14. Потоки - продолжение

Мы продолжаем знакомство с многопоточностью в Linux и уже знаем, что если запрос на досрочное завершение потока поступил в неподходящий момент, поток может отложить досрочное завершение до тех пор, пока не станет готов к нему.

Механизм отложенного досрочного завершения очень полезен, но для действительно эффективного управления завершением потоков необходим еще и механизм, оповещающий поток о досрочном завершении. Оповещение о завершении потоков в Unix-системах реализовано на основе тех же принципов, что и оповещение о завершении самостоятельных процессов. Если нам нужно выполнять какие-то специальные действия в момент завершения потока (нормального или досрочного), мы устанавливаем функцию-обработчик, которая будет вызвана перед тем, как поток завершит свою работу. Для потоков наличие обработчика завершения даже более важно, чем для процессов. Предположим, что поток выделяет блок динамической памяти и затем внезапно завершается по требованию другого потока. Если бы поток был самостоятельным процессом, ничего особенно неприятного не случилось бы, так как система сама убрала бы за ним мусор. В случае же процесса-потока не высвобожденный блок памяти так и останется <висеть> в адресном пространстве многопоточного приложения. Если потоков много, а ситуации, требующие досрочного завершения, возникают часто, утечки памяти могут оказаться значительными. Устанавливая обработчик завершения потока, высвобождающий занятую память, мы можем быть уверены, что поток не оставит за собой бесхозных блоков памяти (если, конечно, в системе не случится какого-то более серьезного сбоя).

Для установки обработчика завершения потока применяется макрос pthread_cleanup_push(3). Подчеркиваю жирной красной чертой, pthread_cleanup_push() - это **макрос, а не функция**. Неправильное использование макроса pthread_cleanup_push() может привести к неожиданным синтаксическим ошибкам. У макроса pthread_cleanup_push() два аргумента. В первом аргументе макросу должен быть передан адрес функции-обработчика завершения потока, а во втором - нетипизированный указатель, который будет передан как аргумент при вызове функции-обработчика. Этот указатель может указывать на что угодно, мы сами решаем, какие данные должны быть переданы обработчику завершения потока. Макрос pthread_cleanup_push() помещает переданные ему адрес функции- обработчика и указатель в специальный стек. Само слово <стек> указывает, что мы можем назначить потоку произвольное число функций-обработчиков завершения. Поскольку в стек записывается не только адрес функции, но и ее аргумент, мы можем назначить один и тот же обработчик с несколькими разными аргументами.

В процессе завершения потока функции-обработчики и их аргументы должны быть извлечены из стека и выполнены. Извлечение обработчиков из стека и их выполнение может производиться либо явно, либо автоматически. Автоматически обработчики завершения потока выполняются при вызове потоком функции pthread_exit(), завершающей работу потока, а также при выполнении потоком запроса на досрочное завершение. Явным образом обработчики завершения потока извлекаются из стека с помощью макроса pthread_cleanup_pop(3). Во всех случаях обработчики извлекаются из стека (и выполняются) в порядке, противоположном тому, в котором они были помещены в стек. Если мы используем макрос pthread_cleanup_pop() явно, мы можем указать, что обработчик необходимо только извлечь из стека, но выполнять его не следует. Мы рассмотрим методы назначения и выполнения обработчиков завершения потока на простом примере (программа exittest):

```
#include <stdlib.h>
#include <stdio.h>
#include <errno.h>
#include <pthread.h>
```

```
void exit_func(void * arg)
{ free(arg);
printf("Freed the allocated memory.\n");
}
void * thread_func(void *arg)
{ int i;
void * mem;
pthread_setcancelstate(PTHREAD_CANCEL_DISABLE, NULL);
mem = malloc(1024);
printf("Allocated some memory.\n");
pthread_cleanup_push(exit_func, mem);
pthread_setcancelstate(PTHREAD_CANCEL_ENABLE, NULL);
for (i = 0; i < 4; i++) {
sleep(1);
printf("I'm still running!!!\n");
}
pthread_cleanup_pop(1);
}
int main(int argc, char * argv[])
{ pthread_t
thread;
pthread_create(&thread, NULL, thread_func, NULL);
pthread_cancel(thread);
pthread_join(thread, NULL);
printf("Done.\n");
return EXIT_SUCCESS;
}
```

Главная функция программы не содержит ничего такого, что не было бы нам уже знакомо. Мы создаем новый поток и тут же посылаем запрос на его завершение. Все новые элементы программы exittest сосредоточены в функции потока thread_func(). Поток начинает работу с того, что запрещает свое досрочное завершение. Этот запрет необходим на время выполнения важных действий, которые нельзя прерывать. Если запрос на досрочное завершение поступит во время действия запрета, он не пропадет. Как мы уже знаем, запрет досрочного завершения не отменяет выполнение соответствующего запроса, а только откладывает его. Далее поток выделяет блок памяти. Для того чтобы избежать утечек памяти, мы должны гарантировать высвобождение выделенного блока, для чего, как вы уже догадались, мы используем функцию-обработчик завершения потока exit_func(). Мы добавляем функцию exit_func() в стек обработчиков завершения потока с помощью макроса pthread_cleanup_push(). Обратите внимание на второй параметр макроса. Вторым параметром, как мы знаем, должен быть нетипизированный указатель. Этот указатель будет передан функции-обработчику в качестве аргумента. Поскольку задача функции exit_func() заключается в том, чтобы высвободить блок памяти mem, в качестве аргумента функции мы устанавливаем указатель на этот блок. Функция exit_func() высвобождает блок памяти с помощью функции free(3) и выводит диагностическое сообщение. После установки обработчика завершения потока наш поток разрешает досрочное завершение. Теперь вам должно быть понятно, зачем мы запретили досрочное завершение потока во время этих операций. Если бы поток завершился после выделения блока памяти, но до назначения функции-обработчика, выделенный блок не был бы удален. Далее поток выводит четыре диагностических сообщения с интервалом в одну секунду и завершает работу.

Перед выходом из функции потока мы вызываем макрос pthread_cleanup_pop(). Этот макрос извлекает функцию-обработчик из стека. Аргумент макроса pthread_cleanup_pop() позволяет указать, следует ли выполнять функцию- обработчик, или требуется только удалить ее из стека. Мы передаем макросу ненулевое значение, что указывает на необходимость выполнить обработчик. Если вы забудете поставить вызов pthread_cleanup_pop() в конце функции потока, компилятор выдаст сообщение о синтаксической ошибке. Объясняется это, конечно,

тем, что pthread_cleanup_push() и pthread_cleanup_pop() - макросы. Первый макрос, кроме прочего, открывает фигурные скобки, которые второй макрос должен закрыть, так что число обращений к pthread_cleanup_push() в функции потока всегда должно быть равно числу обращений к pthread_cleanup_pop(), иначе программу не удастся скомпилировать.

То, что макрос pthread_cleanup_pop() должен быть вызван столько же раз, сколько и макрос pthread_cleanup_push(), очень удобно в том случае, если обработчик завершения потока вызывается явным образом, но что происходит, если обработчики завершения потока вызываются неявно? Если неявный вызов обработчиков происходит вследствие досрочного завершения потока, механизм досрочного завершения вызовет обработчики сам, а код, добавленный макросами pthread_cleanup_pop(), выполнен не будет. Однако обработчики завершения потока могут быть выполнены и в результате вызова функции pthread_exit(). Наличие вызова pthread_exit() не избавит вас от необходимости добавлять макросы pthread_cleanup_pop(), ведь они необходимы для охранения правильной синтаксической структуры программы. Как же функция pthread_exit() взаимодействует с кодом, добавленным макросами pthread_cleanup_pop()? Если вызов pthread_exit() расположен до вызовов pthread_cleanup_pop(), поток завершится до обращения к коду макросов, при этом все обработчики завершения потока будут вызваны функцией pthread_exit(). Если мы расположим вызов pthread_exit() после вызовов pthread_cleanup_pop(), обработчики завершения будут выполнены до вызова pthread_exit(), и этой функции останется только завершить работу потока, не вызывая никаких обработчиков. А нужно ли вообще вызывать pthread_exit() в конце функции потока, если вызовы макросов pthread_cleanup_pop() все равно необходимы? Ответ на этот вопрос зависит от обстоятельств. Помимо вызова обработчиков завершения потока, функция pthread_exit() может выполнять в вашем потоке и другие важные действия, и в этом случае ее вызов необходим.

Еще один тонкий момент связан с выходом из функции потока с помощью оператора return. Сам по себе такой выход из функции потока не приводит к вызову обработчиков завершения. В нашем примере мы вызвали обработчики явно с помощью pthread_cleanup_pop() непосредственно перед выполнением return, но рассмотрим другой вариант функции thread_func():

```
void * thread_func(void *arg)
{ int i;
void * mem;
pthread_setcancelstate(PTHREAD_CANCEL_DISABLE, NULL);
mem = malloc(1024);
printf("Allocated some memory.\n");
pthread_cleanup_push(exit_func, mem);
pthread_setcancelstate(PTHREAD_CANCEL_ENABLE, NULL);
for (i = 0; i < 4; i++) {
sleep(1);
printf("I'm still running!!!\n");
if (i == 2) return;
}
pthread_cleanup_pop(1);
return;
}
```

Пусть этот вариант выглядит несколько неестественно, суть его в том, что теперь в функции потока определено несколько точек выхода. При выполнении условия i == 2 функция потока завершится в результате выполнения оператора return и обработчик завершения потока при этом вызван не будет. Эту проблему нельзя решить добавлением еще одного макроса pthread_cleanup_pop(). Вариант функции

```
...
if (i == 2) {
```

```
pthread_cleanup_pop(1);
return;
}
pthread_cleanup_pop(1);
return;
}
```

вообще не скомпилируется, поскольку лишний макрос pthread_cleanup_pop() нарушит синтаксис программы. Правильное решение заключается в использовании функции pthread_exit() вместо return:

```
if (i == 2) pthread_exit(0);
```

Вполне возможно, что вам, уважаемый читатель, как и мне, уже несколько раз хотелось досрочно завершить обсуждение досрочного завершения потоков. Потерпите немного, мы уже приближаемся к финишу. Осталось ответить на вопрос, зачем нам нужна возможность устанавливать несколько обработчиков завершения потока? Ответов на этот вопрос может быть много, но я дам только один. Представьте себе, что вы программируете сложную функцию потока, которая интенсивно работает с динамической памятью. Как только в вашей функции выделяется новый блок памяти, вы устанавливаете обработчик завершения потока, который высвободит этот блок в случае неожиданного завершения. Тут стоит отвлечься на секунду и заметить, что установка обработчика, высвобождающего память во время завершения потока, не мешает вам самостоятельно высвободить эту память, когда она перестанет быть нужна. Придется только немного поиграть с указателями (на диске вы найдете программу exittest2.c, которая демонстрирует явное высвобождение памяти в потоке совместно с использованием обработчика завершения). Если затем в вашей функции понадобится выделить новый блок памяти, потребуется еще один обработчик для его высвобождения. Даже если вы заранее знаете, сколько раз ваша программа будет выделать блоки памяти, назначать обработчик для высвобождения каждого блока можно только после того, как блок выделен.

14.1. Средства синхронизации потоков

Изучая взаимодействие между процессами, мы много внимания уделили средствам синхронизации процессов. У потоков тоже есть свои специальные средства синхронизации. Вернемся к первому примеру из предыдущей статьи, - программе threads. Напомню, что в том примере мы создавали два потока, используя одну и ту же функцию потока. В процессе создания каждого потока этой функции передавалось целочисленное значение (номер потока). При этом для передачи значения каждому потоку использовалась своя переменная. В той статье я подробно объяснил, почему использование одной и той же переменной для передачи значения функциям разных потоков может привести к ошибке.

Коротко говоря, - проблема заключалась в том, что мы не могли знать, когда именно новый поток начнет выполняться. С помощью средств синхронизации потоков мы можем решить эту проблему и использовать одну переменную для передачи значений обоим потокам. Рассмотрим модифицированный вариант программы threads, программу threads2.c.

```
#include <stdlib.h>
#include <stdio.h>
#include <errno.h>
#include <pthread.h>
#include <semaphore.h>
sem_t sem;
void * thread_func(void *arg)
{ int i;
int loc_id = * (int *) arg;
sem_post(&sem);
```

```
for (i = 0; i < 4; i++) {
printf("Thread %i is running\n", loc_id);
sleep(1);
}
}
int main(int argc, char * argv[])
{ int id, result;
pthread_t thread1, thread2;
id = 1;
sem_init(&sem, 0, 0);
result = pthread_create(&thread1, NULL, thread_func, &id);
if (result != 0) {
perror("Creating the first thread");
return EXIT_FAILURE;
}
sem_wait(&sem);
id = 2;
result = pthread_create(&thread2, NULL, thread_func, &id);
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;
}
sem_destroy(&sem);
printf("Done\n");
return EXIT_SUCCESS;
}
```

В новом варианте программы мы используем одну переменную id для передачи значения обоим потокам. Если вы скомпилируете и запустите программу threads2, то увидите, что она работает корректно. Секрет нашего успеха заключается в использовании средств синхронизации. Для синхронизации потоков мы задействовали семафоры. Читатели этой серии статей уже знакомы с семафорами System V, предназначенными для синхронизации процессов. В данном случае мы применяем семафоры другого типа - семафоры POSIX, которые специально предназначены для работы с потоками. Все объявления функций и типов, относящиеся к этим семафорам, можно найти в файле semaphore.h. Семафоры POSIX создаются (инициализируются) с помощью функции sem_init(3). Первый параметр функции sem_init() - указатель на переменную типа sem_t, которая служит идентификатором семафора. Второй параметр - pshared - в настоящее время не используется, и мы оставим его равным нулю. В третьем параметре функции sem_init() передается значение, которым инициализируется семафор. Дальнейшая работа с семафором осуществляется с помощью функций sem_wait(3) и sem_post(3). Единственным аргументом функции sem_wait() служит указатель на идентификатор семафора. Функция sem_wait() приостанавливает выполнение вызвавшего ее потока до тех пор, пока значение семафора не станет большим нуля, после чего функция уменьшает значение семафора на единицу и возвращает управление. Функция sem_post() увеличивает значение семафора, идентификатор которого был передан ей в качестве параметра, на единицу. Присвоив семафору значение 0, наша программа создает первый поток и вызывает функцию sem_wait(). Эта функция приостановит выполнение функции main() до тех пор, пока функция потока не вызовет функцию sem_post(), а это

случится только после того как функция потока обработает значение переменной id. Таким образом, мы можем быть уверены, что в момент создания второго потока первый поток уже закончит работу с переменной id, и мы сможем использовать эту переменную для передачи данных второму потоку. После завершения обоих потоков мы вызываем функцию sem_destroy(3) для удаления семафора и высвобождения его ресурсов.

Семафоры - не единственное средство синхронизации потоков. Для разграничения доступа к глобальным объектам потоки могут использовать мьютексы. Все функции и типы данных, имеющие отношение к мьютексам, определены в файле pthread.h. Мьютекс создается вызовом функции pthread_mutex_init(3). В качестве первого аргумента этой функции передается указатель на переменную pthread_mutex_t, которая играет роль идентификатора нового мьютекса. Вторым аргументом функции pthread_mutex_init() должен быть указатель на переменную типа pthread_mutexattr_t. Эта переменная позволяет установить дополнительные атрибуты мьютекса. Если нам нужен обычный мьютекс, мы можем передать во втором параметре значение NULL. Для того чтобы получить исключительный доступ к некоему глобальному ресурсу, поток вызывает функцию pthread_mutex_lock(3), (в этом случае говорят, что <поток захватывает мьютекс>). Единственным параметром функции pthread_mutex_lock() должен быть идентификатор мьютекса. Закончив работу с глобальным ресурсом, поток высвобождает мьютекс с помощью функции pthread_mutex_unlock(3), которой также передается идентификатор мьютекса. Если поток вызовет функцию pthread_mutex_lock() для мьютекса, уже захваченного другим потоком, эта функция не вернет управление до тех пор, пока другой поток не высвободит мьютекс с помощью вызова pthread_mutex_unlock() (после этого мьютекс, естественно, перейдет во владение нового потока). Удаление мьютекса выполняется с помощью функции pthread_mutex_destroy(3). Стоит отметить, что в отличие от многих других функций, приостанавливающих работу потока, вызов pthread_mutex_lock() не является точкой останова. Иначе говоря, поток, находящийся в режиме отложенного досрочного завершения, не может быть завершен в тот момент, когда он ожидает выхода из pthread_mutex_lock().

14.2. Атрибуты потоков

Создавая новый поток, вы можете указать ряд дополнительных атрибутов, определяющих некоторые его параметры. Из всех этих атрибутов более всего востребован атрибут DETACHED, позволяющий создавать отделенные потоки. Во всех рассмотренных выше примерах мы использовали функцию pthread_join(), позволяющую дождаться завершения потока и получить значение, возвращенное его функцией. Для того чтобы функция pthread_join() могла получить значение функции потока, завершившегося до вызова pthread_join(), система сохраняет данные о потоке после его завершения (это похоже на появления <зомби> после завершения самостоятельного процесса). Если наша программа интенсивно работает с потоками и синхронизация потоков с помощью pthread_join() нам не нужна, мы можем сэкономить ресурсы системы, используя отделенные потоки. Отделенные потоки отличаются от обычных (присоединяемых) потоков тем, что после завершения отделенного потока система не сохраняет информацию о нем. Если вызвать функцию pthread_join() для отделенного потока, она вернет сообщение об ошибке.

Вы можете превратить присоединяемый поток в отделенный с помощью вызова функции pthread_detach(3), однако придать потоку свойство <отделенности> можно и на этапе его создания, с помощью дополнительного атрибута DETACHED. Для того чтобы назначить потоку дополнительные атрибуты, нужно сначала создать объект, содержащий набор атрибутов. Этот объект создается функцией pthread_attr_init(3). Единственный аргумент этой функции - указатель на переменную типа pthread_attr_t, которая служит идентификатором набора атрибутов. Функция pthread_attr_init() инициализирует набор атрибутов потока значениями, заданными по умолчанию, так что мы можем модифицировать только те атрибуты, которые нас интересуют, и не беспокоиться об остальных. Для добавления

атрибутов в набор используются специальные функции с именами pthread_attr_set<имя_атрибута>. Например, для того, чтобы добавить атрибут <отделенности>, мы вызываем функцию pthread_attr_setdetachstate(3). Первым аргументом этой функции должен быть адрес объекта набора атрибутов, а вторым аргументом - константа, определяющая значение атрибута. Константа PTHREAD_CREATE_DETACHED указывает, что создаваемый поток должен быть отделенным, тогда как константа PTHREAD_CREATE_JOINABLE определяет создание присоединяемого (joinable) потока, который может быть синхронизирован функций pthread_join(3). После того, как мы добавили необходимые значения в набор атрибутов потока, мы вызываем функцию создания потока pthread_create(). Набор атрибутов потока передается в качестве второго аргумента этой функции.

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