Тема 4. Многозадачность в ОС GNU/Linux: часть 3

Главные вопросы, которые мы обсудим, представлены на СЛАЙДЕ 1. По ходу лекции даются примеры системных вызовов для синхронизации потоков.

4.1 Проблема синхронизация потоков

Программирование потоков – непростая задача, ведь они обычно выполняются асинхронно. Иначе говоря, невозможно определить, когда система предоставляет доступ к процессору одному потоку, а когда – второму, третьему и т.д. Длительность этого доступ может быть как длительной, так и короткой, в зависимости от того, как ОС переключается с одной задачи на другую. Если в вычислительной системе есть несколько исполнительных устройств, то потоки могут выполняться в буквальном смысле одновременно.

Один из неприятных моментов в разработке многопоточных приложений – это их отладка, т.к. не всегда существует возможность воссоздания ситуации, приведшей к проблеме. В каких-то случаях программа работает корректно, а в других – приводит к сбоям в системе. Не получится заставить ОС распланировать выполнение потоков так, как она делала при предыдущем запуске программы.

Большинство ошибок в многопоточных программах связано с тем, что потоки пытаются обращаться к одним и тем же данным, как отмечалось на предыдущей лекции. Это и достоинство потоков, и их недостаток. Если один поток заполняет некую структуру данными в то время, когда второй поток обращается к этой же структуре, поведение программы становится непредсказуемым. Часто неправильно спроектированные потоковые приложения корректно работают только в том случае, если один поток планируется системой, которая устанавливает ему более высокий приоритет, чем у другого потока с более низким приоритетом. Высокоприоритетная задача имеет право чаще и быстрее обращаться к процессору. Подобные ошибки носят название **состояния гонки**, поскольку потоки пытаются обогнать друг друга в попытке обращения к одним данным.

4.2 Состояние гонки

Пусть в программу поступает группа запросов, которые обрабатываются несколькими одновременными потоками. Очередь запросов представлена связным списком структур типа *job*.

Когда каждый поток завершает свою операцию, он обращается к очереди и проверяет, есть ли в ней еще необработанные элементы. Если указатель *g_JobQueue* не равен NULL, то поток удаляет из списка его первый элемент и перемещает указатель на следующее «задание».

Потоковая функция, обрабатывающая с очередью заданий, представлена на СЛАЙДЕ 2.

Теперь мы предположим следующую ситуацию. Два потока завершают свои операции примерно в одно и то же время, а в очереди остается только одно задание. Первый поток проверяет, равен ли указатель *g_JobQueue* значению NULL. Обнаружив, что очередь не пуста, входит в цикл, где сохраняет указатель на объект задания в переменной *nextJob*. В этот момент ОС прерывает первый поток и активизирует второй,

который тоже проверяет указатель *g_JobQueue*, устанавливает, что он не равен NULL, и записывает тот же указатель в локальную переменную *nextJob*. Теперь мы получили два потока, выполняющих одно и то же задание.

К сожалению, дальше будет еще хуже. Первый поток удаляет последнее задание в очереди, делая переменную *g_JobQueue* равной NULL. Когда второй поток попытается выполнить операцию *g_JobQueue->next* возникает известная ошибка сегментации.

Мы продемонстрировали гонку за ресурсами. Если программа счастливо избежит такого планирования, то данная ошибка может не проявиться. Вероятнее всего в сильно загруженной системе произойдет необъяснимый сбой.

Чтобы исключить возможность гонки, необходимо сделать операции **атомарными**. Это означает неделимость и непрерываемость операции; если она началась, то уже не может быть приостановлена или прервана до своего завершения. Выполнение других операций в это время невозможно. СЛАЙД 3.

4.3 Исключающие семафоры, или мьютексы

Один из традиционных вариантов избегания эффекта гонки заключается в том, чтоб позволить только одному потоку в один момент времени обращаться к разделяемым данным.

Реализация такого решения требует поддержки от ОС. В Linux, как в ряде других ОС, имеется специальное средство, называемое **исключающим семафором**, или **мьютексом**. Это специальная блокировка, которую в определенный момент времени может устанавливать только один поток. Если мьютекс захвачен каким-то потоком, то другие потоки, обращающиеся к мьютексу, оказываются заблокированными либо переведенными в состояние ожидания. Как только мьютекс освобождается, поток может продолжить выполнение. СЛАЙД 4.

Для создания мьютекса, нужно объявить переменную типа *pthread_mutex_t* и передать указатель на нее функции *pthread_mutex_init()*. Один из параметров этой функции (а именно – второй) представляет собой указатель на объект атрибутов мьютекса. Если этот объект пуст (как в функции *pthread_create()*), используются атрибуты по умолчанию. Объект мьютекса инициализируется единожды, а делается это двумя способами (СЛАЙД 5). Первый:

```
pthread_mutex_t mtx;
pthread_mutex_init(&mtx, NULL);
```

Второй:

```
pthread_mutex_t mtx = PTHREAD_MUTEX_INITIALZER;
```

Как видно, он чуть проще, объекту мьютекса присваивают специальное значение, а функцию инициализации не вызывают. Это удобно, когда объекты мьютекса являются глобальными переменными.

После создания и инициализации мьютекса потоки могут попытаться захватить его, вызвав функцию *pthread_mutex_lock()*. Если мьютекс свободен, то его новым владельцем становится захвативший его поток, а функция немедленно завершается. Если мьютекс уже был захвачен другим потоком, то выполнение функции *pthread_mutex_lock()* блокируется и возобновляется только после того, как мьютекс становится свободным.

Одновременно несколько потоков могут ожидать освобождения мьютекса. Когда такое событие наступает, только один из них разблокируется и получает возможность захватить мьютекс. Другие потоки остаются заблокированными.

В свою очередь, функция *pthread_mutex_unlock()* освобождает мьютекс. Она должна вызываться только из того потока, который захватил мьютекс.

В листинге кода, приведенного на СЛАЙДЕ 6, представлена новая версия потоковой функции, обрабатывающей очередь заданий. Только теперь мы эту очередь «защитили»

мьютексом. Перед доступом к очереди для чтения или записи каждый поток сначала захватывает мьютекс. Только по окончании всех операций от проверки очереди до удаления задания из нее произойдет освобождение мьютекса. Как следствие – не возникает состояния гонки.

Из кода видно, что все операции по доступу к разделяемому ресурсу – указателю *g_JobQueue* – происходят между вызовами *pthread_mutex_lock()* и *pthread_mutex_unlock()*. Элемент очереди заданий – переменная *nextJob* – обрабатывается после того, как ссылка на него удалена из очереди. За счет этого мы «защищаем» наш объект от других потоков.

Если очередь пуста (*g_JobQueue* равен NULL), то цикл не завершается немедленно. Иначе бы мьютекс оставался в захваченном состоянии и не позволил бы ни одному из остальных потоков получить доступ к очереди. Вместо этого мы записываем в переменную *nextJob* значение NULL и завершаем цикл **только после освобождения** мьютекса.

На самом деле мьютекс блокирует доступ к участку программного кода, а вовсе не к переменной. В обязанности программиста входит написание кода для захвата мьютекса до доступа к переменной и последующего его освобождения. Например, функция добавления элемента к очереди может выглядеть так, как показано на СЛАЙДЕ 7.

Взаимоблокировки мьютексов

Выше мы показали, что мьютексы являются механизмом, позволяющим одному потоку блокировать выполнения другого потока. Это может приводить к новому для нас классу ошибок – **тупикам**, или **взаимоблокировкам**. Их смысл состоит в том, что один или более потоков ожидают наступления события, которое никогда не произойдет.

Самая простая тупиковая ситуация – один поток пытается захватить один и тот же мьютекс дважды подряд. Дальнейшие события зависят от типа мьютекса (СЛАЙД 8):

1. Захват **быстрого мьютекса** (по умолчанию) приводит к тупику. Функция, обращающаяся к захваченному быстрому мьютексу, заблокирует поток до освобождения мьютекса. Однако он принадлежит самому потоку, поэтому блокировка может быть только путем аварийного завершения.

2. Захват **рекурсивного мьютекса** не приводит к тупику. Такой мьютекс запоминает количество обращений к нему функции *pthread_mutex_lock()* из потока, владеющего мьютексом. Для его освобождения необходимо вызвать такое же количество раз функцию *pthread_mutex_unlock()*.

3. Попытку повторного захвата **контролируемого мьютекса** ОС обнаруживает и сигнализирует об этом. При очередном вызове функции *pthread_mutex_lock()* будет возвращен код ошибки *EDEADLK*.

Для создания мьютексов 2 и 3 типов необходимо создавать объект атрибутов, объявив переменную *pthread_mutexattr_t* и передав указатель на нее функции инициализации *pthread_mutexattr_init()*. Затем следует задать тип мьютекса вызовом функции *pthread_mutexattr_setkind_np()*. Ее первый аргумент – указатель на объект атрибутов мьютекса. Если нам нужен рекурсивный мьютекс, задаем *PTHREAD_MUTEX_RECURSIVE_NP*; если контролируемый – *PTHREAD_MUTEX_ERRORCHECK_NP*. Указатель на полученный объект атрибутов необходимо передать известной нам функции *pthread_mutex_init()*. После этого необходимо удалить объект атрибутов с помощью функции *pthread_mutexattr_destroy()*. СЛАЙД 9.

Фрагмент программного кода, приведенный на СЛАЙДЕ 10, иллюстрирует процесс создания контролируемого мьютекса. Префиксы *np* и *NP* означают «не портабельный». Они специфичны для Linux и непереносимы в другие ОС, поэтому их не рекомендуется использовать в программах широкого назначения.

Неблокирующие проверки мьютексов

Иногда возникают ситуации, когда нужно проверить состояние мьютекса без блокировки потока. Для потока далеко не всегда приемлемо нахождение в режиме пассивного ожидания, т.к. это время он мог бы провести, как говорится, с большей пользой. Использовавшаяся ранее функция *pthread_mutex_lock()* не возвращает значение до тех пор, пока мьютекс не будет освобожден. Значит, она нам не подходит.

Зато подходит функция *pthread_mutex_trylock()*. Если она обнаруживает, что мьютекс свободен, то работает аналогично функции *pthread_mutex_lock()*, при этом она возвращает 0. Если оказывается, что мьютекс уже захвачен каким-то потоком, то функция *pthread_mutex_trylock()* не блокирует программу, а немедленно завершается с кодом *EBUSY*. СЛАЙД 11.

4.4 Потоковые семафоры

В нашем предыдущем примере группа потоков обрабатывает задания из очереди. Потоковая функция запрашивает задания до тех пор, пока очередь не опустеет, после чего поток завершается. Эта схема хорошо работает в том случае, когда задания были размещены в очереди заранее или новые задания поступают так же часто, как их запрашивают потоки. Однако если потоки начнут работать слишком быстро, очередь опустеет, и потоки завершатся. Может оказаться, что задание поступило, а потоков, которые его обрабатывают, уже нет. Значит, нужен механизм, которые блокирует потоки в случае, когда очередь пуста, а новые задания еще не поступили.

Этот механизм называется **семафором**. По сути это счетчик, используемый для синхронизации потоков. ОС гарантирует, что проверка и модификация значения семафора могут выполняться безопасно и не приведут к возникновению .

Счетчик – это неотрицательное целое число, а семафор поддерживает две базовые операции:

- Операция **ожидания** (*sem_wait*) уменьшает значение счетчика на единицу. Если значение счетчика уже равно нулю, то операция блокируется до тех пор, пока значение счетчика не станет положительным, например, в результате деятельности других потоков. После снятия блокировки значение семафора уменьшается на 1, и операция завершается.

- Операция **установки** (*sem_post*) увеличивает значение счетчика на единицу. Если до этого счетчик был равен 0, и существовали потоки, заблокированные в операции ожидания этого же семафора, то один из них разблокируется и завершается свою операцию. Значение счетчика снова становится равным 0.

Для работы с семафорами следует включить заголовочный файл *semaphore.h*. СЛАЙД 12.

В Linux есть две отличающиеся друг от друга реализации семафоров. Мы описываем в этом разделе реализацию, соответствующую стандарту POSIX. Они используются для организации взаимодействия потоков. Другая реализация предназначена для межпроцессного взаимодействия. В соответствующем разделе лекционного курса мы ее рассмотрим.

Семафор представляется переменной типа *sem_t*. Его всегда нужно инициализировать вызовом функции *sem_init()*. Ей передается указатель на переменную семафора, второй параметр всегда равен 0, третий – это начальное значение счетчика семафора.

Для выполнения блокирующего ожидания семафора вызывается функция *sem_wait()*, для неблокирующего ожидания – функция *sem_trywait()*, для установки семафора – *sem_post()*. Если ожидание может привести к блокированию потока из-за того, что счетчик равен 0, то функция *sem_trywait()* немедленно завершается с кодом *EAGAIN*.

Также есть функция *sem_getvalue ()*, позволяющая узнать текущее значение счетчика семафора. Это значение помещаются в переменную типа *int*, на которую ссылается второй аргумент функции. Не рекомендуется с использованием этой целочисленной переменной определять, какая операция выполняется, т.к. это может приводить к гонке. Между

вызовами *sem_getvalue()* и какой-нибудь иной функции работы с семафором другой поток может изменить значение счетчика. Гарантию дают только функции *sem_wait()* и *sem_post()*. СЛАЙД 13.

Мы снова модифицируем предыдущую программу. Можно реализовать проверку того, сколько заданий содержится в очереди. Этот код приведен на СЛАЙДАХ 14-15.

До извлечения задания из очереди каждый поток дожидается семафора. Если счетчик семафора равен 0, т.е. очередь пуста, то поток блокируется до тех пор, пока в очереди не появится новое задание, и счетчик не станет положительным числом. Реализованная нами функция *EnqueueJob()* добавляет новое задание в очередь. Подобно потоковой функции она захватывает мьютекс перед обращением к очереди. После добавления задания функция *EnqueueJob()* устанавливает семафор. Тем самым потоки уведомляются, что задание доступно. В нашем коде потоки никогда не завершаются. Если задания не поступают в течение длительного времени, все потоки блокируются функцией *sem_wait()*.

4.5 Условные переменные

В предыдущих параграфах мы увидели, как с помощью мьютекса можно защитить программный объект от одновременного доступа со стороны двух и более потоков. Мы так же показали, как посредством обычного семафора можно реализовать счетчик обращений, доступный нескольким потокам. **Сигнальная** переменная, она же – **условная** переменная – это еще один механизм синхронизации потоков доступный в Linux.

Предположим, нам надо написать функция потока, который входит в бесконечный цикл, выполняя на каждой итерации некоторое действие. Работа цикла должна контролироваться флагом. Действие выполняется при условии, что он «поднят».

В листинге кода, приведенного на СЛАЙДАХ 16-17, на каждой итерации потоковая функция проверяет, установлен ли флаг. Т.к. к нему могут обращаться несколько потоков, то он защищается мьютексом. Эта реализация корректна, но неэффективна. Если флаг «опущен», то поток будет понапрасну тратить ресурсы процессора, занимаясь бесконечной проверкой флага, а также захватывая и освобождая мьютекс. Нам хотелось бы как-то перевести поток в неактивный режим, пока другой поток не «поднимет» этот флаг.

Условная переменная такую проверку организовать позволяет: поток либо выполняется, либо блокируется. Как и в случае семафора, поток может **ожидать** условную переменную.

Рассмотрим систему из двух потоков: *A* и *B*. Ожидающий поток *A* блокируется до тех пор, пока поток *B* не просигнализирует об изменении состояния переменной. У условной переменной счетчика нет, в отличие от семафора. Поток *A* должен перейти в состояние ожидание до прихода сигнала от *B*. Если сигнал послан раньше, он может потеряться, и поток *A* заблокируется, пока какой-то другой поток не пошлет сигнал повторно.

Мы можем внести в нашу программу следующие изменения, чтобы сделать ее более эффективной.

1. Потоковая функция в цикле проверяет флаг. Если он не «поднят», то поток переходит в режим ожидания условной переменной.

2. Функция *SetThreadFlag()* устанавливает флаг и сигнализирует об изменении условной переменной. Если потоковая функция была блокирована в ожидании сигнала, то она разблокируется и опять проверяет флаг.

Остался один не очень приятный момент – гонка между проверкой флага и операция сигнализирования или ожидания сигнала. Например, потоковая функция проверяет флаг и обнаруживает, что он не «поднят». В этот момент планировщик задач ОС прерывает выполнение данного потока и активизирует основной поток. Он как раз выполняет функцию *SetThreadFlag()*, которая «поднимает» флаг и сигнализирует об изменении

условной переменной. Но в этот момент нет потоков, ожидающих данного сигнала, т.е. он потерян. Когда ОС снова активизирует потоковую функцию, он начнет ждать сигнал, который вероятнее всего больше не придет.

Чтобы избежать этой проблемы, нам нужно одновременно захватить флаг вместе с условной переменной с помощью мьютекса. Любая условная переменная должна использоваться вместе с мьютексом для предотвращения эффекта гонок. Иными словами, наша потоковая функция должна выполнять следующие несложные шаги.

1. В цикле захватывается мьютекс и читается значение флага.

2. Если флаг «поднят», то мьютекс разблокируется, и выполняются требуемые действия.

3. Если флаг «опущен», одновременно выполняются операции освобождения мьютекса и перехода в режим ожидания сигнала.

Вся соль в третьем шаге. Здесь ОС позволяет выполнять атомарную операцию освобождения мьютекса и перехода к ожиданию сигнала. Вмешательство других потоков в этот момент не допускается.

Условная переменная имеет тип *pthread_cond_t*. Следует помнить, что ей обязательно нужно сопоставить мьютекс. Для работы с условными переменными предназначены следующие функции (СЛАЙД 18):

- *pthread_cond_init()* инициализирует условную переменную. Первый аргумент – указатель на объект условной переменной, второй – игнорируется.

- *pthread_cond_signal()* сигнализирует об изменении переменной. При этом разблокируется один из потоков, ожидающий сигнала. Если таких нет – сигнал игнорируется. Аргумент – указатель на объект условной переменной.

- *pthread_cond_broadcast()* похожа *pthread_cond_signal()*, только разблокирует **все** потоки, ожидающие сигнал.

- *pthread_cond_wait()* блокирует вызывающий ее поток до тех пор, пока не будет получен сигнал об изменении условной переменной. Первый аргумент – указатель на объект условной переменной, второй – указатель на мьютекс. В момент вызова данной функции мьютекс уже должен быть захвачен вызывающим потоком. Функция атомарно освобождает мьютекс и блокирует поток в ожидании сигнала. Когда он поступает, функция разблокирует поток и автоматически захватывает мьютекс.

Перечисленные далее этапы должны выполняться всякий раз, когда программа так или иначе меняет результат проверки условия, контролируемого условной переменной:

1. Захватить мьютекс, дополняющий условную переменную.

2. Выполнить действие, включающее изменение результата проверки условия .

3. Послать сигнал одному или нескольким потокам об изменении условия.

4. Освободить мьютекс. СЛАЙД 19.

В листинге кода, приведенного на СЛАЙДАХ 20-21, показана модифицированная версия предыдущего примера. В ней флаг защищается условной переменной. В потоковой функции мьютекс захватывается до проверки переменной *g_ThreadFlag*. Захват автоматически снимается функцией *pthread_cond_wait()* перед тем, как поток заблокируется, и также автоматически восстанавливается по окончании функции.

Условие, контролируемое переменной, может быть сколь угодно сложным. Однако перед выполнением любой операции, способной повлиять на результат проверки условия, необходимо захватить мьютекс, и только после этого можно посылать сигнал.

Интересно, что условная переменная вообще может не быть связанной с каким-либо условием, а служить обычным средством блокирования потока до тех пор, пока другой поток не разбудит его. Для этой цели у нас использовался семафор. Основное различие между ними – семафор запоминает сигнал, даже если нет ни одного блокированного потока, а условная переменная регистрирует сигнал только тогда, когда есть ожидающие потоки. Кроме того, семафор всегда разблокирует лишь один поток, а вот функцией *pthread_cond_broadcast()* можно разблокировать несколько потоков.

Взаимоблокировки двух и более потоков

Взаимоблокировка происходит тогда, когда два или более потоков блокируются в ожидании события, наступление которого на самом деле зависит от действия одного из заблокированных потоков. Например, поток *A* ожидает изменения условной переменной, устанавливаемой в потоке *B*. Тот ожидает сигнала от потока *A*, т.е. возникает тупик, поскольку ни один из потоков не пошлет сигнал другому потоку. Необходимо аккуратно избегать таких ситуаций, потому что они весьма трудны для обнаружения.

Как правило, тупик возникает тогда, когда группа потоков пытается захватить один набор объектов. Допустим, у нас есть программа, в которой два потока с разными функциями, должны захватить одну и ту же пару мьютексов. Например, поток *A* захватывает сначала мьютекс *M1*, потом мьютекс *M2*. В то время как поток *B* захватывает их в обратном порядке. Весьма вероятно, что ОС после захвата *M1* первым потоком активизирует второй поток, который захватит *M2*. Далее оба потока оказываются блокированными, т.к. им закрыт доступ к мьютексам друг друга.

Литература и дополнительные источники к Теме 4

1. Робачевский, А. Операционная система Unix, 2 изд./ А.Робачевский, С.Немнюгин, О.Стесик. – СПб.: БХВ-Петербург, 2010. – 656 с.
2. POSIX thread (pthread) libraries - http://www.yolinux.com/TUTORIALS/LinuxTutorialPosixThreads.html
3. Инструменты Linux для Windows-программистов - http://rus-linux.net/nlib.php?name=/MyLDP/BOOKS/Linux-tools/index.html
4. Лав, Р. Linux. Системное программирование/ Р.Лав. – СПб.: Питер, 2008. – 416 с.