08. Потоки и взаимодействие потоков

Егор Орлов

Курс: UNIX-DEV-SYS. Системное программирование в среде UNIX (Linux/FreeBSD). ВИШ СПбПУ, 2021

Содержание

1. Потоки, отличия потоков от процессов

1.1. Однопоточные и многопоточные процессы

- Поток существует внутри процесса, являясь более мелкой единицей исполнения, претендующей на ресурсы CPU, представляет из себя последовательно выполняемый программный

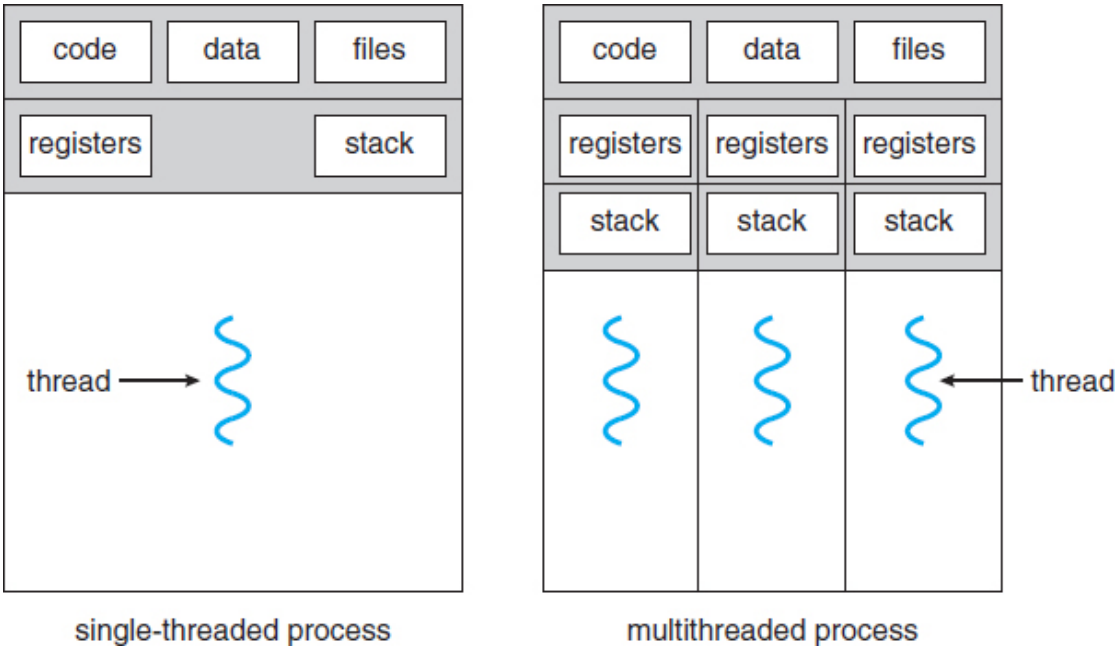

Рис. 1: Однопоточные и многопоточные процессы

код, который делит области памяти с другими потоками в рамках процесса.

- При запуске программы ОС Linux создает процесс, а в нем - единственный поток(thread, нить). Этот первый поток может дальше создавать еще потоки, а может не создавать.

1.2. Отличия потоков от процессов

- Тема сложная, особенно в случае с Linux-ом, где явной границы между ними нет.

  Для того, чтобы не запутаться, начнем с классики - т.е. POSIX threads, а потом посмотрим с какими особенностями это реализовано в Linux.

1.2.1. Порождение

- При создании дочернего процесса ОС копирует (не копирует, а помечает для необходимости копирования при изменении - CopyOnWrite) область памяти процесса в его потомка, создавая тем самым копии переменных, дескрипторов, прочих атрибутов.
- Когда программа создает поток, ничего никуда не копируется, просто создается сущность в ОС, претендующая на свою долю процессорного времени.

1.2.2. Жизнедеятельность

- В дочернем **процессе** - модификация содержимого памяти, например изменение переменных или вызом exec() и тем самым подмена всего образа процесса, переназначение дескрипторов, типа dup2() и т.п. - никак не влияет на процесс-родитель.

- В новом **потоке** - мы работаем с теми же самыми областями глобальной и динамической памяти, наболром файловых дескрипторов, что и в других потоках данного процесса. При этом контекст выполнения (регистры), стек (т.е. локальные переменные) у потока свои.

  Технически начало выполнение потока - это передача управление подпрограмме, с которой и формируется области памяти, собственные для данного потока. Т.е. стек у каждого потока свой. Стеки потоков напрямую никак не объявляются в других потоках, но все они находятся в одном общем адресном пространстве процесса, т.е. при желании к ним можно (но обычно не нужно) адресоваться из других потоков.

1.2.3. Обмен данными

- В случае **процессов** требуются специальные телодвижения, чтобы данные между процессами передать и, иногда добавляются еще и требования синхронизации, если способ передачи не предусматривает автоматическую синхронизацию.
- В случае **потоков** передать легко и просто (глобальная память и т.п.), на первое место выходит проблема синхронизации, взаимных блокировок, т.к. возможны ситуации гонок (race conditions)

1.2.4. Порождение дочерних процессов-потоков

- Процесс может порождать потоки
- При создании нового процесса из потока средствами fork() получается копия процесса со всеми потоками
- При вызове в каком-либо потоке exec() замещается образ процесса, т.е. все потоки, что были, заканчиваются.

1.2.5. Завершение работы

- Неординарное завершение работы дочернего процесса никак не влияет на родителя и другие процессы данного родителя. Например такие ситуации как ошибка сегментации или получение завершающего сигнала.
- Нарушение сегментации, произошедшее при выполнении кода потока уровнит весь процесс с другими потоками. Аналогично приходит завершающий сигнал и процесс со всеми своими потоками завершается.

1.3. Tread-safety - потокобезопастность, реентерабельность

- Многопоточное программирование предъявляет специальные требования к используемым в потоках функциям (а точнее к тому, как эти функции используют общие ресурсы, т.е. глобальные переменные)
- **Потокобезопасная функция** может быть вызвана одновременно из разных потоков, даже когда вызовы используют разделяемые данные, поскольку все обращения к разделяемым данным упорядочены.
- **Реентерабельная функция** также может быть вызвана одновременно из нескольких потоков, но только если каждый вызов использует свои собственные данные.

  Таким образом, потокобезопасная функция всегда реентерабельна, но реентерабельная функция не всегда потокобезопасна.

Зачастую под этими терминами понимают одно и то же, т.е. потокобезопасность.

Часто встречается - вот это реентерабельная функция, ей можно подсунуть локальные переменные (из стека потока) для хранения своих состояний и таким образом она станет потокобезопасной. Т.е. правильный вызов реентерабельных функций обеспечивает потокобезопасность.

1.3.1. Требование реентерабельности

1. Никакая часть вызываемого кода не должна модифицироваться;
2. Вызываемая процедура не должна сохранять информацию между вызовами;
3. Если процедура изменяет какие-либо данные, то они должны быть уникальными для каждого потока;
4. Процедура не должна возвращать указатели на объекты, общие для разных потоков.

- В общем случае, для обеспечения реентерабельности необходимо, чтобы вызывающий процесс или функция **каждый раз передавал вызываемому процессу все необходимые данные**.

  Таким образом, функция, которая **зависит только от своих параметров**, не использует глобальные и статические переменные и вызывает только реентерабельные функции, будет реентерабельной.

  Если функция использует глобальные или статические переменные, необходимо обеспечить их локальное хранение.

1.3.2. Примеры не реентерабельных функций

```
int g_var = 1;

int f() {
  g_var = g_var + 2;
  return g_var;
}

int g() {
  return f() + 2;
}
```

Если два процесса вызывают **f()** в одно и то же время, результат непредсказуем.

Поэтому, **f()** не реентерабельна. Но и **g()** не реентерабельна, поскольку она использует нереентерабельную функцию **f()**.

```
int accum(int b) {
  static int a = 0;
  ++a;
  return (a+b);
}
```

Функция накапливает значение в статической переменной и поэтому не является реентерабельной.

1.3.3. Пример strtok()/strtok_r()

- Обычная и реентерабельная функция, использующая контекст (сохраняющая состояние между вызовами)

```
#include <string.h>
char *strtok(char *str, const char *delim);
char *strtok_r(char *str, const char *delim, char **saveptr);
```

При вызове **strtok_r()** если **saveptr** указывает на данные в стеке, получаем потокобезопасный код. Если на глобальную переменную - то нет.

1.4. Преимущества и недостатки

1.4.1. Преимущества потоков перед процессами

- Меньшие накладные расходы на создание потока
- Более простой обмен данными
- Создавать проще и быстрее

1.4.2. Недостатки потоков в сравнении с процессами

- Необходимость обеспечения потоковой безопасности функций
- Ошибка в одном потоке может привезти к завершению процесса
- Необходимость синхронизации при доступе к общим областям памяти и ресурсам

2. Работа с потоками средствами библиотеки pthreads

2.1. POSIX Threads API

- Не является частью стандартной библиотеки C

```
$ cc -o main main.c -lpthread
```

- Заголовочный файл

```
#include <pthread.h>
```

2.2. Создание потока POSIX

- Идентификатор потока

```
pthread_t thread_id;
```

- Функция создания потока

```
#include <pthread.h>
int pthread_create(pthread_t *thread, const pthread_attr_t *attr,
                          void *(*start_routine)(void *), void *arg);
```

- **thread** - указатель на идентификатор потока, заполняется в результате вызова и используется в дальнейших вызовах pthreads API
- **attr** - указатель на структуру атрибутов потока **pthread_attr_t**, если NULL, то поток создается с параметрами по-умолчанию

- **start_routine** - **потоковая функция** - то, что начинает выполнять поток после создания
- **arg** - аргумент для потоковой функции - **потоковый аргумент**

  Возвращает **0** в случае успеха, значение ошибки в случае неудачи. В случае неудачи идентифкатор потока на который указывает **thread** остаются неопределенными.

- **Потоковая функция** имеет следующий вид

```
void* (*) (void*)
```

- По завершению функции **pthread_create()** (в случае успеха) родительский поток переходит к выполнению следующей инструкции, новый поток начинает выполнять потоковую функцию.
- Пример **thread-create.c**

```
#include <pthread.h>
#include <stdio.h>
void* thr_func(void* arg) {
        while(1) fputc('c',stdout);
        return NULL;
}
int main() {
        pthread_t thr_id;
        pthread_create(&thr_id, NULL, &thr_func,NULL);
        while(1) fputc('p',stdout);
        return 0;
}
```

Создает дочерний поток пишущий в **stdout**

```
$ cc -o thread-create thread-create.c -lpthread
```

Сборка с библиотекой **pthread**

```
$ ps x | grep thread-create
  31798 pts/4    Rl+    0:08 ./thread-create
  31804 pts/5    S+     0:00 grep thread-create
```

- **l** - многопоточный процесс

```
$ ps xm | grep -A 2 thread-create
  31798 pts/4    -      5:23 ./thread-create
      - -        Sl+    2:42 -
      - -        Sl+    2:40 -
--
  31960 pts/5    -      0:00 grep -A 2 thread-create
      - -        S+     0:00 -
```

Просмотр информации о потоках

2.3. Завершение потока

- Варианты
  - вызов функции **pthread_exit()**, ее параметр - статус возврата, может быть получен другим потоком процесса путем вызова **pthread_join()**

- окончание выполнения потоковой функции (возврат из нее), код возврата - аналогично предыдущему пункту
- выполнение потока отменяется из другого потока процесса вызовом **pthread_cancel()**
- любой поток в процессе вызывает **exit()** и завершает таким образом процесс со всеми потоками, то же самое, если главный поток делает возврат из **main()**

Естественно, потоки могут завершаться и до и после завершения главного потока.

- **pthread_join()** - аналог функции **wait()** для потока

```
#include <pthread.h>
int pthread_join(pthread_t thread, void **retval);
```

Возвращает **0** при успехе или код ошибки при неудаче

retval - если не NULL, туда будет записано значение, возвращенное потоком - возврат потоковой функции или аргумент **pthread_exit()**

```
void *
```

Тип возвращаемого значения (как и у потокового аргумента)

2.4. Передача данных потоку

- Производится через **потоковый аргумент**, а вернее через данные, например структуру, на которые этот потоковый аргумент указывает.

  Такая структура обычно делается собственная для каждой потоковой функции, что позволяет использовать потоковую функцию для порождения нескольких потоков, каждый со своими параметрами. Т.е. выполнять один и тот же код, но с разными данными.

- Пример **thread-create2.c**

```
#include <pthread.h>
#include <stdio.h>
struct thr_params {
        char c;
        int count;
};
void* thr_func(void* arg) {
        struct thr_params *p = (struct thr_params*)arg;
        for (int i=0; i < p->count; ++i)
                fputc(p->c,stdout);
        return NULL;
}
int main() {
        pthread_t thr1_id, thr2_id;
        struct thr_params thr1_params;
        thr1_params.count = 1000;
        thr1_params.c = 'a';
        pthread_create(&thr1_id, NULL, &thr_func, &thr1_params);
        struct thr_params thr2_params;
        thr2_params.count = 1500;
        thr2_params.c = 'b';
```

```
        pthread_create(&thr2_id, NULL, &thr_func, &thr2_params);
        pthread_join(thr1_id, NULL);
        pthread_join(thr2_id, NULL);
        return 0;
}
```

Необходимо убедиться, что данные, переданные потоку по ссылке не удалятся до тех пор, как поток не завершит работу с ними. Это касается как локальных переменных (стек), которые будут удаляться из памяти при завершении работы подпрограммы, так и динамическим переменным, которые могут быть освобождены средствами **free()**

- Пример **thread-create3.c** - возвращает значение из потоков

```
#include <pthread.h>
#include <stdio.h>
struct thr_params {
        char c;
        int count;
};
void* thr_func(void* arg) {
        struct thr_params *p = (struct thr_params*)arg;
        int i;
        for (i=0; i < p->count; ++i)
                fputc(p->c,stdout);
        return (void *)i;
}
int main() {
        pthread_t thr1_id, thr2_id;
        int thread_ret;
        struct thr_params thr1_params;
        thr1_params.count = 1000;
        thr1_params.c = 'a';
        pthread_create(&thr1_id, NULL, &thr_func, &thr1_params);
        struct thr_params thr2_params;
        thr2_params.count = 1500;
        thr2_params.c = 'b';
        pthread_create(&thr2_id, NULL, &thr_func, &thr2_params);
        pthread_join(thr1_id, (void *)&thread_ret);
        printf("Thread %lu retuns %d\n", thr1_id, thread_ret);
        pthread_join(thr2_id, (void *)&thread_ret);
        printf("Thread %lu retuns %d\n", thr2_id, thread_ret);
        return 0;
}
```

2.5. Определение текущего потока

- Иногда в подпрограмме надо уметь определять, какой поток выполняет ее в данный момент

```
#include <pthread.h>
pthread_t pthread_self(void);
```

Возвращает идентификатор текущего потока

```c
#include <pthread.h>
int pthread_equal(pthread_t t1, pthread_t t2);
```

Сравнение двух идентифкаторов

- Пример **thread-self.c** - убеждаемся, что не ждем сами себя (в этом случае **thread_join()** вернет ошибку)

```c
#include <pthread.h>
#include <stdio.h>
struct thr_params {
        char c;
        int count;
};
void* thr_func(void* arg) {
        struct thr_params *p = (struct thr_params*)arg;
        for (int i=0; i < p->count; ++i)
                fputc(p->c,stdout);
        return NULL;
}
int wait_thread(pthread_t thr_id) {
        if (!pthread_equal(pthread_self(), thr_id))
                pthread_join(thr_id, NULL);
}
int main() {
        pthread_t thr1_id, thr2_id;
        struct thr_params thr1_params;
        thr1_params.count = 1000;
        thr1_params.c = 'a';
        pthread_create(&thr1_id, NULL, &thr_func, &thr1_params);
        struct thr_params thr2_params;
        thr2_params.count = 1500;
        thr2_params.c = 'b';
        pthread_create(&thr2_id, NULL, &thr_func, &thr2_params);
        wait_thread(thr1_id);
        wait_thread(thr2_id);
        return 0;
}
```

2.6. Ожидаемые и отсоединенные потоки

- По умолчанию потоки создаются **ожидаемыми**, т.е. данные о нем хранятся в системе до вызова **pthread_join()**
- Для того, чтобы ресурсы освобождались сразу же поток можно делать **отсоединенным** - по завершению сразу уничтожается, другие потоки не могут вызвать для него **pthread_join()**

```c
#include <pthread.h>
int pthread_detach(pthread_t thread);
```

отсоединение потока

2.7. Отмена потока

- Возможность вызвать из одного потока уничтожение другого

```
#include <pthread.h>
int pthread_cancel(pthread_t thread);
```

Посылает запрос на отмену **другого** потока. При этом не ждет его завершения.

- После этой функции стоит дождаться выполнения потока с целью корректного освобождения ресурсов, только если поток не является отсоединенным.
- Режимы отмены потока
  - **асинхронно отменяемый** - можно выполнить отмену в любой точке выполнения
  - **синхронно отменяемый**(по-умолчанию) - можно отменить но только в определенных точках, запрос на отмену помещается в очередь и поток отменяется по достижению определенной точки
  - **неотменяемый** - запросы на отмену игнорируются

```
#include <pthread.h>
int pthread_setcanceltype(int type, int *oldtype);
```

Устанавливает тип отмены в значение **type**, возвращает старое значение типа в буфере, передаваемом указателем **oldtype**

Влияет только на текущий поток.

```
PTHREAD_CANCEL_DEFERRED
PTHREAD_CANCEL_ASYNCHRONOUS
```

- Установка точки отмены для синхронно отменяемого потока

```
#include <pthread.h>
void pthread_testcancel(void);
```

При вызове этой функции обрабатываются находящиеся в очереди запросы на отмену.

Для синхронно отменяемых потоков имеет смысл вставлять в код время-то времени.

- Пример **thread-cancel.c**

```
#include <pthread.h>
#include <stdio.h>
struct thr_params {
        char c;
        int count;
};
void* thr_func(void* arg) {
        struct thr_params *p = (struct thr_params*)arg;
        for (int i=0; i < p->count; ++i)
                fputc(p->c,stdout);
        printf("Before cancel\n");
        pthread_testcancel();
        printf("After cancel");
        return NULL;
}
```

```
int main() {
        pthread_t thr1_id, thr2_id;
        struct thr_params thr1_params;
        thr1_params.count = 1000;
        thr1_params.c = 'a';
        pthread_create(&thr1_id, NULL, &thr_func, &thr1_params);
        pthread_cancel(thr1_id);
        struct thr_params thr2_params;
        thr2_params.count = 1500;
        thr2_params.c = 'b';
        pthread_create(&thr2_id, NULL, &thr_func, &thr2_params);
        pthread_cancel(thr2_id);
        return 0;
}
```

2.8. Неотменяемые потоки и критические секции

- Поток можно сделать неотменяемым, вызвав функцию

```
#include <pthread.h>
int pthread_setcancelstate(int state, int *oldstate);
```

- **state** - состояние отменяемости

```
PTHREAD_CANCEL_ENABLE
PTHREAD_CANCEL_DISABLE
```

  Возврат в отменяемое состояние - повторный вызов функции с соответсвующим значением **state**

- **Критическая секция** - участок программы, который должен либо выполниться полностью, либо не выполниться вообще организуется при помощи перевод потока в неотменяемый режим

```
pthread_setcanselstate(PTHREAD_CANCEL_DISABLE, NULL);
// код критической секции
pthread_setcanselstate(PTHREAD_CANCEL_ENABLE, NULL);
```

2.9. Области потоковых данных

- Глобальные переменне у потоков общие
- Стек у каждого свой
- Иногда требуется продублировать определенную переменную, чтобы у каждого потока была ее собственная копия
- **Области потоковых данных** - дублируемые в каждый поток области памяти. Позволяют не заниматься закидыванием в стек поточных функций тех переменных, которые нужны каждому потоку свои.
- Все потоковые переменные (называются ключи) имеют тип

```
void *
```

- Создание нового ключа (новой потоковой переменной)

```
#include <pthread.h>
int pthread_key_create(pthread_key_t *key, void (*destructor)(void*));
```

- **key** - указатель на переменную, в которую будет записано значение ключа, посредством которого любой поток сможет обращаться к **своей** копии данных
- **destructor** - указатель на функцию очистки ключа, эта функция будет автоматически вызываться при уничтожении потока. Обычно используется при работе с динамической памятью.

  Этой функции передается значение ключа, потоковая копия которого должна быть уничтожена. Функция будет вызвана в том числе и в случае отмены потока в произвольной точке.

  Параметр устанавливается в значение NULL, Если функция очистки не нужна.
- После создания ключа, каждый поток может назначить ему собственное значение, вызывая функцию

```
#include <pthread.h>
int pthread_setspecific(pthread_key_t key, const void *value);
```

- Для того, чтобы получить значение ключа

```
#include <pthread.h>
void *pthread_getspecific(pthread_key_t key);
```

- Пример **thread-spec.c** - установка отдельных переменных для каждого потока

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

pthread_key_t thread_key;
FILE* logfile;

void write_to_log() {
        char* message =  (char *)pthread_getspecific(thread_key);
        fprintf(logfile, "%s\n", message);
}
void* thread_func(void *args) {
        char thread_var[30];
        sprintf(thread_var, "Thread ID: %lu\n",pthread_self());
        pthread_setspecific(thread_key, thread_var);
        write_to_log();
        return NULL;
}
int main() {
        char *log_file_name = "threads.log";
        logfile = fopen(log_file_name, "w");
        pthread_t threads[5];
        pthread_key_create(&thread_key, NULL);
```

```
        int i;
        for (i=0; i < 5; i++)
                pthread_create(&(threads[i]), NULL, thread_func, NULL);
        for (i=0; i < 5; i++)
                pthread_join(threads[i], NULL);
        fclose(logfile);
        return 0;
}
```

- Пример **thread-speclog.c** - отдельное имя файла на каждый поток

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

pthread_key_t log_file_key;

void write_to_log(const char *message) {
        FILE* log =  (FILE*)pthread_getspecific(log_file_key);
        fprintf(log, "%s\n", message);
}
void close_log(void* log) {
        fclose((FILE*)log);
}
void* thread_func(void *args) {
        char log_file_name[30];
        sprintf(log_file_name, "thread-%lu.log",pthread_self());
        FILE* log = fopen(log_file_name, "w");
        pthread_setspecific(log_file_key, log);
        write_to_log("Thread start\n");
        return NULL;
}
int main() {
        pthread_t threads[5];
        pthread_key_create(&log_file_key, close_log);
        int i;
        for (i=0; i < 5; ++i)
                pthread_create(&(threads[i]), NULL, thread_func, NULL);
        for (i=0; i < 5; ++i)
                pthread_join(threads[i], NULL);
        return 0;
}
```

2.10. Обработчики очистки

- Для ключей есть функции очистки ключей, которые гарантируют, что после завершении или отмены потока не произойдет потери ресурсов.
- Такие же функции часто требуются не только для ключей, но и для обычных ресурсов.
- Такие функции называются **обработчики очистки** или **обработчики выхода из потока**

- Таких функций для потока можно зарегистрировать несколько, зарегистрированные обработчики заносятся в стек и вызываются в порядке, обратном их порядку регистрации.

```
#include <pthread.h>
void pthread_cleanup_push(void (*routine)(void *), void *arg);
void pthread_cleanup_pop(int execute);
```

- **pthread_cleanup_push()** - регистрирует функцию **routine**, которая будет вызва- на с аргументом **arg**, когда поток выполнит одно из следующих действий
  - вызовет функцию **pthread_exit()** или вернет управление
  - ответит на запрос о принудительном завершении;
  - вызовет функцию **pthread_cleanup_pop()** с ненулевым аргументом **execute**.
- **pthread_cleanup_pop()** - удаляет зарегистрированный обработчик, в зависимости от значения параметра **execute**, либо выполняет его, либо нет.

```
void deallocate_buffer(void *buffer) {
    free(buffer);
}
void* thread_func(void *args) {
    void* buffer = malloc(2048);
    pthread_clenup_push(deallocate_buffer, buffer);
    pthread_cleanup_pop(1);
}
```

- **Примечание** аналогичная функция для процессов

```
#include <stdlib.h>
int atexit(void (*function)(void));
```