим, что в разных потоках критическая секция состоит в общем случае из разных последовательностей команд.

Чтобы исключить эффект гонок по отношению к критическим данным, необходимо обеспечить, чтобы в каждый момент времени в критической секции, связанной с этими данными, находился только один поток. При этом неважно, находится этот поток в активном или в приостановленном состоянии. Этот прием называют *взаимным исключением.* Операционная система использует разные способы реализации взаимного исключения. Некоторые способы пригодны для взаимного исключения при вхождении в критическую секцию только потоков одного процесса, в то время как другие могут обеспечить взаимное исключение и для потоков разных процессов.

Самый простой и в то же время самый неэффективный способ обеспечения взаимного исключения состоит в том, что операционная система позволяет потоку запрещать любые прерывания на время его нахождения в критической секции. Однако этот способ практически не применяется, так как опасно доверять управление системой

пользовательскому потоку — он может надолго занять процессор, а при крахе потока в критической секции крах потерпит вся система, потому что прерывания никогда не будут разрешены.

2. Механизмы синхронизации.

Блокирующие переменные

Для синхронизации потоков одного процесса прикладной программист может использовать глобальные *блокирующие переменные.* С этими переменными, к которым все потоки процесса имеют прямой доступ, программист работает, не обращаясь к системным вызовам ОС.

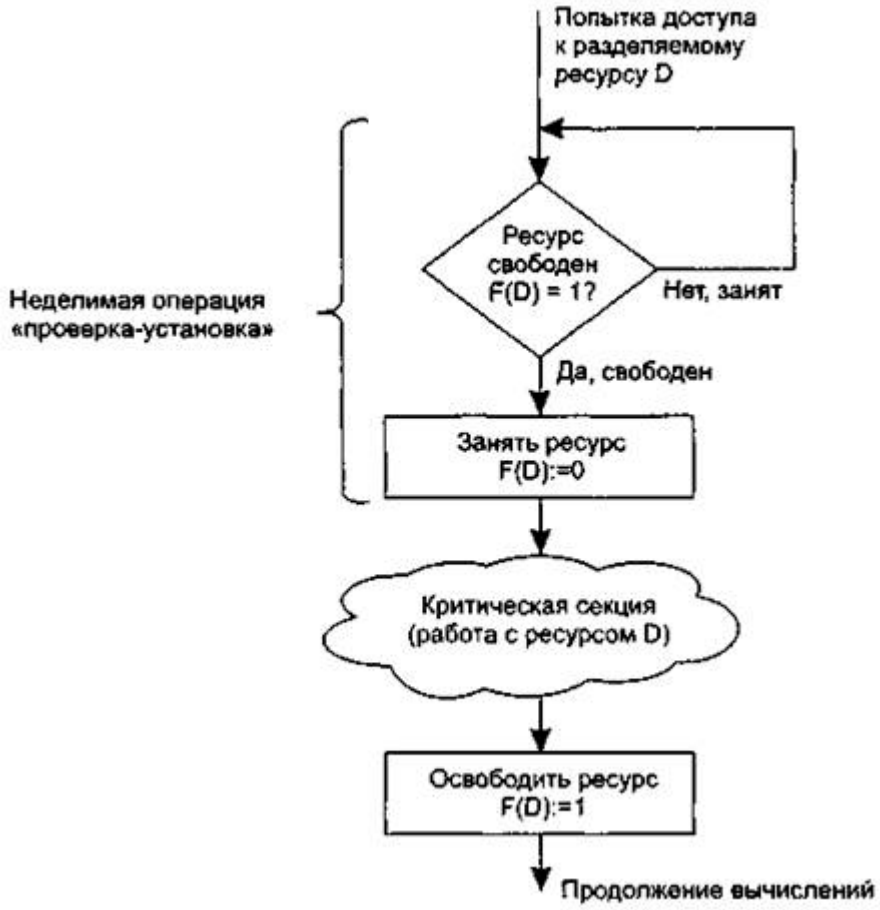

Рис. 4.18. Реализация критических секций с использованием блокирующих переменных

Каждому набору критических данных ставится в соответствие двоичная переменная, которой поток присваивает значение 0, когда он входит в критическую секцию, и значение 1, когда он ее покидает. На рис. 4.18 показан фрагмент алгоритма потока, использующего для реализации взаимного исключения доступа к критическим данным D блокирующую переменную F(D). Перед входом в критическую секцию поток проверяет, не работает ли уже какой-нибудь поток с данными D. Если переменная F(D) установлена в 0, то данные заняты и проверка циклически повторяется. Если же данные свободны (F(D) = 1), то значение переменной F(D) устанавливается в 0 и поток входит в критическую секцию. После того как поток выполнит все действия с данными D, значение переменной F(D) снова устанавливается равным 1.

Блокирующие переменные могут использоваться не только при доступе к разделяемым данным, но и при доступе к разделяемым ресурсам любого вида.

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

Однако следует заметить, что одно ограничение на прерывания все же имеется. Нельзя прерывать поток между выполнением операций проверки и установки блокирующей переменной. Поясним это. Пусть в результате проверки переменной поток определил, что ресурс свободен, но сразу после этого, не успев установить переменную в 0, был прерван. За время его приостановки другой поток занял ресурс, вошел в свою критическую секцию, но также был прерван, не завершив работы с разделяемым ресурсом. Когда управление было возвращено первому потоку, 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 многих компьютеров предусмотрена единая, неделимая команда анализа и присвоения значения логической переменной (например, команды BTC, BTR и BTS процессора Pentium). При отсутствии такой команды в процессоре соответствующие действия должны реализовываться специальными системными примитивами', которые бы запрещали прерывания на протяжении всей операции проверки и установки.

Реализация взаимного исключения описанным выше способом имеет существенный недостаток: в течение времени, когда один поток находится в критической секции, другой поток, которому требуется тот же ресурс, получив доступ к процессору, будет непрерывно опрашивать блокирующую переменную, бесполезно тратя выделяемое ему процессорное время, которое могло бы быть использовано для выполнения какого-нибудь другого потока. Для устранения этого недостатка во многих ОС предусматриваются специальные системные вызовы для работы с критическими секциями.

На рис. 4.19 показано, как с помощью этих функций реализовано взаимное исключение в операционной системе Windows NT. Перед тем как начать изменение критических данных, поток выполняет системный вызов EnterCriticalSection(). В рамках этого вызова сначала выполняется, как и в предыдущем случае, проверка блокирующей переменной, отражающей состояние критического ресурса. Если системный вызов определил, что ресурс занят (F(D) = 0), он в отличие от предыдущего случая не выполняет циклический опрос, а переводит поток в состояние ожидания (D) и делает отметку о том, что данный поток должен быть активизирован, когда соответствующий ресурс освободится. Поток, который в это время использует данный ресурс, после выхода из критической секции должен выполнить системную функцию LeaveCriticalSectionO, в результате чего блокирующая переменная принимает значение, соответствующее свободному состоянию ресурса (F(D) = 1), а операционная система просматривает очередь ожидающих этот ресурс потоков и переводит первый поток из очереди в состояние готовности.

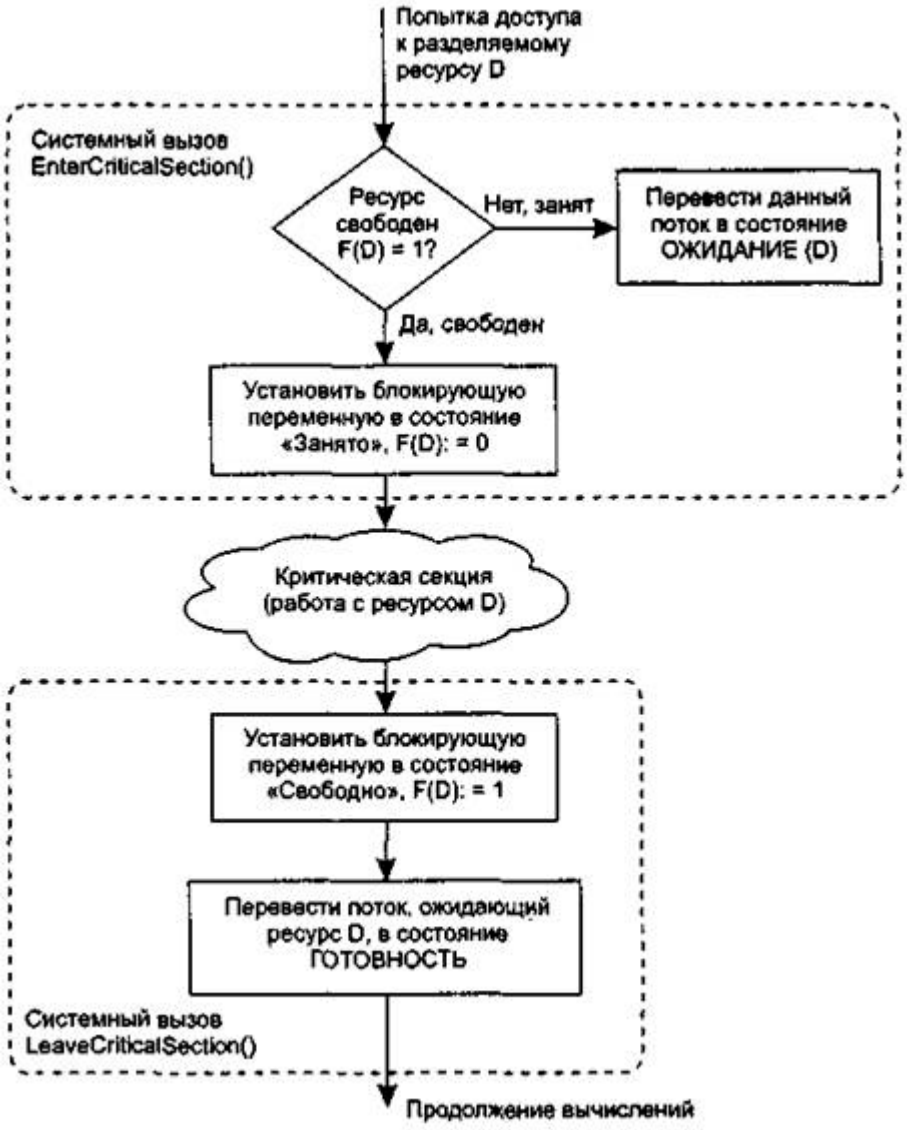

Рис. 4.19. Реализация взаимного исключения с использованием системных функций входа в критическую секцию и выхода из нее

Таким образом исключается непроизводительная потеря процессорного времени на циклическую проверку освобождения занятого ресурса. Однако в тех случаях, когда объем работы в критической секции небольшой и существует высокая вероятность в очень скором доступе к разделяемому ресурсу, более предпочтительным может оказаться использование блокирующих переменных. Действительно, в такой ситуации накладные расходы ОС по реализации функции входа в критическую секцию и выхода из нее могут превысить полученную экономию.