Глава 11

Потоки

{ XE "Поток:понятие" }*Потоки* (threads) позволяют одновременно выполнять некоторые действия в контексте одной программы. Механизмы работы с потоками реализованы в Linux в отдельной библиотеке Pthread.

Потоки в Linux — довольно сложная тема, заслуживающая написания отдельной книги. В этой главе описываются лишь основы работы с потоками. Дополнительную информацию по этой теме можно получить из источников, перечисленных в последнем разделе этой главы.

В рамках этой главы будут рассмотрены следующие темы:

- Понятие потоков и их особенности.
- Создание и завершение потока.
- Синхронизация потоков.
- Получение информации о потоках.
- Обмен данными между потоками.

11.1. Концепция потоков в Linux

Любой процесс — это выполняющийся код плюс данные, которыми он манипулирует. Этот закон "вступил в силу" с момента изобретения первого компьютера и актуален по сей день.

Вернемся к Linux. Когда один процесс вызывает `fork()`, в системе появляется другой процесс, который выполняет тот же код. Но данные, которыми манипулирует этот код, являются независимой копией данных процесса-родителя.

Чтобы продемонстрировать это, рассмотрим небольшой пример (листинг 11.1).

Листинг 11.1. Пример pdata.c

```
#include <stdio.h>
#include <unistd.h>

int main (void)
{
       int a = 10;
       pid_t status = fork ();

       /* Child */
       if (!status) {
              a++;
              printf ("Child's a=%d\n", a);
              return 0;
       }

       /* Parent */
       wait ();
       printf ("Parent's a=%d\n", a);

       return 0;
}
```

Пример показывает, что переменная a в дочернем процессе не имеет ничего общего (кроме имени) с переменной a в родительском процессе.

> ***Примечание***
>
> Если в приведенном примере (листинг 11.1) вместо значений переменной a вы захотите посмотреть адреса, то будете удивлены, что они численно совпадают. Это происходит из-за того, что каждый процесс в Linux использует собственное адресное пространство. Поскольку сразу после вызова `fork()` процессы практически идентичны, то локальные адреса переменной a совпадают.

Процессы могут иметь общие данные, но для этого программисты обращаются к методам *межпроцессного* **{** XE "Межпроцессное взаимодействие:понятие" **}***взаимодействия* (interprocess communication), которые будут рассматриваться далее в этой книге. Отметим лишь, что

межпроцессное взаимодействие подчас требует сложных манипуляций со стороны программиста. Кроме того, в некоторых случаях плохо реализованное межпроцессное взаимодействие может стать мишенью для злоумышленников.

Потоки позволяют в рамках одной программы выполнять одновременно несколько действий, используя при этом общие данные. В Linux потоки выполняются так же, как и процессы, т. е. независимо.

Потоки в Linux реализуются очень просто:

1. Создается функция, которая называется *потоковой функцией*{ XE "Функция:потоковая" }.
2. При помощи функции { XE "Функция:pthread_create()" }`pthread_create()` создается поток, в котором начинает параллельно остальной программе выполняться потоковая функция.
3. Вызывающая сторона продолжает выполнять какие-то действия, не дожидаясь завершения потоковой функции.

В самом начале этой главы сообщалось, что потоки реализованы в библиотеке pthread. Чтобы подключить эту библиотеку к программе, нужно передать компоновщику опцию `-lpthread`.

11.2. Создание потока: *pthread_create()*

Потоки подобно процессам работают с идентификаторами. Только эти идентификаторы существуют локально в рамках текущего процесса. Для их хранения предусмотрен специальный тип `pthread_t`, который становится доступным при включении в программу заголовочного файла { XE "Заголовочный файл:pthread.h" }pthread.h.

Для создания потока и запуска потоковой функции используется функция `pthread_create()`, объявленная в заголовочном файле pthread.h следующим образом:

```
int pthread_create (pthread_t * THREAD_ID, void * ATTR,
              void *(*THREAD_FUNC) (void*), void * ARG);
```

Итак, `pthread_create()` принимает четыре аргумента:

- По адресу в `THREAD_ID` помещается идентификатор нового потока (если таковой был создан).
- Безтиповый указатель `ATTR` служит для указания *атрибутов потока*{ XE "Поток:атрибуты" }. Если этот аргумент равен `NULL`, то поток создается с атрибутами по умолчанию. В рамках данной книги мы не будем передавать потокам специальные атрибуты.

- Пугающий своим видом аргумент PTHREAD_FUNC является указателем на потоковую функцию. Это обычная функция, возвращающая бестиповый указатель (void*) и принимающая бестиповый указатель в качестве единственного аргумента.
- Аргумент ARG — это бестиповый указатель, содержащий аргументы потока. Если потоковая функция не требует наличия аргументов, то в качестве ARG можно указать NULL.

Чтобы лучше понять концепцию потоков в Linux, нужно представить себе, что программа при запуске создает один поток, а функция pthread_create() позволяет создавать дополнительные потоки. Это означает, что основную программу следует трактовать как "родительский поток".

Следует также понимать, что если один из потоков завершает программу, то все остальные потоки тут же завершаются. Это еще одно важное отличие потоков от процессов.

Рассмотрим теперь небольшой пример (листинг 11.2).

Листинг 11.2. Пример nothread.c

```
#include <stdio.h>
#include <unistd.h>

void * any_func (void * args)
{
       fprintf (stderr, "Hello World\n");
       sleep (5);
       return NULL;
}

int main (void)
{
       any_func (NULL);
       fprintf (stderr, "Goodbye World\n");
       while (1);

       return 0;
}
```

Эта программа выводит сообщение "Hello World", ждет 5 с, выводит второе сообщение "Goodbye World" и уходит в бесконечный цикл. Чтобы принудительно завершить программу, нажмите комбинацию клавиш <Ctrl>+<C>. Итак, пятисекундная пауза между сообщениями обусловлена тем, что программа ждет завершения функции `any_func()`.

Давайте теперь модернизируем программу таким образом, чтобы `any_func()` была потоковой функцией (листинг 11.3).

Листинг 11.3. Программа thread1.c

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

void * any_func (void * args)
{
       fprintf (stderr, "Hello World\n");
       sleep (5);
       return NULL;
}

int main (void)
{
       pthread_t thread;
       int result;

       result = pthread_create
              (&thread, NULL, &any_func, NULL);

       if (result != 0) {
              fprintf (stderr, "Error\n");
              return 1;
       }

       fprintf (stderr, "Goodbye World\n");
       while (1);

       return 0;
}
```

Для удобства добавим также make-файл (листинг 11.4).

Листинг 11.4. Файл Makefile

```
thread1: thread1.c
	gcc -o $@ $^ -lpthread

clean:
	rm -f thread1
```

Теперь функции `main()` и `any_func()` работают параллельно, поэтому никакой видимой задержки между выводом двух сообщений не происходит. Бесконечный цикл в этой программе выполняет особую роль. Мы уже говорили, что при завершении программы одним из потоков все остальные потоки немедленно завершаются. Если бы в нашей программе не было бесконечного цикла, то возврат из функции `main()` мог бы произойти раньше, чем вывод сообщения "Hello World".

Для передачи данных в поток используется четвертый аргумент функции `pthread_create()`. Этот указатель автоматически становится аргументом потоковой функции. Следующий пример демонстрирует эту возможность (листинг 11.5).

Листинг 11.5. Пример threadarg.c

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

void * any_func (void * arg)
{
	int a = *(int*) arg;
	fprintf (stderr, "Hello World "
		"with argument=%d\n", a);
	return NULL;
}

int main (int argc, char ** argv)
{
	pthread_t thread;
	int arg, result;

	if (argc < 2) {
```

```
            fprintf (stderr, "Too few arguments\n");
            return 1;
        }

        arg = atoi (argv[1]);
        result = pthread_create (&thread, NULL,
                    &any_func, &arg);

        fprintf (stderr, "Goodbye World\n");
        while (1);

        return 0;
}
```

В приведенном примере первый аргумент программы (`argv[1]`) преобразуется в целое число, которое затем передается в потоковую функцию. Если требуется передать в поток несколько аргументов, то их можно разместить в структуре, указатель на которую также передается в потоковую функцию. Этот подход продемонстрирован в следующем примере (листинг 11.6).

Листинг 11.6. Пример threadstruct.c

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

struct thread_arg
{
        char * str;
        int num;
};

void * any_func (void * arg)
{
        struct thread_arg targ =
            *(struct thread_arg *) arg;
        fprintf (stderr, "str=%s\n", targ.str);
        fprintf (stderr, "num=%d\n", targ.num);
        return NULL;
}
```

```
int main (void)
{
       pthread_t thread;
       int result;
       struct thread_arg targ;
       targ.str = "Hello World";
       targ.num = 2007;

       result = pthread_create (&thread, NULL,
                     &any_func, &targ);

       while (1);
       return 0;
}
```

В этом приведенном примере потоковую функцию `any_func()` можно было бы реализовать следующим образом:

```
void * any_func (void * arg)
{
       struct thread_arg * targ =
              (struct thread_arg *) arg;
       fprintf (stderr, "str=%s\n", targ->str);
       fprintf (stderr, "num=%d\n", targ->num);
       return NULL;
}
```

Один процесс может создать сколько угодно потоков (в пределах разумного). Следующая программа (листинг 11.7) демонстрирует создание двух независимо работающих потоков.

Листинг 11.7. Программа multithread.c

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

void * thread_func1 (void * arg)
{
       fprintf (stderr, "thread1: %s\n", (char*) arg);
```

```
        sleep (5);
        return NULL;
}

void * thread_func2 (void * arg)
{
        fprintf (stderr, "thread2: %s\n", (char*) arg);
        sleep (5);
        return NULL;
}

int main (void)
{
        pthread_t thread1, thread2;
        char * thread1_str = "Thread1";
        char * thread2_str = "Thread2";

        if (pthread_create (&thread1, NULL,
                        &thread_func1, thread1_str) != 0) {
                fprintf (stderr, "Error (thread1)\n");
                return 1;
        }

        if (pthread_create (&thread2, NULL,
                        &thread_func2, thread2_str) != 0) {
                fprintf (stderr, "Error (thread2)\n");
                return 1;
        }

        fprintf (stderr, "Hello World\n");
        while (1);
        return 0;
}
```

11.3. Завершение потока: *pthread_exit()*

Известно, что программа обычно завершается посредством возврата из функции `main()` или через вызов `exit()`. Аналогичным образом работают и потоки, которые могут завершаться возвратом из потоковой функции или вызовом