# Тема 3. Многозадачность в ОС GNU/Linux: часть 2



Главные вопросы, которые мы обсудим, представлены на СЛАЙДЕ 1. По ходу лекции даются примеры системных вызовов для работы с потоками.

## 3.1 Многозадачность: потоки

Потоки, как и процессы, – это механизм реализации многозадачности. Потоки работают параллельно. ОС планирует их работу асинхронно, прерывая время от времени каждый из них, чтобы предоставить шанс остальным.

Поток существует внутри процесса, являясь более мелкой единицей работы. При вызове программы ОС создает для нее новый процесс, а в нем единственный поток, последовательно выполняющий программный код. Этот поток может создавать другие потоки. Все они находятся в одном процессе, выполняя ту же самую программу, но возможно в разных ее местах.

В предыдущей лекции мы показали, как программы порождают дочерние процессы. Они получают при создании копию большей части ресурсов родительского процесса . Когда программа создает поток, ничего не копируется. Оба потока – старый и новый – имеют доступ к адресному пространству процесса, общим файлам и другим ресурсам ОС. Если, скажем, один поток меняет значение некоторой переменной, то это изменение отразится на другом потоке. Аналогично, когда один поток закрывает файл, второй поток теряет возможность работать с этим файлом. В связи с тем, что процесс и все его потоки могут выполнять одну программу одновременно, как только один из потоков рискнул вызвать функцию *exec()*, все остальные потоки завершаются.

В Linux реализован *прикладной программный интерфейс* (API) для работы с потоками в виде библиотеки функций, соответствующей стандарту POSIX, которая называется *pthreads*. Все функции и типы данных библиотеки объявлены в файле *pthread.h*. Они не входят в стандартную библиотеку языка С, поэтому при сборке программы в командной строке нужно указывать опцию -*lpthread*. (СЛАЙД 2)

## 3.2 Создание потока

Каждому потоку в процессе назначается собственный идентификатор. При ссылке на идентификаторы потоков в программах нужно использовать тип данных *pthread_t*. После создания поток начинает выполнять **потоковую функцию**. Это ничем не отличающаяся от остальных функция, которая содержит программный код потока. По завершении функции поток уничтожается. В Linux потоковые функции принимают единственный параметр типа *void* * и возвращают значение того же типа. Параметр называется **аргументом потока**. Аналогично, через **возвращаемое значение** программы могут принимать данные от потоков.

Функция *pthread_create()* создает поток для выполнения внутри адресного пространства вызывающего процесса. Ей передаются параметры (СЛАЙД 3):

```
#include <pthread.h>
int pthread_create
(
  pthread_t* thread,               // указатель на идентификатор потока
  const pthread_attr_t* attr,      // указатель на запись с атрибутами
  void* (*start_routine) (void *), // указатель на функцию потока
```

```
  void* arg                          // параметр для нового потока
);
```

Функция *pthread_create()* немедленно завершается, и родительский поток переходит к выполнению инструкции, следующей после вызова функции. Одновременно новый поток начинает выполнять функцию. ОС планирует работу обоих потоков асинхронно, поэтому программа не должна рассчитывать на какую-то согласованность между ними.

Программа, представленная на СЛАЙДЕ 4, создает поток, который непрерывно записывает символы ‘y’ в *STDERR*. После создания дополнительного потока главный поток так же начинает печатать символы, только вместо ‘y’ выводятся ‘e’. Когда мы запустим эту программу, мы увидим, что символы печатаются непредсказуемым образом.

При нормальных обстоятельствах поток завершается одним из двух способов. Первый – выход из потоковой функции. Возвращаемое значение считается значением, передаваемым из потока в программу. Второй способ – вызов функции *pthread_exit()*. Это может быть сделано из потоковой функции, а также и из явно или неявно вызываемых ею функций. Аргумент функции *pthread_exit()* становится значением, возвращаемым потоком.

### *Передача данных потоку*

Потоковый аргумент – это весьма удобное средство передачи данных потокам. Но поскольку его тип *void* *, то данные, конечно же, содержатся не в самом аргументе. Он всего лишь указывает на какую-то область памяти . Лучше всего создать для каждой потоковой функции собственную структуру, в которой определялись бы «параметры», ожидаемые потоковой функцией.

Из-за наличия потокового аргумента появляется возможность использовать одну и ту же потоковую функцию с разными потоками. Все они будут выполнять один и тот же код, но с разными данными.

Мы модифицируем предыдущий пример кода. Создаются два потока: один отображает ‘y’, другие – символы ‘e’. Чтобы вывод на консоль не был бесконечным, потокам передается аргумент, определяющий, сколько раз следует отобразить символ. Одна и та же функция используется двумя потоками, но каждый из них параметризуется соответствующей структурой. См. код на СЛАЙДАХ 5-6.

Наша очевидная недоработка заключается в том, что основной поток создает обе структуры в виде локальных переменных, а затем передает указатели на них дочерним потокам. И ничего не мешает системе распланировать работу потоков так, чтобы функция *main()* завершилась до того, как будут завершены два других потока. Но если это произойдет, то структуры окажутся удаленными из памяти, хотя оба потока все еще ссылаются на них!

### *Ожидание завершения потоков*

Одно из простых решений описанной проблемы заключается в том, чтобы заставить функцию *main()* дождаться завершения потоков. Нам нужна функция наподобие *wait()*, которую мы использовали для процессов. Такая функция есть, и она называется *pthread_join()*. У нее два аргумента: идентификатор ожидаемого потока и указатель на переменную *void* *, в которую будет записано значение, возвращаемое потоком. Если последнее не важно, мы можем задать в качестве второго аргумента *NULL*.

В коде, представленном на СЛАЙДАХ 7-8, приведена исправленная версия программы со СЛАЙДОВ 5-6. В данном случае функция *main()* не завершается, пока оба дочерних потока не выполнят свои задания и не перестанут ссылаться на переданные им структуры.

Главный вывод из этого: нужно убедиться, что любые данные, переданные потоку по ссылке, не удаляются до тех пор, пока поток не завершит свою работу с ними. Это

относится и к локальным переменным, и к динамически распределяемым переменным.

### *Значения, возвращаемые потоками*

Если второй аргумент функции *pthread_join()* не равен *NULL*, в него помещается значение, возвращаемое потоком. Как и потоковый аргумент, это значение имеет тип *void* *. Если поток возвращает обычный *int*, то его можно свободно привести к *void* *, а затем выполнить обратное преобразование по завершении функции *pthread_join()*.

Программа, чей код представлен на СЛАЙДАХ 9 и 10 в отдельном потоке вычисляет простое число с номером *n*. Тем временем функция *main()* может продолжать свои собственные вычисления. Дотошный слушатель сразу увидит крайне неэффективную реализацию, ведь существуют более мощные алгоритмы.

### *Идентификаторы потоков*

Иногда программе необходимо определить, какой поток выполняется в данный момент. Функция *pthread_self()* возвращает идентификатор потока, в котором она вызвана. Для сравнения двух разных идентификаторов предназначена функция *pthread_equal()*.

Эти функции удобны для проверки соответствия заданного идентификатора текущему потоку. Например, поток не должен вызывать функцию *pthread_join()*, чтобы ждать самого себя. В этом случае возвращается код ошибки *EDEADLK*, а избежать ее позволяет простая проверка (СЛАЙД 11):

```
if (0 == pthread_equal(pthread_self(), otherThread))
  pthread_join(otherThread, NULL);
```

### *Атрибуты потоков*

**Потоковые атрибуты** – это механизм настройки поведения отдельных потоков. Это, как указывалось выше, один из аргументов функции создания потока. Если он равен *NULL*, то поток конфигурируется стандартными атрибутами.

Для задания собственных атрибутов потока выполняются следующие действия (СЛАЙД 12).

1. Создается объект типа *pthread_attr_t*.
2. Вызывается функция *pthread_attr_init()*, которой передается указатель на объект.
3. В объект записываются требуемые значения атрибутов.
4. Указатель на этот объект передается функции *pthread_create()*.
5. Чтобы удалить объект из памяти, вызывается функция *pthread_attr_destroy()*. Сама переменная типа *pthread_attr_t* не удаляется; ее можно инициализировать вновь функцией *pthread_attr_init()*.

Один и тот же объект можно использовать для запуска нескольких потоков. Нет необходимости хранить объект после того, как поток был создан.

Для большинства наших приложений интерес может представлять единственный атрибут: **статус отсоединения потока**. Другие атрибуты используются для *realtime*-приложений. Поток можно создать как **ожидаемый** (такими были все предыдущие потоки) и **отсоединенный**. Ожидаемый поток после завершения хранится в системных таблицах и не удаляется оттуда, пока какой-то другой поток не вызовет функцию *pthread_join()*, чтобы запросить это значение. Только тогда ресурсы можно считать освобожденными.

Отсоединенный поток уничтожается сразу после завершения. Другие потоки не могут вызвать по отношению к нему функцию *pthread_join()* или получить возвращаемое ими значение.

Чтобы задать статус отсоединения потока, пользуются функцией *pthread_attr_setdetachstate()*. Первый аргумент – указатель на объект атрибутов потока, второй – требуемый статус. Ожидаемые потоки создаются по умолчанию, поэтому в качестве второго аргумента имеет смысл указывать только значение

*PTHREAD_CREATE_DETACHED*.

Заготовка кода программы, представленная на СЛАЙДЕ 13, создает отсоединенный поток, устанавливая должным образом его атрибуты.

Если поток был создан как ожидаемый, его позднее можно «отсоединить» вызовом функции *pthread_detach()*. Обратное – невозможно.

## 3.3 Отмена потока

Обычно поток завершается при выходе из потоковой функции или в результате вызова функции *pthread_exit()*. Однако, еще можно запросить из одного потока завершение другого. Это называется **отменой** или **принудительным завершением**.

Чтобы отменить поток, вызывается функция *pthread_cancel()*, которой передается идентификатор требуемого потока. Далее обязательно требуется дождаться завершения потока с целью освобождения ресурсов, если только поток не является отсоединенным. Отмененный поток возвращает специальное значение *PTHREAD_CANCELED*.

Во многих задачах поток выполняет код, который нельзя просто отменить. Скажем, потоку выделены ресурсы, с которыми работает, а затем освобождает. Если отмена потока произойдет где-то посередине, то освободить занятые ресурсы станет невозможно, вследствие чего они окажутся потерянными для системы. Чтобы учесть эту ситуацию, поток должен решить, где и когда он может быть отменен.

С точки зрения возможности отмены потоки могут быть в одном из трех состояний (СЛАЙД 14).

- **Асинхронно отменяемый** в любой точке его выполнения.
- **Синхронно отменяемый** только в определенных точках выполнения по запросу, который может помещаться в очередь.
- **Неотменяемый** (все попытки отмены игнорируются).

По умолчанию поток считается синхронно отменяемым.

### *Синхронные и асинхронные потоки*

Асинхронно отменяемый поток свободен в любое время. Синхронно отменяемый свободен, когда ему удобно. Соответствующие места в программе называются **точками отмены**. Запрос на отмену помещается в очередь и находится в ней до тех пор, пока не дойдет до следующей точки отмены.

Чтобы сделать поток асинхронно отменяемым, пользуются функцией *pthread_setcanceltype()*. Она влияет только на тот поток, в котором вызвана. Первый аргумент – *PTHREAD_CANCEL_ASYNCHRONOUS* (для асинхронных) либо *PTHREAD_CANCEL_DEFERRED* (для синхронных). Второй аргумент – указатель на переменную, куда записывается предыдущее состояние потока.

Делаем поток асинхронным (СЛАЙД 15):

```
pthread_setcanceltype(PTHREAD_CANCEL_ASYNCHRONOUS, NULL);
```

Что такое точка отмены, и где она должна находиться? Прямого ответа нет. Точка отмены создается функцией *pthread_testcancel()*. Все, что она делает, – это отработка отложенного запроса на отмену в синхронном потоке. Ее нужно периодически вызывать в потоковой функции в ходе выполнения длительной работы, причем там, где поток можно завершить без риска потери ресурсов.

Некоторые функции неявно создают точки отмены. О них можно прочитать в документации по функции *pthread_cancel()*. Они могут вызываться в других функциях, которые тем самым становятся точками отмены.

### *Неотменяемые потоки*

Поток, вообще говоря, может отказаться удаляться, вызвав функцию

*pthread_setcancelstate()*. Это оказывает влияние только на вызывающий поток. Первый аргумент *PTHREAD_CANCEL_ENABLE*, если нужно запретить отмену потока, и *PTHREAD_CANCEL_DISABLE* – в противном случае. Второй аргумент – указатель на переменную, в которой хранится предыдущее состояние потока.

Запрещаем отмену потока (СЛАЙД 16):

```
pthread_setcancelstate(PTHREAD_CANCEL_DISABLE, NULL);
```

Эта функция *pthread_setcancelstate()* позволяет организовывать **критические секции**. Под этим термином понимается участок кода программы, который должен быть выполнен целиком либо не выполнен вообще. Иначе говоря, уж если поток вошел в критическую секцию, то он во что бы то ни стало должен дойти до ее конца.

Допустим, для некоторого банковского приложения требуется написать функцию перевода денег с одного счета на другой. Для этого нужно добавить заданную сумму на баланс одного счета и вычесть такую же сумму с баланса другого счета. Очевидны последствия, если отмена потока произойдет между двумя этими операциями. Т.е. чтобы этого не случилось, мы должны обеспечить выполнение обеих операций в критической секции.

В коде, показанном на СЛАЙДЕ 17, приведен пример функции *ProcessTransaction()*, осуществляющей нашу гениальную мысль. Отмена потока запрещается, пока баланс обоих счетов не будет изменен.

По окончании критической секции восстанавливается предыдущее состояние потока, а не *PTHREAD_CANCEL_ENABLE*. Это позволит безопасно вызывать функцию *ProcessTransaction()* в другой критической секции.

В общем случае не рекомендуется отменять поток, если его можно просто завершить. Лучше как-то просигнализировать потоку о том, что он должен прекратить работу, а затем дождаться его завершения. Этому вопросу будет посвящена одна из следующих лекций.

## 3.4 Потоковые данные

В отличие от процессов потоки делят общее адресное пространство: если один поток модифицирует область памяти, то это изменение отражается на остальных потоках. Иными словами, разные потоки могут работать с одними и теми же данными без использования средств межзадачной коммуникации.

Однако у каждого потока есть собственный стек вызова, что позволяет всем потокам выполнять разный код, а также вызывать функции традиционным способом. При каждом вызове функции в стеке потока создается набор локальных переменных, сохраняющихся там до завершения.

Иногда, тем не менее, нужно дублировать определенную переменную, чтобы у каждого потока было ее собственная копия. С этой целью Linux предоставляет потокам **область потоковых данных**. Переменные, сохраняемые там, дублируются для каждого потока, что позволяет потокам свободно работать с ними, без вмешательства в функционирование друг друга. Доступ к потоковым данным невозможно получить по ссылкам на общие переменные, т.к. у потоков общее адресное пространство. Для работы с потоковыми данными в Linux имеется набор функций.

Все потоковые данные объявляются как переменные типа *void **. Объявлять их можно сколько угодно, ссылаться на них можно по ключу. Для создания ключей, т.е. новых переменных, предназначена функция *pthread_key_create()*. Первым аргументом этой функции является указатель на переменную типа *pthread_key_t*. В нее записывается значения ключа, с помощью которого любой поток сможет обращаться к своей копии данных. Второй аргумент – указатель на **функцию очистки ключа**. Она должна автоматически вызываться при завершении и уничтожении потока. Ей передается значение ключа, соответствующее данному потоку. Эта функция вызывается также и

после отмены потока в произвольной точке. Если потоковая переменная равна *NULL*, то функция очистки не вызывается. Если такая функция вообще не нужна, то в качестве второго аргумента функции *pthread_key_create()* передается значение *NULL*. СЛАЙД 18.

После создания ключа любой поток может назначать ему свое значение посредством вызова функции *pthread_setspecific()*. Первый аргумент – ключ, второй – требуемое значение типа *void* *. В свою очередь для чтения данных используется *pthread_getspecific()* с одним аргументом – ключом. СЛАЙД 19.

На практике часто встречаются и создаются программы, которые распределяют задачи по потокам. Можно за каждым отдельным потоком закреплять журнальный файл, куда записываются сообщения о ходе выполнения конкретной задачи. Для хранения указателя на журнальный файл удобно использовать область потоковых данных.

В листинге кода на СЛАЙДАХ 22-23 приведен один из вариантов, как это можно осуществить. Для хранения указателя функция *main()* создает ключ, запоминаемый в переменной g_*ThreadLogKey*, которая доступна всем потокам. В начале выполнения потоковой функции открывается журнальный файл, и сохраняется указатель на него в соответствующем ключе. Позднее поток может вызвать функцию *WriteToThreadLog()* для записи конкретных сообщений в журнальный файл. Эта функция извлекает из области потоковых данных указатель на журнал и помещает в файл нужное сообщение.

Потоковой функции не нужно закрывать журнал, поскольку функция *CloseThreadLog()* при создании ключа была указана для него как очищающая. Эта функция будет вызвана ОС, когда поток завершится, с передачей ей значения ключа. В ней-то и произойдет очистка с закрытием журнального файла.

### *Обработчики очистки*

Функции очистки ключей гарантируют, что потери ресурсов не произойдет ни при каких обстоятельствах. Однако иногда нужно создавать функцию, которая связывается не с дублируемыми ключами, а с обычными ресурсами. Такие функции называют **обработчиками очистки**.

Обработчик вызывается по окончании потока, принимает один аргумент типа *void* *, который передается обработчику при регистрации. Как следствие, возможно использование одного и того же обработчика для очистки разных экземпляров ресурса.

Особо отметим, что обработчик очистки – это временная мера. Он требуется в тех случаях, когда поток завершается либо отменяется, не успев выполнить определенный участок программного кода. При нормальных обстоятельствах ресурс должен освобождаться явным образом.

Для регистрации обработчика нужно вызвать функцию *pthread_cleanup_push()*. Ей передается указатель на обработчик и значение аргумента. Очевидно, что каждому такому вызову должен соответствовать вызов функции *pthread_cleanup_pop()*. Он отменяет регистрацию обработчика. Для удобства эта функция принимает целочисленный флаг. Когда он не равен нулю, при отмене регистрации выполняется очистка. СЛАЙД 24.

В листинге кода, представленном на СЛАЙДЕ 25, обработчик очистки применяется для удаления динамического буфера при завершении потока.

Здесь функции *pthread_cleanup_pop()* передается аргумент с ненулевым значением, поэтому функция очистки вызывается автоматически, хотя в данном случае достаточным оказался единственный вызов стандартной функции *free()*.

## Литература и дополнительные источники к Теме 3

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.