# 13. Потоки

Многопоточность является естественным продолжением многозадачности, точно также как виртуальные машины, позволяющие запускать несколько ОС на одном компьютере, представляют собой логическое развитие концепции разделения ресурсов. В рамках неформального, но простого, определения, поток - это выполнение последовательности машинных инструкций. В многопоточном приложении одновременно работает несколько потоков. Некоторые авторы избегают термина <поток> и используют вместо него термин <нить>, вероятно для того, чтобы потоки программ не путались с потоками ввода-вывода. Для обозначения последовательного выполнения цепочки инструкций мне лично больше нравится термин <поток>, которым я и буду пользоваться. Надеюсь, читатели Linux Format не запутаются в контекстах и, встретив слово поток, всегда поймут, идет ли речь о потоках программы, потоках ввода вывода, или о бурных паводковых потоках.

Прежде чем приступать к программированию потоков, следует ответить на вопрос, а нужны ли они вам. Мы уже знаем, насколько хорошо развиты в Linux средства межпроцессного взаимодействия. С помощью управления процессами в Linux можно решить многие задачи, которые в других ОС решаются только с помощью потоков. Потоки часто становятся источниками программных ошибок особого рода. Эти ошибки возникают при использовании потоками разделяемых ресурсов системы (например, общего адресного пространства) и являются частным случаем более широкого класса ошибок - ошибок синхронизации. Если задача разделена между независимыми процессами, то доступом к их общим ресурсам управляет операционная система, и вероятность ошибок из-за конфликтов доступа снижается. Впрочем, разделение задачи между несколькими независимыми процессами само по себе не защитит вас от других разновидностей ошибок синхронизации. В пользу потоков можно указать то, что накладные расходы на создание нового потока в многопоточном приложении ниже, чем накладные расходы на создание нового самостоятельного процесса. Уровень контроля над потоками в многопоточном приложении выше, чем уровень контроля приложения над дочерними процессами. Кроме того, многопоточные программы не склонны оставлять за собой вереницы зомби или <осиротевших> независимых процессов.

Первая подсистема потоков в Linux появилась около 1996 года и называлась, без лишних затей, - LinuxThreads. Библиотека LinuxThreads была попыткой организовать поддержку потоков в Linux в то время, когда ядро системы еще не предоставляло никаких специальных механизмов для работы с потоками. Позднее разработку потоков для Linux вели сразу две конкурирующие группы - NGPT и NPTL. В 2002 году группа NGPT фактически присоединилась к NPTL и теперь реализация потоков NPTL является стандартом Linux. Подсистема потоков Linux стремится соответствовать требованиям стандартов POSIX, так что новые многопоточные приложения Linux должны без проблем компилироваться на новых POSIX-совместимых системах.

## 13.1. Потоки и процессы

Тем, кто впервые познакомился с концепцией потоков, изучая программирование для Windows, модель потоков Linux покажется непривычной. В среде Microsoft Windows процесс, - это контейнер для потоков (именно этими словами о процессах говорит Джефри Рихтер в своей классической книге <Программирование приложений для Microsoft Windows>). Процесс-контейнер содержит как минимум один поток. Если потоков в процессе несколько, приложение (процесс) становится многопоточным. В мире Linux все выглядит иначе. В Linux каждый поток является процессом, и для того, чтобы создать новый поток, нужно создать новый процесс. В чем же, в таком случае, заключается преимущество многопоточности Linux перед многопроцессностью? В многопоточных приложениях Linux для создания

дополнительных потоков используются процессы особого типа. Эти процессы представляют собой обычные дочерние процессы главного процесса, но они разделяют с главным процессом адресное пространство, файловые дескрипторы и обработчики сигналов. Для обозначения процессов этого типа, применяется специальный термин - легкие процессы (lightweight processes). Прилагательное <легкий> в названии процессов- потоков вполне оправдано. Поскольку этим процессам не нужно создавать собственную копию адресного пространства (и других ресурсов) своего процесса- родителя, создание нового легкого процесса требует значительно меньших затрат, чем создание полновесного дочернего процесса. Поскольку потоки Linux на самом деле представляют собой процессы, в мире Linux нельзя говорить, что один процесс содержит несколько потоков. Если вы скажете это, в вас тут же заподозрят вражеского лазутчика!

Интересно рассмотреть механизм, с помощью которого Linux решает проблему идентификаторов процессов потоков. В Linux у каждого процесса есть идентификатор. Есть он, естественно, и у процессов-потоков. С другой стороны, спецификация POSIX 1003.1c требует, чтобы все потоки многопоточного приложения имели один идентификатор. Вызвано это требование тем, что для многих функций системы многопоточное приложение должно представляться как один процесс с одним идентификатором. Проблема единого идентификатора решается в Linux весьма элегантно. Процессы многопоточного приложения группируются в группы потоков (thread groups). Группе присваивается идентификатор, соответствующий идентификатору первого процесса многопоточного приложения. Именно этот идентификатор группы потоков используется при <общении> с многопоточным приложением. Функция getpid(2), возвращает значение идентификатора группы потока, независимо от того, из какого потока она вызвана. Функции kill() waitpid() и им подобные по умолчанию также используют идентификаторы групп потоков, а не отдельных процессов. Вам вряд ли понадобится узнавать собственный идентификатор процесса-потока, но если вы захотите это сделать, вам придется воспользоваться довольно экзотичной конструкцией. Получить идентификатор потока (thread ID) можно с помощью функции gettid(2), однако саму функцию нужно еще определить с помощью макроса _syscall. Работа с функцией gettid() выглядит примерно так:

```
#include <sys/types.h>
#include <linux/unistd.h>
...
_syscall0(pid_t,gettid);
...
pid_t my_tid;
my_tid = gettid();
```

Более подробную информацию вы можете получить на страницах man, посвященных gettid() и _syscall. Потоки создаются функцией pthread_create(3), определенной в заголовочном файле <pthread.h>. Первый параметр этой функции представляет собой указатель на переменную типа pthread_t, которая служит идентификатором создаваемого потока. Второй параметр, указатель на переменную типа pthread_attr_t, используется для передачи атрибутов потока. Третьим параметром функции pthread_create() должен быть адрес функции потока. Эта функция играет для потока ту же роль, что функция main() - для главной программы. Четвертый параметр функции pthread_create() имеет тип void *. Этот параметр может использоваться для передачи значения, возвращаемого функцией потока. Вскоре после вызова pthread_create() функция потока будет запущена на выполнение параллельно с другими потоками программы. Таким образом, собственно, и создается новый поток. Я говорю, что новый поток запускается <вскоре> после вызова pthread_create() потому, что перед тем как запустить новую функцию потока, нужно выполнить некоторые подготовительные действия, а поток-родитель между тем продолжает выполняться. Непонимание этого факта может привести вас к ошибкам, которые трудно будет обнаружить.

Если в ходе создания потока возникла ошибка, функция pthread_create() возвращает ненулевое значение, соответствующее номеру ошибки.

Функция потока должна иметь заголовок вида:

```
void * func_name(void * arg)
```

Имя функции, естественно, может быть любым. Аргумент arg, - это тот самый указатель, который передается в последнем параметре функции pthread_create(). Функция потока может вернуть значение, которое затем будет проанализировано заинтересованным потоком, но это не обязательно. Завершение функции потока происходит если:

a. функция потока вызвала функцию pthread_exit(3);
b. функция потока достигла точки выхода;
c. поток был досрочно завершен другим потоком.

Функция pthread_exit() представляет собой потоковый аналог функции _exit(). Аргумент функции pthread_exit(), значение типа void *, становится возвращаемым значением функции потока. Как (и кому?) функция потока может вернуть значение, если она не вызывается из программы явным образом? Для того, чтобы получить значение, возвращенное функцией потока, нужно воспользоваться функцией pthread_join(3). У этой функции два параметра. Первый параметр pthread_join(), - это идентификатор потока, второй параметр имеет тип <указатель на нетипизированный указатель>. В этом параметре функция pthread_join() возвращает значение, возвращенное функцией потока. Конечно, в многопоточном приложении есть и более простые способы организовать передачу данных между потоками. Основная задача функции pthread_join() заключается, однако, в синхронизации потоков. Вызов функции pthread_join() приостанавливает выполнение вызвавшего ее потока до тех пор, пока поток, чей идентификатор передан функции в качестве аргумента, не завершит свою работу. Если в момент вызова pthread_join() ожидаемый поток уже завершился, функция вернет управление немедленно. Функцию pthread_join() можно рассматривать как эквивалент waitpid(2) для потоков. Эта функция позволяет вызвавшему ее потоку дождаться завершения работы другого потока. Попытка выполнить более одного вызова pthread_join() (из разных потоков) для одного и того же потока приведет к ошибке.

Посмотрим, как все это работает на практике. Ниже приводится фрагмент листинга программы threads.c.

```
#include <stdlib.h>
#include <stdio.h>
#include <errno.h>
#include <pthread.h>
void * thread_func(void *arg)
{ int i;
int loc_id = * (int *) arg;
for (i = 0; i < 4; i++) {
printf("Thread %i is running\n", loc_id);
sleep(1);
}
}
int main(int argc, char * argv[])
{ int id1, id2, result;
pthread_t thread1, thread2;
id1 = 1;
result = pthread_create(&thread1, NULL, thread_func, &id1);
if (result != 0) {
perror("Creating the first thread");
return EXIT_FAILURE;
}
id2 = 2;
```

```
result = pthread_create(&thread2, NULL, thread_func, &id2);
if (result != 0) {
perror("Creating the second thread");
return EXIT_FAILURE;
}
result = pthread_join(thread1, NULL);
if (result != 0) {
perror("Joining the first thread");
return EXIT_FAILURE;
}
result = pthread_join(thread2, NULL);
if (result != 0) {
perror("Joining the second thread");
return EXIT_FAILURE;
}
printf("Done\n");
return EXIT_SUCCESS;
}
```

Рассмотрим сначала функцию thread_func(). Как вы, конечно, догадались, это и есть функция потока. Наша функция потока очень проста. В качестве аргумента ей передается указатель на переменную типа int, в которой содержится номер потока. Функция потока распечатывает этот номер несколько раз с интервалом в одну секунду и завершает свою работу. В функции main() вы видите две переменных типа pthread_t. Мы собираемся создать два потока и у каждого из них должен быть свой идентификатор. Вы также видите две переменные типа int, id1 и id2, которые используются для передачи функциям потоков их номеров. Сами потоки создаются с помощью функции pthread_create().В этом примере мы не модифицируем атрибуты потоков, поэтому во втором параметре в обоих случаях передаем NULL. Вызывая pthread_create() дважды, мы оба раза передаем в качестве третьего параметра адрес функции thread_func, в результате чего два созданных потока будут выполнять одну и ту же функцию. Функция, вызываемая из нескольких потоков одновременно, должна обладать свойством реентерабельности (этим же свойством должны обладать функции, допускающие рекурсию). Реентерабельная функция, это функция, которая может быть вызвана повторно, в то время, когда она уже вызвана (отсюда и происходит ее название). Реентерабельные функции используют локальные переменные (и локально выделенную память) в тех случаях, когда их не-реентерабельные аналоги могут воспользоваться глобальными переменными.

Мы вызываем последовательно две функции pthread_join() для того, чтобы дождаться завершения обоих потоков. Если мы хотим дождаться завершения всех потоков, порядок вызова функций pthread_join() для разных потоков, очевидно, не имеет значения.

Для того, чтобы скомпилировать программу threads.c, необходимо дать команду:

```
gcc threads.c -D_REENTERANT -lpthread -o threads
```

Команда компиляции включает макрос _REENTERANT. Этот макрос указывает, что вместо обычных функций стандартной библиотеки к программе должны быть подключены их реентерабельные аналоги. Реентерабельный вариант библиотеки glibc написан таким образом, что вы, скорее всего, вообще не обнаружите никаких различий в работе с реентерабельными функциями по сравнению с их обычными аналогами. Наконец, мы указываем компоновщику, что программа должна быть связана с библиотекой libpthread, которая содержит все специальные функции, необходимые для работы с потоками.

У вас, возможно, возникает вопрос, зачем мы использовали две разные переменные, id1 и id2, для передачи значений двум потокам? Почему нельзя использовать одну переменную, скажем id, для обоих потоков? Рассмотрим такой фрагмент кода:

```
id = 1;
```

```
pthread_create(&thread1, NULL, thread_func, &id);
id = 2;
pthread_create(&thread2, NULL, thread_func, &id);
```

Конечно, в этом случае оба потока получат указатель на одну и ту же переменную, но ведь значение этой переменной нужно каждому потоку только в самом начале его работы. После того, как поток присвоит это значение своей локальной переменной loc_id, ничто не мешает нам использовать ту же переменную id для другого потока. Все это верно, но проблема заключается в том, что мы не знаем, когда первый поток начнет свою работу. То, что функция pthread_create() вернула управление, не гарантирует нам, что поток уже выполняется. Вполне может случиться так, что первый поток будет запущен уже после того, как переменной id будет присвоено значение 2. Тогда оба потока получат одно и то же значение id. Впрочем, мы можем использовать одну и ту же переменную для передачи данных функциям потока, если воспользуемся средствами синхронизации. Этим средствам будет посвящена следующая статья.

## 13.2. Досрочное завершение потока

Функции потоков можно рассматривать как вспомогательные программы, находящиеся под управлением функции main(). Точно так же, как при управлении процессами, иногда возникает необходимость досрочно завершить процесс, многопоточной программе может понадобиться досрочно завершить один из потоков. Для досрочного завершения потока можно воспользоваться функцией pthread_cancel(3). Единственным аргументом этой функции является идентификатор потока. Функция pthread_cancel() возвращает 0 в случае успеха и ненулевое значение в случае ошибки. Несмотря на то, что pthread_cancel() может завершить поток досрочно, ее нельзя назвать средством принудительного завершения потоков. Дело в том, что поток может не только самостоятельно выбрать порядок завершения в ответ на вызов pthread_cancel(), но и вовсе игнорировать этот вызов. Вызов функции pthread_cancel() следует рассматривать как запрос на выполнение досрочного завершения потока. Функция pthread_setcancelstate(3) определяет, будет ли поток реагировать на обращение к нему с помощью pthread_cancel(), или не будет. У функции pthread_setcancelstate() два параметра, параметр state типа int и параметр oldstate типа <указатель на int>. В первом параметре передается новое значение, указывающее, как поток должен реагировать на запрос pthread_cancel(), а в переменную, чей адрес был передан во втором параметре, функция записывает прежнее значение. Если прежнее значение вас не интересует, во втором параметре можно передать NULL.

Чаще всего функция pthread_setcancelstate() используется для временного запрета завершения потока. Допустим, мы программируем поток, и знаем, что при определенных условиях программа может потребовать его досрочного завершения. Но в нашем потоке есть участок кода, во время выполнения которого завершать поток крайне нежелательно. Мы можем оградить этот участок кода от досрочного завершения с помощью пары вызовов pthread_setcancelstate():

```
pthread_setcancelstate(PTHREAD_CANCEL_DISABLE, NULL);
... //Здесь поток завершать нельзя
pthread_setcancelstate(PTHREAD_CANCEL_ENABLE, NULL);
```

Первый вызов pthread_setcancelstate() запрещает досрочное завершение потока, второй - разрешает. Если запрос на досрочное завершение потока поступит в тот момент, когда поток игнорирует эти запросы, выполнение запроса будет отложено до тех пор, пока функция pthread_setcancelstate() не будет вызвана с аргументом PTHREAD_CANCEL_ENABLE. Что именно произойдет дальше, зависит от более тонких настроек потока. Рассмотрим пример программы (на диске вы найдете ее в файле canceltest.c)

```
#include <stdlib.h>
#include <stdio.h>
#include <pthread.h>
int i = 0;
void * thread_func(void *arg)
{ pthread_setcancelstate(PTHREAD_CANCEL_DISABLE, NULL);
for (i=0; i < 4; i++) {
sleep(1);
printf("I'm still running!\n");
}
pthread_setcancelstate(PTHREAD_CANCEL_ENABLE, NULL);
pthread_testcancel();
printf("YOU WILL NOT STOP ME!!!\n");
}
int main(int argc, char * argv[])
{ pthread_t
thread;
pthread_create(&thread, NULL, thread_func, NULL);
while (i < 1) sleep(1);
pthread_cancel(thread);
printf("Requested to cancel the thread\n");
pthread_join(thread, NULL);
printf("The thread is stopped.\n");
return EXIT_SUCCESS;
}
```

В самом начале функции потока thread_func() мы запрещаем досрочное завершение потока, затем выводим четыре тестовых сообщения с интервалом в одну секунду, после чего разрешаем досрочное завершение. Далее, с помощью функции pthread_testcancel(), мы создаем точку отмены (cancellation point) потока. Если досрочное завершение потока было затребовано, в этот момент поток должен завершиться. Затем мы выводим еще одно диагностическое сообщение, которое пользователь не должен видеть, если программа сработает правильно.

В главной функции программы мы создаем поток, затем дожидаемся, пока значение глобальной переменной i станет больше нуля (это гарантирует нам, что поток уже запретил досрочное завершение) и вызываем функцию pthread_cancel(). После этого мы переходим к ожиданию завершения потока с помощью pthread_join(). Если вы скомпилируете и запустите программу, то увидите, что поток распечатает четыре тестовых сообщения I'm still running! (после первого сообщения главная функция программы выдаст запрос на завершение потока).

Поскольку поток завершится досрочно, последнего тестового сообщения вы не увидите. Интересна роль функции pthread_testcancel(). Как уже отмечалось, эта функция создает точку отмены потока. Зачем нужны особые точки отмены? Дело в том, что даже если досрочное завершение разрешено, поток, получивший запрос на досрочное завершение, может завершить работу не сразу. Если поток находится в режиме отложенного досрочного завершения (именно этот режим установлен по умолчанию), он выполнит запрос на досрочное завершение, только достигнув одной из точек отмены. В соответствии со стандартом POSIX, точками отмены являются вызовы многих <обычных> функций, например open(), pause() и write(). Про функцию printf() в документации сказано, что она может быть точкой отмены, но в Linux при попытке остановиться на printf() происходит нечто странное - поток завершается, но pthread_join() не возвращает управления. Поэтому мы создаем явную точку отмены с помощью вызова pthread_testcancel().

Впрочем, мы можем выполнить досрочное завершение потока, не дожидаясь точек останова. Для этого необходимо перевести поток в режим немедленного завершения, что делается с помощью вызова pthread_setcanceltype(PTHREAD_CANCEL_ASYNCHRONOUS, NULL); В

этом случае беспокоиться о точках останова уже не нужно. Вызов pthread_setcanceltype(PTHREAD_CANCEL_DEFERRED, NULL); снова переводит поток в режим отложенного досрочного завершения.

**Литература:**

D. P. Bovet, M. Cesati, Understanding the Linux Kernel, 3rd Edition, O'Reilly, 2005
W. R. Stevens, S. A. Rago, Advanced Programming in the UNIX® Environment: Second Edition, Addison Wesley Professional, 2005