8. POSIX НИШКИ

Ще разгледаме нишките по стандарта POSIX 1003.1c, известни още като pthreads, които могат да се използват в съвременните версии на Unix и Linux системите. В някои версии, като UNIX System V, Solaris, Linux и др., нишките се реализират в ядрото. В други версии, като BSD, нишките не се поддържат от ядрото и се реализирата в потребителското пространство в една от библиотеките. Затова ще използваме термина функция, а не системен примитив. Функциите за работа с нишки, които ще разгледаме, осигуряват:

- Основни операции с нишки, като създаване, завършване и др.
- Механизми за синхронизация на работата на конкурентните нишки в един процес.

8.1. Основни операции с нишки

Първата - главна нишка във всеки процес се създава автоматично при създаване на процес. Друга нишка може да се създаде, когато една нишка изпълни функцията `pthread_create`.

```
#include <pthread.h>
int pthread_create(pthread_t *thread, pthread_attr_t *attr,
                   void *(*start_routine)(void *), void *arg);
                                        Връща 0 при успех, число ≠ 0 при грешка.
```

Всяка нишка има идентификатор, който й се присвоява при създаването и е от тип `pthread_t`. При успех се връща идентификаторът на новата нишка чрез аргумента *`thread`*.

Аргументът *`attr`* определя някои характеристики на създаваната нишка или както ги наричат атрибути. Ако е зададено значение `NULL`, то атрибутите имат значения по премълчаване. Един от атрибутите определя типа на нишката: joinable или detached. Тип joinable означава, че друга нишка в процеса може да се синхронизира с момента на завършването й, т.е. след завършване на нишката тя не изчезва веднага (подобно на състояние зомби при процеси). Тип detached означава, че при завършване на нишката веднага се освобождават всички ресурси, заемани от нея, т.е. тя изчезва в момента на завършване. По премълчаване нишката се създава от тип joinable. Има и други атрибути, които определят дисциплината и параметри на планиране на нишки.

Когато нишката бъде създадена тя започва да изпълнява функцията *`start_routine`*, на която се предава аргумент *`arg`*. За разлика от процесите, където бащата и синът продължават изпълнението си след `fork` от една и съща точка, тук не е така, защото нишките на един процес имат общи ресурси. При успех функцията връща 0, а при грешка връща различен от 0 код на грешка.

Така създадената нишка завършва когато:

- Изпълни `return` от *`start_routine`*;
- Извика явно `pthread_exit`;
- Друга нишка я прекрати чрез `pthread_cancel`.

```
#include <pthread.h>
void pthread_exit(void *retval);
```

Аргументът *`retval`* е код на завършване, който нишката изработва. Той е предназначен за всяка друга нишка на процеса, която изпълни `pthread_join`. Но ако нишката е от тип detached, то след `pthread_exit` от нея не остава никаква следа, следователно и кодът не се съхранява. От тази функция няма връщане.

Чрез функцията `pthread_join` една нишка може да синхронизира изпълнението си със завършването на друга нишка (аналог е на примитива `wait` при процеси, но тук няма изискването текущата нишка да е баща на чаканата).

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

Връща 0 при успех, число ≠ 0 при грешка.

Аргументът *thread* е идентификатор на нишката, чието завършване се чака от текущата нишка. Текущата нишка се блокира докато не завърши нишка *thread*. Най-много една нишка може да изчака завършването на коя да е друга нишка, т.е. ако няколко нишки изпълнят `pthread_join` за една и съща нишка, вторият `pthread_join` ще върне грешка. При успех функцията връща 0 и чрез аргумента *value_ptr* се предава кодът на завършване на нишката *thread*. При грешка връща код на грешка различен от 0.

Пример

Програма 8.1 илюстрира състезание между нишки. Двете нишки изпълняват една и съща функция, на която се предават аргументи чрез структура.

```
/* ------------------------------------------------------------ */
/*  Threads - Race condition with thteads */

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

/* Parameters to thread function */
struct pr_parms
{
  char ch;     /* The character to print */
  int count;   /* The number of times to print it */
};

int main (void)
{
  pthread_t th1;
  pthread_t th2;
  struct pr_parms th1_args;
  struct pr_parms th2_args;
  void *ch_print();

/* Create a thread to print 3000 x's */
  th1_args.ch = 'x';
  th1_args.count = 3000;
  pthread_create(&th1, NULL, ch_print, &th1_args);

/* Create a thread to print 2000 o's */
  th2_args.ch = 'o';
  th2_args.count = 2000;
  pthread_create(&th2, NULL, ch_print, &th2_args);

/* Wait the first thread to finish */
  pthread_join(th1, NULL);
/* Wait the second thread to finish */
  pthread_join(th2, NULL);

  exit(0);
}
```

```
/* Thread function */
void *ch_print (void *param)
{
  struct pr_parms* p = (struct pr_parms*)param;
  int i, j;
  for ( i =0; i < p->count; ++i){
    fputc (p->ch, stdout);
    for(j=0; j<10000; j++);  /* for some delay */
  }
  pthread_exit((void *)0);
}
/* ---------------------------------------------------------- */
```

При компилиране на програма с нишки трябва да включим библиотеката `pthread`:

```
$ cc pthread_print.c -lpthread
```

Като изпълним програмата на стандартния изход получаваме изход, в който се редуват последователности от символа ‘o’ и последователности от символа ‘x’ с различни дължини.

8.2. Механизъм mutex

Mutex е механизъм за блокиране и деблокиране на нишки, чрез който може да се реализира взаимно изключване на нишки при достъп до общи променливи. Един обект mutex има две състояния:

- unlocked – свободен и не принадлежи на никоя нишка
- locked – заключен и принадлежи на нишката, която го е заключила.

Ако обект mutex е в състояние locked, той може да принадлежи само на една нишка. Следва описание на функциите за работа с обект mutex.

```
#include <pthread.h>
int pthread_mutex_init(pthread_mutex_t *mutex,
                       pthread_mutexattr_t *mutexattr);
int pthread_mutex_lock(pthread_mutex_t *mutex);
int pthread_mutex_trylock(pthread_mutex_t *mutex);
int pthread_mutex_unlock(pthread_mutex_t *mutex);
int pthread_mutex_destroy(pthread_mutex_t *mutex);
```

Връщат 0 при успех, число ≠ при грешка.

Нов обект mutex се създава чрез функцията `pthread_mutex_init`. Идентификатор на новосъздадения обект mutex е променлива от тип `pthread_mutex_t` и се връща чрез аргумента *`mutex`*. Първоначално обектът mutex е в състояние unlocked. Вторият аргумент *`mutexattr`* задава атрибутите на създавания mutex. В Linux се реализира атрибут тип на mutex, който може да е: fast, recursive или error checking. Типът влияе на семантиката на последващите операции над обекта mutex. По премълчаване, ако аргументът е `NULL`, се създава mutex от тип fast. При успех функцията връща 0, а при грешка връща различен от 0 код на грешка.

Mutex е общ ресурс, т.е. ресурс на процеса, а не на нишката, която го създава. Това означава, че не се унищожава или освобождава при завършване на нишката, която го е създала.

Чрез функцията `pthread_mutex_lock` текущата нишка се опитва да заключи обекта *`mutex`*. Възможните случаи при изпълнението са следните.

1. Ако *mutex* е свободен, то състоянието му се сменя в заключен от текущата нишка и тя продължава изпълнението си.

2. Ако *mutex* е заключен от някоя друга нишка, то текущата нишка бива блокирана, докато се смени състоянието му в unlocked от нишката, която го притежава в момента.

3. Ако *mutex* е заключен от текущата нишка, действието зависи от типа на *mutex*.

- При тип fast текущата нишка се блокира (това може да доведе до дедлок).
- При тип recursive се увеличава брояч (броят се многократните заключвания на един обект mutex от една нишка) и функцията завършва успешно.
- При тип error checking това се счита за грешка.

С функцията `pthread_mutex_unlock` текущата нишка се опитва да освободи обекта *mutex*. Възможните случаи при изпълнението са:

1. Ако текущата нишка е настоящия притежател на обекта *mutex*, то се сменя състоянието му в unlocked. Ако има нишки, чакащи този обект *mutex* (блокирани в `pthread_mutex_lock`), то една от тях се събужда и й се дава възможност да се опита отново да получи *mutex*. Но ако *mutex* е от тип recursive се намалява броячът и когато той стане 0, тогава се прави отключване и събуждане.

2. Ако *mutex* е в състояние unlocked или е locked но от друга нишка, то действието зависи от типа на *mutex*:

- При тип error checking това е грешка.
- При тип fast и recursive не се прави проверка.

При успех функциите връщат 0, а при грешка връща различен от 0 код на грешка.

Функцията `pthread_mutex_destroy` унищожава обекта *mutex*, който трябва да е в състояние отключен иначе се счита за грешка. При успех функцията връща 0, а при грешка връща различен от 0 код на грешка.

Пример

Програма 8.2 илюстрира взаимно изключване при нишки чрез mutex.

```
/* ------------------------------------------------------------ */
/* Threads - Mutual Exclusion with mutex */

#include <pthread.h>
#include "ourhdr.h"

pthread_mutex_t mut;
int sum;

main(void)
{
  int ret, status, cnt1, cnt2;
  pthread_t th1, th2;
  void *race_func();

  cnt1 = 1;
  cnt2 = 2;
  sum = 0;

  ret = pthread_mutex_init(&mut, NULL);
  if ( ret != 0 ) err_exit("Error in pthread_mutex_init");

  ret = pthread_create(&th1, NULL, race_func, &cnt1);
  if ( ret != 0 ) err_exit("Error in pthread_create 1");

  ret = pthread_create(&th2, NULL, race_func, &cnt2);
  if ( ret != 0 ) err_exit("Error in pthread_create 2");
```

```
  pthread_join(th1, (void *)&status);
//  printf("Thread 1 exit status: %d\n", status);

  pthread_join(th2, (void *)&status);
//  printf("Thread 2 exit status: %d\n", status);

  printf("\nSUM = %d\n", sum);
  exit(0);
}

/* Thread function */
void *race_func(void *ptr)
{
  int cnt, i, j;
  cnt = *(int *)ptr;
  printf("Thread %d\n", cnt);
  for (i=1; i <= 100; i++) {
       pthread_mutex_lock(&mut);
       sum = sum + cnt;
       printf("%d\n", sum);
       pthread_mutex_unlock(&mut);
       for (j=0; j<400000; j++);    /*for some delay */
  }
  pthread_exit((void *)0);
}
/* ------------------------------------------------------------ */
```

Изпълнихме програмата и получихме следния изход.

```
$ a.out
Thread 1
1
2
3
Thread 2
5
7
8
9
10
11
12
14
16
18
```

<още много редове, в които се вижда как се редуват двете нишки>

```
300

SUM = 300
```

8.3. Механизъм condition

Condition е съкращение от condition variables или условни променливи. Това е друг механизъм за синхронизация, чрез който могат да се реализират по-сложни условия за блокиране и деблокиране на нишки. С всеки обект condition е свързано условие, при което нишката може да работи (или обратно, условие при което нишката ще се блокира). Това условие може да е произволно сложно. В следващия пример ще разгледаме едно много просто условие: ако променливата `flag` е 0 нишката ще се блокира, а ако `flag` е 1 ще работи.

Една нишка може да изпълнява операция *wait* върху обект condition, при което нишката се блокира докато друга нишка изпълни операция *signal* над същия обект condition. Когато една нишка изпълнява операция *signal*, то ако има нишки блокирани по обекта condition, се събужда една, в противен случай нищо не се прави. Операциите *wait* и *signal* са подобни на операциите P и V при семафори, но има една съществена разлика. Семафорът има брояч, който помни събуждането ако няма кого да буди, докато тук ако няма блокирани нишки, събуждането не се помни.

Как трябва да се използва обект condition за реализиране на условна синхронизация? Следва описание на схемата на работа в случай на две нишки.

- В нишката, която трябва да чака условието за да работи:
  1. Проверява условието.
  2. Ако условието не е изпълнено, извиква операцията *wait*.
- В нишката, която променя и сигнализира условието:
  1. Променя променливите, от които зависи истинността на условието.
  2. Извиква операцията *signal.*

Всяка от двете нишки изпълнява две стъпки (проверка или изменение на условието и операция над обекта condition). За да няма състезание между тях е необходимо да се осигури атомарност на двете стъпки. Затова всеки обект condition се използва заедно с един обект mutex. Този mutex трябва да се заключва явно преди първата стъпка и да се отключва след втората, във всяка от двете нишки. Освен това той автоматично се отключва и заключва от операцията *wait*.

Следва описание на функциите за работа с обект condition.

```
#include <pthread.h>
int pthread_cond_init(pthread_cond_t *cond,
                      pthread_condattr_t *condattr);
int pthread_cond_wait(pthread_cond_t *cond, pthread_mutex_t *mutex);
int pthread_cond_signal(pthread_cond_t *cond);
int pthread_cond_broadcast(pthread_cond_t *cond);
int pthread_cond_destroy(pthread_cond_t *cond);
                                    Връщат 0 при успех, число ≠ при грешка.
```

Нов обект се създава чрез функцията `pthread_cond_init`. Идентификатор на новосъздадения обект е променлива от тип `pthread_cond_t` и се връща чрез аргумента *`cond`*. Вторият аргумент *`condattr`* задава атрибутите на създавания обект condition. Ако в него е зададено значение `NULL`, атрибутите имат значения по премълчаване. Реализацията в Linux не поддържа никакви атрибути, така че вторият аргумент се игнорира.

Чрез функцията `pthread_cond_signal` може да се събуди една нишка. Ако има нишки, блокирани в `pthread_cond_wait` по *`cond`*, се събужда една от тях. Ако няма блокирани нишки, нищо не се прави. Функцията `pthread_cond_broadcast` събужда всички нишки, блокирани по *`cond`*.

Нишка се блокира чрез функцията `pthread_cond_wait`. Текущата нишка се блокира по *`cond`*, докато друга нишка я събуди. Когато се извиква тази функция *`mutex`* трябва вече да е заключен от текущата нишка. Затова тази функция автоматично го отключва и блокира нишката. Когато по-късно бъде събудена от друга нишка, изпълняваща `pthread_cond_signal`, то *`mutex`* се заключва отново автоматично. След това нишката трябва отново да се провери условието, свързано с *`cond`* (първата стъпка от горната схема).

Функцията `pthread_cond_destroy` унищожава обекта *`cond`*, ако няма нишки блокирани по него.

При успех функциите връщат 0, а при грешка връщат различен от 0 код на грешка. (В Linux функциите `pthread_cond_init`, `pthread_cond_wait`, `pthread_cond_signal` и `pthread_cond_broadcast` винаги завършват успешно.)

И така схемата на работа в двете нишки е следната.

- В нишката, която трябва да чака условието за да работи:

```
pthread_mutex_lock(&mut);
while( ! условие_за_да_работи )
   pthread_cond_wait(&cv, & mut);
pthread_mutex_unlock(&mut);
```

- В нишката, която променя и сигнализира условието:

```
pthread_mutex_lock(&mut);
изменя променливите, определящи истинността на условието;
pthread_cond_signal(&cv);
pthread_mutex_unlock(&mut);
```

Пример

Програма 8.3 илюстрира работа с обект condition, като реализира синхронизация на две нишки. Условието е много просто: ако променливата `flag` е 0 нишката, в която работи функцията `th_wait`, ще се блокира, а ако `flag` е 1 ще работи. Другата нишка, в която работи функцията `th_sign`, изменя условието, т.е. изменя променливата `flag`.

```
/* ------------------------------------------------------------ */
/*  Threads - Condition Variable */

#include <pthread.h>
#include "ourhdr.h"

pthread_mutex_t flag_mut;
pthread_cond_t flag_cv;
int flag;

main(int argc, char *argv[])
{
  int ret;
  pthread_t th1, th2;
  void *th_sign();
  void *th_wait();
  char *mess1 = "Hello";
  char *mess2 = "World";

  if ( argc >= 3) {
```

```
      mess1 = argv[1];
      mess2 = argv[2]; }

/* Initialize the mutex and condition variables */
  ret = pthread_mutex_init(&flag_mut, NULL);
  if ( ret!=0 ) err_exit("Error in pthread_mutex_init");

  pthread_cond_init(&flag_cv, NULL);

  flag = 0;

/* Create the threads */
  ret = pthread_create(&th1, NULL, th_wait, mess2);
  if ( ret != 0 ) err_exit("Error in pthread_create 1");

  ret = pthread_create(&th2, NULL, th_sign, mess1);
  if ( ret != 0 ) err_exit("Error in pthread_create 2");

/* Wait the threads to finish */
  pthread_join(th1, NULL);
  pthread_join(th2, NULL);

  exit(0);
}

/* Signal Thread function */
void *th_sign(void *ptr)
{
  char *msg;
  msg = (char *)ptr;
  printf("Signal thread\n");
  printf("%s\n", msg);

/* Signal the condition: flag is set */
  pthread_mutex_lock(&flag_mut);
  flag = 1;
  pthread_cond_signal(&flag_cv);
  pthread_mutex_unlock(&flag_mut);

  pthread_exit((void *)0);
}

/* Wait Thread function */
void *th_wait(void *ptr)
{
  char *msg;
  msg = (char *)ptr;
  printf("Wait thread\n");

/* Wait the condition: flag is set */
  pthread_mutex_lock(&flag_mut);
  while( !flag )
       pthread_cond_wait(&flag_cv, &flag_mut);
  pthread_mutex_unlock(&flag_mut);

/* Do the work */
  printf("%s\n", msg);
  pthread_exit((void *)0);
}
/* ------------------------------------------------------------ */
```

Изпълнихме програмата и получихме следния изход.

```
$ a.out
Wait thread
Signal thread
Hello
World
$ a.out first second
Wait thread
Signal thread
first
second
```

Реализацията на POSIX нишките в различните версии на Unix и Linux се различава. В някои версии нишките са реализирани като процеси, т.е. когато извикваме `pthread_create` за да създадем нова нишка, се създава нов процес с нов pid, който да изпълнява нишката. Но този процес се различава от процес създаден чрез `fork`. Процесите за двете нишки (оригиналната и новосъздадената) имат общо адресно пространство и други ресурси. В по-новите версии на Linux нишките в един процес имат еднакъв pid.

Пример

Програма 8.4 ще ни демонстрира дали нишката е реализирана като процес или не. Главната нишка създава друга нишка, двете нишки извикват `getpid` и извеждат на стандартния изход значението на pid си, след което изпълняват безкраен цикъл.

```
/* ------------------------------------------------------------ */
/*  Threads - Test for pid of thread */

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

main(void)
{
  pthread_t th1;
  void *print_pid();

  printf("Main pid=%d\n", getpid());
  pthread_create(&th1, NULL, print_pid, NULL);
  while(1);

  pthread_join(th1, NULL);
  exit(0);
}

/* Thread function */
void *print_pid()
{
  printf("Thread pid=%d\n", getpid());
  while(1);

  pthread_exit((void *)0);
}
/* ------------------------------------------------------------ */
```

Изпълняваме програмата във фонов режим и чрез командата `ps` проверяваме процесите. В Red Hat Linux 5.2 (2.0.36) получихме следния резултат.

```
$ a.out &
Main pid=481
Thread pid=483
$ ps
```

```
  PID TTY STAT TIME COMMAND
  424   2 S    0:00 -bash
  481   2 R    0:22 a.out
  482   2 S    0:00 a.out
  483   2 R    0:22 a.out
  484   2 R    0:00 ps
```

Виждаме, че има три процеса за програмата, въпреки че нишките са две. Първият процес, с pid 481, е за главната нишка. Третият процес, с pid 483, е за създадената с `pthread_create` нишка. Процесът с pid 482 е така наречения "manager thread". Той се създава при първото извикване на `pthread_create` и е част от реализацията на нишките в Linux. (Не трябва да забравим да прекратим процесите с команда `kill 481`.)

В Scientific Linux (2.6.9) получихме друг резултат.

```
$ a.out &
Main pid=30680
Thread pid=30680
$ ps
  PID TTY          TIME CMD
29128 pts/1    00:00:01 -bash
30680 pts/1    00:01:00 a.out
30690 pts/1    00:00:00 ps
```

Виждаме, че двете нишки имат еднакъв pid 30680 и командата `ps` показва един процес за програмата.