# 7 VLÁKNA

## 7.1 Čo je vlákno

Vlákna umožňujú procesom vykonávať viac činností súčasne. Vlákno existuje v rámci procesu a vykonáva určitú časť jeho kódu. Jeden proces môže obsahovať viac vlákien. Vlákna sa podobajú procesom, ale na rozdiel od nich zdieľajú jeden adresný priestor (dáta a kód), pričom každé vlákno má svoj zásobník a svoj kontext.

Hlavný prínos pri použití vlákien je možnosť procesu prelínať rôzne svoje činností a využiť tak pridelený čas procesora efektívnejšie. Napríklad ak proces obsluhuje nejaké prichádzajúce požiadavky zo siete, jedno vlákno môže tie požiadavky preberať, druhé ich môže spracovávať a tretie ich ukladať na disk alebo ináč spracovávať. Tento spôsob práce umožňuje procesu využívať súčasne viacero prostriedkov systému a tak dosiahnuť väčšiu mieru paralelizmu vo svojej práci. Návrh programu, ktorý využíva vlákna musí byť starostlivo premyslený, pretože samotná prítomnosť vlákien nie je zárukou urýchlenia programu. Práve súčasné využitie rôznych prostriedkov systému (procesor, pamäť, periférne zariadenia) dovolí urýchlenie programu s vláknami oproti procesu, ktorý vykonáva to isté bez vlákien.

Ďalšia výhoda pri použití vlákien je ich jednoduchá komunikácia. Tým, že zdieľajú spoločný adresný priestor, réžia na komunikáciu je minimálna.

Prepínanie kontextu medzi vláknami má menšiu réžiu oproti prepínaniu kontextu procesov, pretože v prípade vlákien sa nevymieňa adresný priestor.

Vlákna majú aj nevýhody, hlavne to, že dobrý program využívajúci vlákna sa ťažšie navrhuje a jeho odladenie je zložitejšie. Jednoduchšia komunikácia medzi vláknami neodstraňuje potrebu synchronizácie pri prístupe k spoločným dátam.

Implementácia vlákien sa môže uskutočniť buď **pomocou knižnice** (rozšírenejší spôsob) alebo **priamo v jadre operačného systému**. V prvom prípade operačný systém plánuje vykonávanie procesu a proces delí pridelený čas medzi svojimi vláknami. V tomto prípade základná jednotka plánovania je proces, operačný systém „nevidí“ vlákno.

V prípade implementácie v jadre, operačný systém plánuje priamo vlákna a proces je len účtovacia jednotka, t. j. procesu sa prideľujú prostriedky a jemu sa ráta ich využitie. Skutočné paralelné vykonávanie vlákien je možné len v prípade viacprocesorového systému.

GNU/Linux implementuje vlákna unikátnym spôsobom. V jadre Linuxu neexistuje koncepcia vlákna! Vlákna sú implementované ako štandardné procesy a Linux neposkytuje špeciálne plánovanie alebo dátové štruktúry pre vlákna. V tomto ponímaní vlákno je proces, ktorý zdieľa určité prostriedky s inými procesmi (to sú jeho „príbuzné“ vlákna). Táto implementácia sa líši od implementáciácií v iných systémoch ako sú Windows alebo Solaris.

GNU/Linux implementuje vlákna štandardu POSIX pomocou knižnice, známej pod názvom `pthread`. Všetky funkcie pre prácu s vláknami a typy dát sa nachádzajú v hlavičkovom súbore `<pthread.h>`. Názvy všetkých funkcií z tejto knižnice začínajú predponou `pthread_`.

Preklad aplikácie, ktorá využíva vlákna z knižnice `pthread` sa vykoná príkazom:

```
cc -o meno -lpthread meno.c
```

Informácie o vláknach v procese je možné získať pomocou modifikátorov príkazu `ps` (pozri manuálové stránky Linuxu). Napríklad:

```
$ ps -eLf
UID      PID  PPID   LWP  C NLWP STIME TTY       TIME CMD
marti  21518 21512 21518  0    1 09:41 ?     00:00:00 sshd:

marti@pts/0
marti  21519 21518 21519  0    1 09:41 pts/0 00:00:00 -bash
marti  22514 21519 22514  0    1 13:19 pts/0 00:00:00 bash
marti  22555 22514 22555  0    1 13:23 pts/0 00:00:00 ps –eLf
```

Vo výpise stĺpec „LWP“ označuje tzv. „odľahčený proces“ ((angl. *lightweight process*). To je proces, pomocou ktorého sa vykonáva vlákno. NLWP je počet vlákien v procese. Ostatné stĺpce majú identický význam ako pri procesoch.

## 7.2 Vytvorenie vlákna

Funkcia, ktorá vytvorí vlákno v rámci procesu sa nazýva `pthread_create()`.

```
#include <pthread.h>

int pthread_create(pthread_t * thread, pthread_attr_t
        *attr, void * (*start_routine)(void *), void * arg);
```

Funkcia vytvorí nové vlákno, ktoré sa vykonáva súčasne s volajúcim vláknom. Argumenty volania sú:

- `thread` - ukazovateľ na premennú, kde sa uloží identifikátor vlákna po jeho vytvorení.
- `attr` - atribúty nového vlákna, ak sa zadá NULL, vlákno sa vytvorí so štandardnými atribútmi, t. j. vlákno nie je „detached“ a plánovanie je štandardné (nie real time)
- `start_routine` - ukazovateľ na funkciu, ktorú vlákno vykonáva,
- `arg` - argument, ktorý sa odovzdáva vláknu ( môže byť aj NULL ). Keďže jeho typ je `void`, odovzdá sa akýkoľvek argument.

Návrat z volania z `pthread_create()` je okamžitý a volajúce vlákno pokračuje príkazom nasledujúcom po volaní. Nové vlákno sa tiež začína vykonávať, pričom Linux plánuje obe vlákna asynchrónne. Preto nie je dobre sa spoliehať na poradie vykonávania vlákien. Nové vlákno končí buď explicitne volaním `pthread_exit()`, alebo návratom zo `start_routine`.

## 7.3 Synchronizácia chodu vlákien

Vlákna musia synchronizovať svoj chod tak ako procesy. Hlavné vlákno, ktoré vytvára nové vlákna, musí počkať na ich ukončenie. V opačnom prípade sa môže stať, že hlavné vlákno sa ukončí skôr ako ním vytvorené vlákna a alokované štruktúry vlákien sa dealokujú. Vlákno, ktoré pokračuje v svojom vykonaní už nebude mať prístup k ním. Funkcia `pthread_join()` umožňuje hlavnému vláknu počkať na ukončenie iného vlákna.

```
#include <pthread.h>

int pthread_join(pthread_t th, void **thread_return);
```

Argumenty:

- `th` - identifikátor vlákna, na ktoré sa čaká,
- `thread_return` - ukazovateľ na premennú, kde sa uloží návratová hodnota ukončeného vlákna. Vlákno musí byť v „prepojiteľnom“ (`joinable`) stave, t. j. nesmie byť vytvorené ako oddelené (`detached`), ani nesmie byť oddelené neskôr pomocou funkcie `pthread_detach()`. Rozdiel spočíva v tom, že keď skončí vlákno, ktoré je „`joinable`“, jeho prostriedky (popisovač a zásobník) sa nedealokujú kým iné vlákno vykonáva operáciu `pthread_join()` naň.

Na ukončenie vlákna smie čakať najviac jedno vlákno. Volanie `pthred_join()` na vlákne, na ktoré čaká už iné vlákno, končí s chybou.

**Príklad 7.1: Tvorba vlákien a odovzdávanie argumentov vlákna**

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

void* print_arg (void* targ)
/* Funkcia vykonávana vo vlákne. Tlačí znak a argument volania na obrazovke
v nekonečnom cykle. */
   int i;
  fprintf(stdout, " \nThread_PID =%d", getpid());
  fprintf(stdout, " Argument = %s", targ);
  for (i=0;i<100;i++) fprintf(stdout, unused);
  return NULL;
 }

int main ()
{
  pthread_t thread_id;

  pthread_create (&thread_id, NULL, &print_arg, " AHOJ ");

 // Vytvorí nove vlákno, ktoré bude vykonávať funkciu print_arg a odovzdá mu argument

  pthread_create (&thread_id, NULL, &print_arg," NAZDAR ");

// Toto vykonáva hlavne vlákno.
   pthread_join(thread_id1, NULL);
   pthread_join(thread_id2, NULL);
   return 0;
}
```

Vlákna v tomto prípade (v systéme GNU/Debian), vypisujú PID procesu, v ktorom bežia.

## 7.4 Návratová hodnota z vlákna

Návratová hodnota z vlákna sa môže získať pomocou argumentu vlákna (`arg`) vo volaní funkcie `pthread_create()`. Podobne ako argument vlákna, aj návratová hodnota je typu `void *`, takže po získaní tejto hodnoty je potrebné zmeniť jej typ na požadovaný.

## 7.5 Atribúty vlákna

Atribúty vlákna poskytujú mechanizmus pre detailnejšie nastavenie správania sa vlákna. Ak argument atribútov vlákna pri jeho tvorbe je nastavený n NULL, vlákno sa vytvorí so štandardnými atribútmi. Na tento účel slúži funkcia `pthread_attr_init()`.

```
#include <pthread.h>

int pthread_attr_init(pthread_attr_t *attr);
```

Pre nastavenie atribútov vlákna sa musí najskôr vytvoriť štruktúra, typu `pthread_attr_t` a inicializovať ju.

Následne v tej štruktúre je možné zmeniť štandardné nastavenia a použiť ju ako argument pri vytváraní vlákna. Systém potrebuje atribúty len pri vytváraní vlákna. Neskoršia zmena atribútov neovplyvňuje už vytvorené vlákno. To dáva možnosť použitia už vytvorenej `pthread_attr_t` pri tvorbe viacerých vlákien.

Atribúty, ktoré sa môžu nastaviť v `pthread_attr` sú:

- `detachstate` - určuje či vlákno bude vytvorené ako **prepojiteľné** (angl. *joinable*, hodnota PTHREAD_CREATE_JOINABLE) alebo ako **oddelené** (angl. *detached*, hodnotaPTHREAD_CREATE_DETACHED).

  Štandardné nastavenie: PTHREAD_CREATE_JOINABLE

  Ako už bolo vysvetlené, ak vlákno je vytvorené ako `joinable`, niektoré jeho prostriedky sa po jeho ukončení uchovajú a ak iné vlákno vykoná `pthread_join()` nad týmto vláknom, môže ich použiť.

  Ak vlákno je vytvorené ako `detached`, všetky jeho prostriedky sú uvoľnené okamžite po jeho ukončení.

  Ak vlákno nebolo vytvorené ako `detached`, môže sa takým stať pomocou volania `pthread_detach()`.

- `schedpolicy` - určuje jeden algoritmus plánovania z týchto možných: SCHED_OTHER (štandardné plánovanie, nie je pre reálny čas), SCHED_RR (Round Robin, t. j. cyklické plánovanie, pre reálny čas) alebo SCHED_FIFO (plánovanie v poradí príchodu, pre reálny čas).

  Štandardné nastavenie: SCHED_OTHER.

  SCHED_RR a SCHED_FIFO sú prístupné len pre privilegované procesy.

Algoritmus plánovania sa môže zmeniť aj po vytvorení vlákna pomocou funkcie `pthread_setschedparam()`.

- `schedparam` - obsahuje parametre plánovania, vlastne prioritu plánovania pre vlákno.

  Štandardné nastavenie: priorita je 0. Tento argument nie je dôležitý pre plánovanie podľa algoritmu SCHED_OTHER, ovplyvňuje len SCHED_RR a SCHED_FIFO.

  Priorita sa môže zmeniť aj po vytvorení vlákna pomocou funkcie `pthread_setschedparam()`.

- `inheritsched` - určuje či nové vlákno zdedí algoritmus plánovania a parametre plánovania od rodičovského vlákna (PTHREAD_INHERIT_SCHED) alebo budú explicitne určené (PTHREAD_EXPLICIT_SCHED).

  Štandardné nastavenie: PTHREAD_EXPLICIT_SCHED

- `scope` - určuje, s kým si bude súperiť vlákno pri získavaní času procesora. Jediná hodnota, ktorú podporuje implementácia LinuxThreads je `PTHREAD_SCOPE_SYSTEM`, ktorá znamená, že vlákno bude súperiť o čas procesora so všetkými procesmi v systéme. Priority vlákien sú interpretované relatívne k prioritám ostatných procesov v systéme.

  Druhá hodnota, ktorú štandard špecifikuje je `PTHREAD_SCOPE_PROCESS`, ktorá znamená, že vlákno si bude súperiť len so svojimi príbuznými vláknami v rámci jedného procesu. Táto hodnota nie je špecifikovaná v LinuxThreads.

  Štandardné nastavenie: `PTHREAD_SCOPE_SYSTEM`.

Atribúty vlákna sa môžu nastaviť aj jednotlivo pomocou príslušnej funkcie pre daný atribút. Názvy funkcií pre nastavenie atribútu pozostávajú z `pthread_attr_setXXX`, kde XXX je názov atribútu. Názvy funkcií pre získanie informácie o atribúte pozostávajú z `pthread_attr_setXXX`, kde XXX je názov atribútu.

Funkcia `int` **`pthread_attr_destroy`**`(pthread_attr_t *attr);`

ruší `pthread_attr`.

**Príklad 7.2: Fragment programu, ktorý nastavuje atribúty vlákna**

```
…
pthread_attr_t attr;
pthread_t thread;
```

```
pthread_attr_init (&attr);
pthread_attr_setdetachstate (&attr,
PTHREAD_CREATE_DETACHED);
pthread_create (&thread, &attr, &thread_function, NULL);
…
```

## 7.6 Ukončenie vlákna

Za normálnych okolností vlákno sa ukončí buď návratom z vykonávanej funkcie alebo volaním `pthread_exit()`.

```
#include <pthread.h>

void pthread_exit(void *retval);
```

Táto funkcia má podobný význam ako systémové volanie `exit` pri procesoch.

V niektorých prípadoch je potrebné ukončenie určitého vlákna. Anglický termín pre takúto ukončenie je „*canceling*“, t. j. zrušenie. Pre zrušenie vlákna je určená funkcia `pthread_cancel()`.

```
#include <pthread.h>

int pthread_cancel(pthread_t thread);
```

Argument funkcie je ID vlákna.

Zrušené vlákno môže buď ignorovať požiadavku na zrušenie, môže jej vyhovieť okamžite alebo až keď sa dostane do určitého bodu svojho vykonávania, kde môže byť zrušené.

Zrušené vlákno môže byť neskôr pripojené (angl. *joined*). Pokiaľ vlákno je vytvorené ako „*detached*“, nie je potrebné ho zrušiť, aby boli uvoľnené jeho prostriedky.

Zrušenie vlákna, ktoré požiada o nejaké prostriedky, použije ich a potom ich uvoľní môže spôsobiť problémy, ak sa vykoná v nevhodnom okamihu. Preto sú určené stavy, v ktorých vlákno môže alebo nemôže byť zrušené. Vzhľadom na možnosti zrušenia, tieto stavy sú:

- **asynchrónne zrušiteľné** (angl. *asynchronously cancelable*) – to znamená, že môže byť zrušené v ľubovoľnom bode svojho vykonávania,

- **synchrónne zrušiteľné** (angl. *synchronously cancelable*) - môže byť zrušené, ale nie v ľubovoľnom bode svojho vykonávania. Požiadavka na zrušenie sa zaradí do frontu do okamihu keď vlákno dosiahne bod, v ktorom môže byť zrušené,
- **nezrušiteľné** (angl. *uncancelable*) – v tomto stave vlákno nemôže byť zrušené.

Pre riadenie týchto vlastností vlákien sú určené nasledujúce funkcie:

```
#include <pthread.h>

int pthread_cancel(pthread_t thread);

int pthread_setcancelstate(int state, int *oldstate);

int pthread_setcanceltype(int type, int *oldtype);

void pthread_testcancel(void);
```

Vo všeobecnosti nie je dobré využívať zrušenie vlákna na ukončenie jeho činnosti. Zrušenie by sa malo využívať len vo výnimočných situáciách. Správne je oznámiť vláknu, že sa má ukončiť a počkať na jeho ukončenie.

V určitých prípadoch sa môže stať, že vlákno už nepotrebuje čas procesora, ktorý mu bol pridelený a chce sa ho vzdať v prospech ďalších vlákien. To umožňuje funkcia `sched_yield ()`.

```
#include <sched.h>

int sched_yield(void);
```

Vlákno, ktoré zavolalo túto funkciu sa dobrovoľne vzdáva procesora v prospech svojich „príbuzných“ vlákien. Plánovač ho zaradí na koniec frontu a plánuje ďalšie vlákno. Príklad použitia je ukázaný v príklade 7.5.

## 7.7 Vlastné dáta vlákien

Vlákna zdieľajú jeden adresný priestor, takže majú prístup ku všetkým globálnym premenným. Niekedy ale vlákna potrebujú mať svoju vlastnú kópiu globálnych premenných. Štandard POSIX dáva možnosť vytvorenia **dát, špecifické pre vlákno** (ďalej DŠV) tým, že umožní vytvorenie oblasti v pamäti, kde sa

uchovajú. Tam vlákno môže DŠV meniť bez toho, že by ovplyvňovalo ostatné vlákna. Aj keď vlákna zdieľajú jeden adresný priestor, nemôžu sa na takéto dáta odkazovať.

Vlákno môže vytvoriť ľubovoľný počet DŠV položiek a každá má typ `void*`. Na každú položku sa odkazuje ako na kľúč (*key*). DŠV kľúče sú spoločné pre všetky vlákna, ale každé vlákno môže priradiť svoju hodnotu kľúču. Na oblasť obsahujúcu DŠV položky sa môžeme pozerať ako na pole ukazovateľov na položky typu `void*`. DŠV kľúče sú indexmi do poľa a hodnota je zodpovedajúci prvok poľa.

```
#include <pthread.h>

int pthread_key_create(pthread_key_t *key,
                       void (*destr_function) (void  *));
int pthread_key_delete(pthread_key_t key);
int pthread_setspecific(pthread_key_t key,
                        const void *pointer);
void * pthread_getspecific(pthread_key_t key);
```

`pthread_key_create()` alokuje nový DŠV kľúč, ktorý sa uloží do argumentu `key`. Jeho počiatočná hodnota je NULL. Argument `destr_function`() špecifikuje funkciu, ktorá sa zavolá po ukončení vlákna (normálne alebo násilné), aby urobila potrebné „upratovanie“. Jej argument je hodnota DŠV zodpovedajúca tomuto kľúču. Ak táto funkcia nie je potrebná, hodnota je NULL.

`pthread_key_delete()` dealokuje DŠV kľúč. Netestuje či má priradenú nenulovú hodnotu, ani či je definovaná funkcia pre záverečné „upratovanie“.

## 7.8 Synchronizácia vlákien

Problém synchronizácie vlákien je podobný ako pri procesoch. Vlákna zdieľajú spoločný adresný priestor a prístup k spoločným dátam môže viesť k ich nekonzistencii v prípade neriadeného prístupu. Vlákna sa vykonávajú súčasne a nie je možné vedieť presné poradie ich vykonávania, pretože je to závislé od plánovača a od zaťaženia systému. Odladenie takého programu je zložité a navyše chyby, vznikajúce zo súperenia vlákien pri prístupe k spoločným dátam *(*angl. *race condition)* sa nemusia prejaviť vždy! Riešenie tohto problému je použitie synchronizačných prostriedkov.

Pre synchronizáciu vlákien Linux poskytuje iné rozhranie, ktoré sa líši od rozhrania pre procesy. Je založené na štandarde POSIX. Definované sú dva základné synchronizačné prostriedky: **semafory** a **mutexy**. Tieto prostriedky sú si podobné s

tým rozdielom, že mutex funguje ako prostriedok pre vzájomné vylúčenie – pripúšťa k spoločným dátam vždy len jedno vlákno, zatiaľ čo semafor funguje skôr ako počítadlo využitia prostriedku, ktorý má viacej inštancií. Napríklad ak program prideľuje prístup k bufru, ktorý má obmedzenú kapacitu. V tom prípade po zaplnení bufra sa prístup k nemu musí zablokovať kým sa uvoľní nejaká položka.

Ktorý zo synchronizačných prostriedkov sa využije závisí od problému, ktorý sa rieši a od toho, ktorý prostriedok je vhodnejší.

## 7.8.1 Mutexy

Mutex je podobný zámku, ktorý môže uzamknúť len jedno vlákno v danom čase. (Anglické slovo mutex je zložené z častí dvoch ďalších slov: *mutual exclusion*, čo znamená vzájomné vylúčenie. V tomto texte sa používa termín mutex pre jeho jasný význam pre odbornú verejnosť.) Ak sa o uzamknutie pokúsi ďalšie vlákno, bude zablokované, kým prvé vlákno neodomkne mutex. Keď prvé vlákno odomkne mutex, druhé ho bude môcť uzamknúť.

### *7.8.1.1 Inicializácia mutexu*

```
#include <pthread.h>

int  pthread_mutex_init(pthread_mutex_t  *mutex,  const
                        pthread_mutexattr_t *mutexattr);
```

Funkcia `pthread_mutex_init()` inicializuje mutex. Argumenty sú:

- `*mutex` - ukazovateľ na premennú typu `pthread_mutex_t`
- `*mutexattr` - atribúty, podľa ktorých sa inicializuje mutex. Ak tento argument je NULL, nastavenie atribútov bude štandardné.

Implementácia LinuxThreads podporuje len jeden atribút, ktorého hodnota je buď `fast`, `recursive` alebo `errorchecking`. Druh mutexu určuje či môže byť znova uzamknutý vláknom, ktoré ho už raz uzamklo. Štandardný typ je `fast`.

Premenné typu `pthread_mutex` môžu byť inicializované aj staticky pomocou konštánt: `PTHREAD_MUTEX_INITIALIZER` (pre mutexy typu `fast`), `PTHREAD_RECURSIVE_MUTEX_INITIALIZER_NP` (pre mutexy typu `recursive` ) a `PTHREAD_ERRORCHECK_MUTEX_INITIALIZER_NP` (pre mutexy typu `errorchecking`), ako je ukázané ďalej:

```
pthread_mutex_t fastmutex = PTHREAD_MUTEX_INITIALIZER;
pthread_mutex_t recmutex  =
PTHREAD_RECURSIVE_MUTEX_INITIALIZER_NP;
```

```
pthread_mutex_t errchkmutex =
                    PTHREAD_ERRORCHECK_MUTEX_INITIALIZER_NP;
```

#### *7.8.1.2 Uzamknutie mutexu*

```
#include <pthread.h>

int pthread_mutex_lock(pthread_mutex_t *mutex);
```

Táto funkcia uzamyká mutex, zadaný v argumente.

Ak mutex **nebol uzamknutý**, uzamkne sa a jeho vlastníkom bude volajúce vlákno. Návrat z volania je okamžitý. Ak mutex **bol uzamknutý** iným vláknom, **funkcia zablokuje** volajúce vlákno, kým sa mutex odomkne.

Špeciálny prípad je, keď mutex **bol uzamknutý volajúcim vláknom**. V tomto prípade správanie sa funkcie je závislé od druhu mutexu.

- `fast` - volajúce vlákno je pozastavené kým sa mutex neuvoľní. Tento stav prakticky je **uviaznutie** (problém uviaznutia je vysvetlený v [9]), pretože jediné vlákno, ktoré môže odomknúť mutex je práve volajúce vlákno!
- `error checking` – funkcia okamžite vracia z volania chybu EDEADLK.
- `recursive` – funkcia končí okamžite a odpamätá si, koľkokrát bol mutex volajúcim vláknom uzamknutý. Taký istý počet odomknutí sa musí vykonať, kým sa mutex vráti do neuzamknutého stavu.

Existuje ešte jeden variant uzamknutia mutexu, ktorý je uvedený ďalej.

```
#include <pthread.h>

int pthread_mutex_trylock(pthread_mutex_t *mutex);
```

Táto funkcia vykonáva skoro to isté ako `pthread_mutex_lock()`, ale s tým rozdielom, že ak mutex je už uzamknutý nezablokuje volajúce vlákno, ale vráti sa s chybou EBUSY.

#### *7.8.1.3 Odomknutie mutexu*

```
#include <pthread.h>

int pthread_mutex_unlock(pthread_mutex_t *mutex);
```

Táto funkcia odomyká mutex. Predpokladá sa, že volajúce vlákno uzamklo mutex a je jeho vlastníkom. Ak mutex je typu

- `fast` - funkcia vždy odomkne mutex,
- `recursive` – zmenší počet uzamknutí mutexu a len ak tento počet klesne na 0, skutočne odomkne mutex,
- `error checking` – funkcia skontroluje či mutex bol uzamknutý tým istým vláknom, ktoré sa teraz pokúša ho odomknúť. Ak to nie je pravda, vráti sa s chybou a mutex zostáva nezmenený.

Mutexy typu `fast` a `recursive` nevykonávajú také kontroly a to dáva možnosť, aby mutex bol odomknutý a iným vláknom, ako je vlastník mutexu.

#### *7.8.1.4 Zrušenie mutexu*

```
#include <pthread.h>

int pthread_mutex_destroy(pthread_mutex_t *mutex);
```

Funkcia zruší mutex z argumentu. Mutex v momente zrušenia musí byť odomknutý.

**Príklad 7.3 : Použitie mutexu na stráženie kritickej sekcie**

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

void * vlakno(void *);

#define NUM_THREADS 4
pthread_t tid[NUM_THREADS];       /* pole identifikátorov vlakien */

int bignum = 0;                   /* zdieľaná premenná */
pthread_mutex_t mut;

main( int argc, char *argv[] ) {
  int i, ret;

  pthread_mutex_init(&mut, NULL);

  for (i=0; i<NUM_THREADS; i++) {
    pthread_create(&tid[i], NULL, vlakno, (void *)i);
  }
  for ( i = 0; i < NUM_THREADS; i++)
```

```
    pthread_join(tid[i], NULL);

  printf("Hlavny program hlasi, ze %d vlakna sa ukoncili
\n",
          i);
  printf("Hlavny program - bignum=%d\n", bignum);

}  /* main */


void * vlakno(void * parm)
{
  int i;
   printf("Bezi vlakno %d\n", parm);
   for(i=0;i<10000;i++) {
     pthread_mutex_lock(&mut);
       bignum++;   /* Kriticka sekcia, kod kde sa pracuje so zdielanymi datami*/
     pthread_mutex_unlock(&mut);
  }
}
```

### 7.8.2 Semafory pre vlákna

Jedna implementácia semaforov bola popísaná v podkapitole 7.7. Tá je nazývaná implementáciou podľa Systemu V, alebo tiež implementácia semaforov pre procesy. V tejto podkapitole budú popísané semafory podľa štandardu POSIX. Tieto semafory sú použiteľné aj pre procesy aj pre vlákna.

Semafor je celé číslo, ktorého hodnota nikdy nie je záporná. **Nad semaforom sú dovolené dve operácie: zníženie hodnoty semafora a zvýšenie jeho hodnoty**. **Tieto operácie sú atomické.**

Semafory podľa štandardu POSIX majú dve podoby: pomenované a nepomenované.

- **Pomenované semafory** - semafory tohto typu majú svoje meno. Dva procesy môžu pracovať s tým istým semaforom ak poznajú jeho meno a použijú ho pri volaní funkcie `sem_open()`. Táto funkcia vytvorí požadovaný semafor ak neexistuje, alebo otvorí už existujúci semafor.
- **Nepomenované semafory** (pamäťové semafory) – tento typ semaforov nemá meno. Je umiestnený do oblastí pamäte, kde ho môže zdieľať viac vlákien (semafor pre vlákna) alebo viac procesov (semafor pre procesy). Nepomenovaný semafor pre vlákna je umiestnený do oblasti v pamäti, ktorá je zdieľaná medzi vláknami. Nepomenovaný semafor pre procesy musí byť umiestnený do úseku zdieľanej pamäte, získanej pomocou systémového volania `semget()`. Pred použitím nepomenovaný semafor musí byť

inicializovaný pomocou `sem_init()`. Operácie nad semaforom umožňujú volania funkcií `sem_wait()` a `sem_post()`. Po ukončení práce, nepomenovaný semafor sa zruší volaním `sem_destroy()`.

#### *7.8.2.1 Inicializácia nepomenovaného semafora*

```
#include <semaphore.h>

int sem_init(sem_t *sem, int pshared, unsigned int
             value);
```

Funkcia inicializuje nepomenovaný semafor na adresu, na ktorú ukazuje premenná `*sem`. Tretí argument volania – `value`, je počiatočná hodnota semafora. Argument `pshared` indikuje či semafor bude zdieľaný medzi vláknami alebo medzi procesmi. Význam hodnôt argumentu `pshared`:

- 0 – zdieľanie semafora medzi vláknami jedného procesu. Taký semafor by mal byť umiestnený na adresu „viditeľnú“ pre všetky vlákna, napr. globálna premenná alebo pamäť pridelená dynamicky.
- nenulová hodnota – znamená zdieľanie semafora medzi procesmi. Taký semafor by mal byť umiestnený na adresu v rámci zdieľanej pamäte. Proces potomok (vytvorený pomocou `fork()`) dedí od rodiča všetky pamäťové referencie a tiež môže používať semafor.

**Inicializácia už inicializovaného semafora môže mať nedefinované výsledky!!!**

#### *7.8.2.2 Operácie* `sem_wait()` *a* `sem_trywait()`

```
#include <semaphore.h>

int sem_wait(sem_t *sem);
int sem_trywait(sem_t *sem);
```

Funkcia `sem_wait()` zníži hodnotu semafora, na ktorý ukazuje argument `sem`. Ak momentálna

- hodnota semafora j>0 – hodnota sa zníži a funkcia sa vráti,
- hodnota semafora = 0 – volanie sa zablokuje kým nebude možné hodnotu znížiť, t. j. hodnota bude väčšia ako 0.

Funkcia `sem_trywait()`má skoro taký istý účinok jako `sem_wait()`, ale ak nie je možné hodnotu semafora znížiť (došlo by k zablokovaniu), funkcia sa vráti s chybou EAGAIN.

#### *7.8.2.3 Získanie hodnoty semafora*

```
#include <semaphore.h>

int sem_getvalue(sem_t *sem, int *sval);
```

Funkcia `sem_getvalue()` uloží do premennej, na ktorú ukazuje `*sval` momentálnu hodnotu semafora. Táto funkcia slúži iba pre informáciu, ale nie je možné založiť prístup vlákna k zdieľaným dátam na základe získanej hodnoty semafora. Príčinou je, že kým sa funkcia ukončí, hodnota semafora môže byť zmenená. **K synchronizačným účelom slúžia iba atomické operácie,** akými sú `sem_wait()` a `sem_post()`.

#### *7.8.2.4 Zrušenie semafora*

```
#include <semaphore.h>

int sem_destroy(sem_t *sem);
```

Funkcia `sem_destroy()` zruší nepomenovaný semafor, nachádzajúci sa na adrese, na ktorú ukazuje argument `*sem`.

Pomocou tejto funkcie je možné zrušiť jedine semafor, ktorý bol inicializovaný pomocou `sem_init()`. Zrušenie semafora, na ktorom sú zablokované iné procesy alebo vlákna vedie k nepredvídateľným výsledkom. Obdobné výsledky by boli aj v prípade, ak sa zruší už zrušený semafor!

**Príklad 7.4 : Použitie vláknových semaforov**

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

void * simple1(void *);
void * simple2(void *);

#define NUM_THREADS 3
pthread_t tid[NUM_THREADS];        /* pole identifikátorov vlákien */
```

```
sem_t semA, semB;

main( int argc, char *argv[] )
{
  int i, ret;

  sem_init(&semA, 0, 0);
  sem_init(&semB, 0, 0);

  pthread_create(&tid[0], NULL, simple1, (void *)1);
  pthread_create(&tid[1], NULL, simple2, (void *)2);
  pthread_create(&tid[2], NULL, simple2, (void *)3);

  for ( i = 0; i < NUM_THREADS; i++)
    pthread_join(tid[i], NULL);

  printf("\n Hlavny program hlasi, ze vsetky %d vlakna sa
            ukoncili\n", i);

}  /* koniec main */

void * simple1(void * param)
{

   printf("Vlakno %d caka na semafor A\n", param);
   sem_wait(&semA);
   printf("     Vlakno %d pracuje v KS_A\n", param);
   printf("Vlakno %d vychadza z KS_A\n", param);
   sem_post(&semA);
   sleep(3);
   printf("          Vlakno %d caka na  semafor B\n",
param);
   sem_wait(&semB);
   printf("                     Vlakno %d pracije v
KS_B\n", param);
   printf("             Vlakno %d vychadza z KS_B\n",
param);
   sem_post(&semB);
}

void * simple2(void * param)
{
   printf("Vlakno %d caka na  semafor A\n", param);
   sem_wait(&semA);
   printf("     Vlakno %d pracije v KS_A\n", param);
   printf("Vlakno %d vychadza z KS_A\n", param);
   sem_post(&semA);
```

```
    printf("         Vlakno %d caka na semafor B\n", param);
    sem_wait(&semB);
    printf("            Vlakno %d pracuje v KS_B\n", param);
    printf("           Vlakno %d vychadza z KS_B\n", param);
    sem_post(&semB);
    sleep(3);
}
```

**Výstup z príkladu**:

```
Vlakno 1 caka na semafor A
        Vlakno 1 pracuje v KS_A
Vlakno 1 vychadza z KS_A
Vlakno 2 caka na  semafor A
        Vlakno 2 pracije v KS_A
Vlakno 2 vychadza z KS_A
                Vlakno 2 caka na semafor B
                        Vlakno 2 pracuje v KS_B
                Vlakno 2 vychadza z KS_B
Vlakno 3 caka na  semafor A
        Vlakno 3 pracije v KS_A
Vlakno 3 vychadza z KS_A
                Vlakno 3 caka na semafor B
                        Vlakno 3 pracuje v KS_B
                Vlakno 3 vychadza z KS_B
                Vlakno 1 caka na  semafor B
                        Vlakno 1 pracije v KS_B
                Vlakno 1 vychadza z KS_B

 Hlavny program hlasi, ze vsetky 3 vlakna sa ukoncili
```

Príklad ukazuje synchronizáciu prístupu troch vlákien k dvom kritickým sekciám. Prístup ku každej sekcii je chránený semaforom. Vlákna pristupujú ku kritickým sekciám v rôznom poradí a preto je potrebné synchronizovať ich prístup. Pri programovaní zložitejších situácií je nutné dávať pozor na poradie operácií `sem_wait()` a `sem_post()`, pretože nesprávne poradie môže viest k uviaznutiu.

### 7.8.3 Podmienkové premenné

Podmienkové premenné *(*angl. *condition variables)* poskytujú ďalšie možnosti pri riešení synchronizačných problémov. GNU/Linux ich ponúka ako ďalší prostriedok pre zložitejšie úlohy, ktoré vlákna vykonávajú.

Podmienkové premenné dovoľujú vláknam, aby pozastavili svoje vykonanie a uvoľnili čas procesora iným vláknam, kým bude splnená nejaká podmienka, súvisiaca so zdieľanými dátami. Základné operácie nad podmienkovými

premennými sú: signalizovať keď podmienka je splnená a čakať na podmienku, kým iné vlákno nesignalizuje jej splnenie.

Podmienková premenná nemá hodnotu. Ak vlákno **A** čaká na podmienkovej premennej, bude blokované, kým vlákno **B** nesignalizuje splnenie podmienky. Ale ak vlákno **B** signalizuje skôr ako vlákno **A** začne čakať, signál bude stratený!

Podmienková premenná je typu `pthread_cond_t` a musí byť vždy spojená s mutexom, ktorý sa inicializuje samostatne.

Funkcie, ktoré manipulujú s podmienkovými premennými sú:

```
#include <pthread.h>

pthread_cond_t cond = PTHREAD_COND_INITIALIZER;

int pthread_cond_init(pthread_cond_t *cond,
                      pthread_condattr_t *cond_attr);
int pthread_cond_signal(pthread_cond_t *cond);
int pthread_cond_broadcast(pthread_cond_t *cond);
int pthread_cond_wait(pthread_cond_t *cond,
                       pthread_mutex_t *mutex);
int pthread_cond_destroy(pthread_cond_t *cond);
```

Činnosť jednotlivých funkcií je vysvetlená ďalej.

- `pthread_cond_init()` – inicializuje podmienkovú premennú. Jej argumenty sú:
  - `*cond` – ukazovateľ na inštanciu typu `pthread_cond_t`,
  - `*cond_attr` – tento argument v GNU/Linux je ignorovaný.
- `pthread_cond_signal()` – signalizuje podmienkovú premennú. Jedno vlákno z čakajúcich na ňu, je odblokované. Ak žiadne vlákno nečakalo na túto podmienkovú premennú, signál je ignorovaný. Argument je ukazovateľ na inštanciu typu `pthread_cond_t`.
- `pthread_cond_broadcast()` – funkcia podobná predchádzajúcej s tým rozdielom, že signalizuje splnenie podmienky všetkým vláknam, ktoré čakajú na podmienkovej premennej.
- `pthread_cond_wait()`- blokuje volajúce vlákno kým iné vlákno nesignalizuje splnenie podmienky. Prvý argument je ukazovateľ na inštanciu `pthread_cond_t`, druhý je ukazovateľ na mutex. Keď sa zavolá táto funkcia, mutex musí už byť uzamknutý volajúcim vláknom. Funkcia odomkne mutex a zablokuje vlákno na podmienkovej premennej. Po signalizácii podmienkovej

premennej sa volajúce vlákno odblokuje a `pthread_cond_wait()` automaticky znova uzamyká mutex.
Uvoľnenie mutexu a zastavenie nad podmienkovou premennou sa deje automaticky. Takže ak všetky vlákna získajú mutex pred signálom pre splnenie podmienky je garantované, že signál sa nemôže vyskytnúť v čase medzi uzamknutím mutexu a začiatkom čakania na podmienku.

- `pthread_cond_destroy()` – zruší podmienkovú premennú. V čase zrušenia žiadne vlákna nesmú čakať na nej.

**Príklad použitia**: predpokladáme, že premenné **x** a **y** sú zdieľané medzi vláknami a sú chránené mutexom *mut* a podmienkovou premennou `cond`, ktorá signalizuje splnenie podmienky **x > y** .

1. **Inicializácia, statická:**

```
int x,y;
pthread_mutex_t mut = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t cond = PTHREAD_COND_INITIALIZER;
```

2. **Čakanie na podmienku x > y** sa vykoná následujúcim spôsobom:

```
pthread_mutex_lock(&mut);
while (x <= y) {
    pthread_cond_wait(&cond, &mut);
}
/* tu sa vykonajú operácie nad x a y, ktoré dovedú k splneniu podmienky */
pthread_mutex_unlock(&mut);
```

3. **Modifikácie nad x a y**, ktoré môžu spôsobiť to, že **x > y** a mali by signalizovať splnenie podmienky, ak je to potrebné

```
pthread_mutex_lock(&mut);
/* modifikácie x a y */
if (x > y) pthread_cond_broadcast(&cond);
   pthread_mutex_unlock(&mut);
```

Ak je isté, že napr. **len jedno** vlákno čaká na splnenie podmienky, potom miesto
`pthread_cond_broadcast()` sa môže použiť `pthread_cond_signal()`.

**Príklad 7.5: Ukážka použitia mutexu a podmienkovej premennej s čakaním na splnenie podmienky**

```
#include <pthread.h>
#include <stdio.h>
#define NUM_THREADS 2

int x=1;
int y=10;
pthread_mutex_t mut = PTHREAD_MUTEX_INITIALIZER; //statická
inicializácia
pthread_cond_t cond = PTHREAD_COND_INITIALIZER; //statická
inicializácia

pthread_t tid[NUM_THREADS];

void * zmena1(){
 int i;
  printf("zmena1 +++ pociatocna hodnota x=%d\n",x);

  for (i=0;i<15;i++)
   {
     pthread_mutex_lock(&mut);
      x++;
     printf("Vlakno c.1   x=%d\n",x);
     if (x > y) {
       pthread_cond_signal(&cond);
   }
     pthread_mutex_unlock(&mut);
     sched_yield();
    /* vlákno sa vzdáva procesora v prospech druhého vlákna. Plánovač ho odsunie
na
       koniec   frontu vlákien a začne vykonávať ďalšie vlákno */
  }
}

void * zmena2(){
  printf(" zmena2 --- pociatocna hodnota x=%d\n",x);
  pthread_mutex_lock(&mut);
  // mutex musí být uzamknutý před tím jako vlákno začne čakať na podmienku!!!
  while (x <= y) {
        printf("Cakam na podmienku\n");
        pthread_cond_wait(&cond, &mut);
      }
       printf(" Podmienka splnena, bezi druhe vlakno \n");
             x= x-10;   // zmena hodnoty zdieľanej premennej
       pthread_mutex_unlock(&mut); }
```

```
main( int argc, char *argv[] )
 {
int i;

pthread_create(&tid[1], NULL, &zmena2, NULL);
pthread_create(&tid[0], NULL, &zmena1, NULL);

for ( i = 0; i < NUM_THREADS; i++)
      pthread_join(tid[i], NULL);

printf("Na konci x=%d\n",x);
return 0;
}
```

**Príklad 7.6: Klasický synchronizačný problém Producent-Konzument pomocou podmienkových premenných a mutexu**

```
#define BSIZE 4
#define NUMITEMS 30
#define NUM_THREADS 2

 char buf[BSIZE];
 int plne;
 int nextin, nextout;
 pthread_mutex_t mutex;
 pthread_cond_t full;
 pthread_cond_t empty;

 void * producent(void *);
 void * konzument(void *);

 pthread_t tid[NUM_THREADS];       /* pole identifikátorov vlákien  */

 main( int argc, char *argv[] )
 {
  int i;

  pthread_cond_init(&(full), NULL);
  pthread_cond_init(&(empty), NULL);

  pthread_create(&tid[1], NULL, konzument, NULL);
  pthread_create(&tid[0], NULL, producent, NULL);

  for ( i = 0; i < NUM_THREADS; i++)
        pthread_join(tid[i], NULL);
```

```
 printf("\nmain() hlasi, ze vlakna skoncili \n");

}  /*main end  */


void * producent(void * parm)
{
  char item[NUMITEMS]="VZDY ROB NAJLEPSIE AKO VIES! \0";
  int i;

  printf("start producenta \n");

    for (i=0; i<NUMITEMS; i++)
    {
       /* vyrobi polozku , jeden znak z pola item[] */

      if (item[i]=='\0') buf[nextin++] = item[i];

     /* konci, ak je koniec retazca. */
      pthread_mutex_lock(&(mutex));

        if (plne >= BSIZE) printf("producent caka.\n");

        while (plne >= BSIZE)
              pthread_cond_wait(&(empty), &(mutex) );
        buf[nextin++] = item[i];
        nextin %= BSIZE;
        plne++;

   /*
      Buď plne < BSIZE a nextin je index  ďalšej prázdnej položky v bufri,
      alebo  plne == BSIZE a nextin je index položky ktorá bude vybraná
      konzumentom
  */

        pthread_cond_signal(&(full));
      pthread_mutex_unlock(&(mutex));

    } //koniec for

  printf("producent konci.\n");
  pthread_exit(0);
}

void * konzument(void * parm)
{
  char item;
```

```
    int i;
    printf("   start konzumenta \n");

    for(i=0; i<NUMITEMS; i++){
       pthread_mutex_lock(&(mutex) );
         if ( plne <= 0) printf("  konzument  caka\n");

         while ( plne <= 0 )
             pthread_cond_wait(&( full), &( mutex) );

         item =  buf[ nextout++];
         if (item == '\0')   {
          /* konci, ak je koniec reťazca. */
                printf(  "KONIEC DAT\n");
                pthread_cond_broadcast(& full);
                pthread_exit(0);
                }
       printf("     Konzument vybral: %c\n", item);
       nextout %= BSIZE;
       plne--;

  /*
      Buď  plne > 0 a  nextout je  index ďalšej plnej položky  v bufri,
      alebo  plne == 0  a  nextout je index ďalšej prázdnej položky, ktorú  producent
      bude plniť
  */

            pthread_cond_signal(&( empty));
      pthread_mutex_unlock(&( mutex));
    }
    printf("   Konzument  konci.\n", parm);
    pthread_exit(0);
  }
```

Tento program je ukážkou riešenia klasického synchronizačného problému producent-konzument pomocou vlákien, pričom čakanie na splnenie podmienky (naplnenie bufra u konzumenta a uvoľnenie bufra u producenta) sa uskutočňuje podmienkovými premennými. Ich využitie sa môže zdať na prvý pohľad zložité, ale stačí si uvedomiť skutočnosť, že podmienková premenná je viazaná na mutex, ktorý sa uzamkne predtým ako je potrebné čakať na splnenie podmienky a po jej splnení sa mutex odomkne. Volania, ktoré vykonávajú popísané činnosti sú v texte programu zvýraznené.

**Výstup z programu:** Vplyv plánovania vlákien v rámci procesu je viditeľný, keď sa program spustí viackrát. Vykonanie vlákien môže mať aj iné poradie.

```
start konzumenta
  konzument  caka
start producenta
     Konzument vybral: V
  konzument  caka
     Konzument vybral: Z
  konzument  caka
     Konzument vybral: D
  konzument  caka
     Konzument vybral: Y
  konzument  caka
     Konzument vybral:
  konzument  caka
     Konzument vybral: R
  konzument  caka
     Konzument vybral: O
  konzument  caka
     Konzument vybral: B
  konzument  caka
     Konzument vybral:
  konzument  caka
     Konzument vybral: N
  konzument  caka
     Konzument vybral: A
  konzument  caka
     Konzument vybral: J
  konzument  caka
     Konzument vybral: L
  konzument  caka
     Konzument vybral: E
  konzument  caka
     Konzument vybral: P
  konzument  caka
producent caka.
     Konzument vybral: S
     Konzument vybral: I
     Konzument vybral: E
     Konzument vybral:
  konzument  caka
producent caka.
     Konzument vybral: A
     Konzument vybral: K
     Konzument vybral: O
     Konzument vybral:
  konzument  caka
producent caka.
     Konzument vybral: V
     Konzument vybral: I
     Konzument vybral: E
     Konzument vybral: S
  konzument  caka
producent konci.
     Konzument vybral: !
     Konzument vybral:
KONIEC DAT

main() hlasi, ze vlakna
skoncili
```

## 7.9 Záver

Uvedená kapitola popísala spôsob vytvorenia vlákien v rámci procesu s cieľom efektívnejšieho využitia času procesora, prideleného procesu. Výsledný program v skutočnosti bude efektívnejší len za predpokladu, že vlákna nevyužívajú súčasne rovnaký prostriedok systému, ale každé využíva iný - napr. jedno vlákno vykonáva výpočet, druhé zapisuje údaje na disk, tretie zachytáva vstupy z klávesnice a pod.