Тема 3. Синхронизация процессов и потоков

Потребность в синхронизации потоков возникает в мультипрограммной операционной системе и связана с совместным использованием аппаратных и информационных ресурс об 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 предсказать, на каком этапе выполнения будет находиться процесс в определенный момент времени.

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. 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.

Потоки 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 необходима синхронизация. 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.

Для синхронизации потоков прикладных программ программист может использовать как собственные средства и приемы синхронизации, так и средства операционной системы.

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.

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, 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.

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.

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.

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.* Во избежание таких ситуаций в системе команд многих компьютеров предусмотрена единая, неделимая команда анализа и присвоения значения логической переменной..

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*.

Здесь 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.

Обобщением блокирующих переменных являются так называемые ***семафоры Дийкстры***. Вместо двоичных переменных Дийкстра (Dijkstra) предложил использовать *переменные, которые могут принимать целые неотрицательные значения.* Такие переменные, используемые для синхронизации вычислительных процессов, получили название **семафоров.**

Для работы с семафорами вводятся два примитива, традиционно обозначаемых P и V. Пусть переменная S представляет собой семафор. Тогда действия V(S) и P(S) определяются следующим образом.

- **V(S):** переменная S увеличивается на 1 единым действием. Выборка, наращивание и запоминание не могут быть прерваны. К переменной S нет доступа другим потокам во время выполнения этой операции.
- **P(S):** уменьшение S на 1, если это возможно. Если невозможно уменьшить S, оставаясь в области целых неотрицательных

значений, то поток, вызывающий операцию P, ждет, пока это уменьшение станет возможным. Успешная проверка и уменьшение также являются неделимой операцией.

В частном случае, когда семафор S может принимать только значения 0 и 1, он превращается в блокирующую переменную, которую по этой причине часто называют ***двоичным семафором***.

Операция P заключает в себе потенциальную возможность перехода потока, который ее выполняет, в состояние ожидания, в то время как операция V может при некоторых обстоятельствах активизировать другой поток, приостановленный операцией P.

Семафор может использоваться и *в качестве блокирующей переменной*. В рассмотренном выше примере, для того чтобы исключить коллизии при работе с разделяемой областью памяти, будем считать, что запись в буфер и считывание из буфера являются критическими секциями.

В случае, когда потоки взаимно блокируют друг друга возникают ***взаимные блокировки***, называемые также *дедлоками* (deadlocks), *клинчами* (clinch), или ***тупиками***.

Невозможность потоков завершить начатую работу из-за возникновения взаимных блокировок снижает производительность вычислительной системы. Поэтому проблеме предотвращения тупиков уделяется большое внимание.

На тот случай, когда взаимная блокировка все же возникает, система должна предоставить администратору-оператору средства, с помощью которых он смог бы распознать тупик, отличить его от обычной блокировки из-за временной недоступности ресурсов. И наконец, если тупик диагностирован, то нужны средства для снятия взаимных блокировок и восстановления нормального вычислительного процесса.

Тупики могут быть *предотвращены на стадии написания программ*, то есть программы должны быть написаны таким образом, чтобы тупик не мог возникнуть при любом соотношении взаимных скоростей потоков.

Другой, более гибкий подход к *предотвращению* тупиков заключается в том, что *ОС каждый раз при запуске задач анализирует их потребности в ресурсах* и определяет, может ли в данной мультипрограммной смеси возникнуть тупик. Если да, то запуск новой задачи временно откладывается. ОС может также использовать определенные правила при назначении ресурсов потокам, например, ресурсы могут выделяться операционной системой в определенной последовательности, общей для всех потоков.

В случаях, когда тупиковую ситуацию не удалось предотвратить, важно быстро и точно ее распознать, поскольку блокированные потоки не выполняют никакой полезной работы. Если тупиковая ситуация образована множеством потоков, занимающих массу 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 потоки. Можно снять только часть из них, освободив ресурсы, ожидаемые остальными потоками, можно вернуть некоторые потоки в область подкачки, можно совершить -«откат» некоторых потоков до так называемой контрольной точки, в которой запоминается вся информация, необходимая для восстановления выполнения программы с данного места. Контрольные точки расставляются в программе в тех местах, после которых возможно возникновение тупика.

Механизмы синхронизации, основанные на использовании глобальных переменных процесса, обладают существенным недостатком — они не подходят для синхронизации потоков разных процессов. В таких случаях операционная система должна предоставлять потокам системные объекты синхронизации, которые были бы видны для всех потоков, даже если они принадлежат разным процессам и работают в разных адресных пространствах.

Примерами таких синхронизирующих объектов ОС являются ***системные семафоры, мьютексы, события, таймеры*** и другие — их набор зависит от конкретной ОС, которая создает эти объекты по запросам процессов.

Чтобы процессы могли разделять синхронизирующие объекты, в разных ОС используются разные методы.

Некоторые ОС возвращают указатель на объект. Этот указатель может быть доступен всем родственным процессам, наследующим характеристики общего родительского процесса.

В других ОС процессы в запросах на создание объектов синхронизации указывают имена, которые должны быть им присвоены. Далее эти имена используются разными процессами для манипуляций объектами синхронизации. В таком, случае работа с синхронизирующими объектами подобна работе с файлами. Их можно создавать, открывать, закрывать, уничтожать.

Кроме того, для синхронизации могут быть использованы такие «обычные» объекты ОС, как файлы, процессы и потоки. Все эти объекты могут находиться в двух состояниях: сигнальном и несигнальном — свободном.

Для каждого объекта смысл, вкладываемый в понятие «сигнальное состояние», зависит от типа объекта. Так, например, поток переходит в сигнальное состояние тогда, когда он завершается. Процесс переходит в сигнальное состояние тогда, когда завершаются все его потоки. Файл переходит в сигнальное состояние в том случае, когда завершается операция ввода-вывода для этого файла. Для остальных объектов сигнальное состояние устанавливается в результате выполнения специальных системных вызовов. Приостановка и активизация потоков осуществляются в зависимости от состояния синхронизирующих объектов ОС.

Поток может ожидать установки сигнального состояния не одного объекта, а нескольких. При этом поток может попросить ОС активизировать его при установке либо одного из указанных объектов, либо всех объектов. В зависимости от объекта синхронизации в состояние готовности могут переводиться либо все ожидающие это событие потоки, либо один из них.

Синхронизация тесно связана с *планированием потоков*,

- *во-первых*, любое обращение потока с системным вызовом Wait(X) влечет за собой действия в подсистеме планирования — этот поток снимается с выполнения и помещается в очередь ожидающих потоков, а из очереди готовых потоков выбирается и активизируется новый поток.
- *Во-вторых*, при переходе объекта в сигнальное состояние (в результате выполнения некоторого потока — либо системного, либо прикладного) ожидающий этот объект поток (или потоки) переводится в очередь готовых к выполнению потоков.

В обоих случаях осуществляется перепланирование потоков, при этом если в ОС предусмотрены изменяемые приоритеты и/или кванты времени, то они пересчитываются по правилам, принятым в этой операционной системе.

Рассмотрим несколько примеров, когда в качестве синхронизирующих объектов используются файлы, потоки и процессы.

Пусть выполнение некоторого приложения требует последовательных работ-этапов. Для каждого этапа имеется свой отдельный процесс. Сигналом для начала работы каждого следующего процесса является завершение предыдущего. Для реализации такой логики работы необходимо в каждом процессе, кроме первого, предусмотреть выполнение системного вызова Wait(X), в котором синхронизирующим объектом является предшествующий поток.

Объект-файл, переход которого в сигнальное состояние соответствует завершению операции ввода-вывода с этим файлом, используется в тех случаях, когда поток, инициировавший эту операцию, решает дождаться ее завершения, прежде чем продолжить свои вычисления.

Однако круг событий, с которыми потоку может потребоваться синхронизировать свое выполнение, отнюдь не исчерпывается завершением потока, процесса иди операции ввода-вывода. Поэтому в ОС, как правило, имеются и другие, более универсальные объекты синхронизации, такие как *событие (event), мьютекс (nmtex), системный семафор* и другие.

Мьютекс, как и семафор, обычно используется для управления доступом к данным.

В отличие от объектов-потоков, объектов-процессов и объектов-файлов, которые при переходе в сигнальное состояние переводят в состояние готовности все потоки, ожидающие этого события, объект - мьютекс «освобождает» из очереди ожидающих только один поток.

Работа мьютекса хорошо поясняется в терминах «владения». Пусть поток, который, пытаясь получить доступ к критическим данным, выполнил системный вызов Wait(X), где X — указатель на мьютекс. Предположим, что мьютекс находится в сигнальном состоянии, в этом случае поток тут же становится его владельцем, устанавливая его в несигнальное состояние, и входит в критическую секцию. После того как поток выполнил работу с критическими данными, он «отдает» мьютекс, устанавливая его в сигнальное состояние. В этот момент мьютекс свободен и не принадлежит ни одному потоку. Если какой-либо поток ожидает его освобождения, то он становится следующим владельцем этого мьютекса, одновременно мьютекс переходит в несигнальное состояние.

Объект-событие обычно используется не для доступа к данным, а для того, чтобы оповестить другие потоки о том, что некоторые действия завершены. Пусть, например, в некотором приложении работа организована таким образом, что один поток читает данные из файла в буфер памяти, а другие потоки обрабатывают эти данные, затем первый поток считывает новую порцию данных, а другие потоки снова ее обрабатывают и так далее. В начале работы первый поток устанавливает объект-событие в несигнальное состояние. Все остальные потоки выполнили вызов Wait(X), где X — указатель события, и находятся в приостановленном состоянии, ожидая наступления этого события. Как только буфер заполняется, первый поток сообщает об этом операционной системе, выполняя вызов Set(X). Операционная система просматривает очередь ожидающих потоков и активизирует все потоки, которые ждут этого события.

Сигнал дает возможность задаче реагировать на событие, источником которого может быть операционная система или другая задача. Сигналы вызывают прерывание задачи и выполнение заранее предусмотренных Действий. Сигналы могут вырабатываться синхронно, то есть как результат работы самого процесса, а могут быть направлены процессу другим процессом; то есть вырабатываться асинхронно. Синхронные сигналы чаще всего приходят от системы прерываний процессора и свидетельствуют о действиях процесса, блокируемых аппаратурой, например деление на нуль, ошибка адресации, нарушение защиты памяти и т. д.

Примером асинхронного сигнала является сигнал с терминала. Во многих ОС предусматривается оперативное снятие процесса с выполнений. Для этого пользователь может нажать некоторую комбинацию клавиш (Ctrl+C, Ctrl+Break), в результате чего ОС вырабатывает сигнал и направляет его активному процессу. Сигнал может поступить в любой момент выполнения процесса (то есть он является асинхронным), требуя от процесса немедленного завершения работы. В данном случае реакцией на сигнал является безусловное завершение процесса.

В системе может быть определен набор сигналов. Программный код процесса, которому поступил сигнал, может либо проигнорировать его, либо прореагировать на него стандартным действием (например, завершиться), либо выполнить специфические действия, определенные прикладным

программистом. В последнем случае в программном коде необходимо предусмотреть специальные системные вызовы, с помощью которых операционная система информируется, какую процедуру надо выполнить в ответ на поступление того или иного сигнала.

Сигналы обеспечивают логическую связь между процессами, а также между процессами и пользователями (терминалами). Поскольку посылка сигнала предусматривает знание идентификатора процесса, то взаимодействие посредством сигналов возможно только между родственными процессами, которые могут получить данные об идентификаторах друг друга.

В распределенных системах, состоящих из нескольких процессоров, каждый из которых имеет собственную оперативную память, блокирующие переменные, семафоры, сигналы и другие аналогичные средства, основанные на разделяемой памяти, оказываются непригодными. В таких системах синхронизация может быть реализована только *посредством обмена сообщениями.*