Synchronizácia vlákien a kritické oblasti

- Zámky
- Semafóry
- Podmienkové premenné

Synchronizácia

- Zaistenie konzistencie zdieľaných údajov
- Postupný prístup viacerých vlákien k synchronizačným objektom
- Zaistenie bezpečnosti spoločných údajov

Niektoré riziká pri programovaní vlákien

- Race conditions:

  - Kód funkcie vlákna sa nevykonáva postupne v tom poradí ako je napísaný, keďže je procesor prideľovaný vláknam z hľadiska programátora náhodne.

  - Vlákna sa nemusia vykonávať v takom poradí ako boli vytvorené a môžu sa vykonávať rôznou rýchlosťou.

  - Dôsledkom toho sú neočakávané výsledky tzv. race conditions.

  - Aby bol výsledok vlákien predikovateľný musia sa využívať prostriedky na ich synchronizáciu.

- Bezpečnosť kódu funkcií vlákien:

  - Funkcie vlákien musia byť bezpečné (thread safe), nesmú obsahovať žiadne statické a globálne premenné bez ochrany súčasného prístupu k ním len pre jedno vlákno

  - Funkcie, ktoré používajú statické a globálne premenné sú nie bezpečné (thread unsafe), môže ich používať len jedno vlákno

Reentrantné funkcie

- Funkcia vlákna je bezpečná ak všetky funkcie, ktoré volá sú reentrantné.
- Reentrantná funkcia neobsahuje žiadne globálne ani statické premenné alebo ak ich obsahuje musí byť chránený prístupu k nim súčasne len pre jedno vlákno.

Príklad nereentrantnej funkcie

- Mnohé nereentrantné funkcie vracajú smerník na premennú typu static.
- Ak je takáto funkcia volaná vo funkcii vlákna, ktoré vykonáva viac vlákien hodnoty sa budú prepisovať:

```
void * called_function ( void *)
{
        static char buffer [ 64 ] ;
        // Fill up the buffer with something good.
        return (void *)buffer ;
}
```

Príklad reentrantnej funkcie

- Nasledovná verzia vrátenia návratovej hodnoty funkcie bude korektná aj keď ju vykonáva viac vlákien:

```
void * called_function ( void *)
{
    char *buffer =(char *)malloc( 64) ;
    // Fill up the buffer with something good.
    return (void *)buffer ;
}
```

Synchronizácia - príklad

- Predpokladajme, že úlohy zo zreťazeného zoznamu sú spracovávané niekoľkými konkurentnými vláknami.
- Každé vlákno po skončení testuje zoznam úloh, ak je nie prázdny vyberie úlohu a nastaví vrchol zoznamu na nasledujúcu úlohu.
- Funkcia vlákna, ktorá spracúva úlohy zoznamu môže vyzerať nasledovne:

Thread Function to Process Jobs from the Queue

```
#include <malloc.h>
struct job {
  /* Link field for linked list. */
  struct job* next;
  /* Other fields describing work to be done... */
};
/* A linked list of pending jobs. */
struct job* job_queue;
```

```
/* Process queued jobs until the queue is empty. */
void* thread_function (void* arg)
{
  while (job_queue != NULL) {
    /* Get the next available job. */
    struct job* next_job = job_queue;
    /* Remove this job from the list. */
    job_queue = job_queue->next;
    /* Carry out the work. */
    process_job (next_job);
   /* Clean up. */
    free (next_job);
  }
  return NULL;
}
```

Synchronizácia

- Problém vznikne ak dve vlákna ukončia svoje úlohy približne v rovnakom čase a vo fronte zostane už len jedna úloha.
- Prvé vlákno zistí, že fronta je nie prázdna, priradí adresu úlohy do next_job a je prerušené.
- Druhé vlákno urobí to isté a keďže úloha ešte nebola zo zoznamu vybraná obe vlákna sa budú snažiť vykonať tú istú úlohu!
- Aby to nenastalo, testovanie a vybratie úlohy z fronty musí byť **atomická** (neprerušiteľná) operácia.

Zámka – MUTEX (MUTual EXclusion Locks)

- **Používa sa na blokovanie prístupu do CS viac vláknam súčasne. Zabezpečuje exkluzívny prístup do CS pre vlákno, ktoré zámku používa.**
- Mutex je špeciálna zámka, ktorá môže súčasne uzamknúť len jedno vlákno.
- Ak je zámka už uzamknutá a druhé vlákno sa pokúša ju znovu uzamknúť je zablokované.
- Zámku môže odomknuť len to vlákno, ktoré ju uzamklo.
- Len ak prvé vlákno odomkne zámku je druhé vlákno odblokované a môže pokračovať.

Zámky

- Zámka sa deklaruje typom pthread_mutex_t
- Inicializuje sa funkciou
  pthread_mutex_init(pthread_mutex_t* pmut, pthread_mutexattr_t* pattrmut)

  kde

  prvý argument je smerník na zámku

  druhý argument je smerník na atribúty zámky

  ak je druhý argument NULL predpokladajú sa defaultové atribúty.
- Nasledovný kód demonštruje deklaráciu a inicializáciu:

```
pthread_mutex_t mutex;
pthread_mutex_init (&mutex, NULL);
```

Dynamická inicializácia

```
#include <pthread.h>
#include “errors.h”
/*
 * Define a structure, with a mutex.
 */
typedef struct my_struct_tag {
    pthread_mutex_t   mutex;   /* Protects access to value */
    int  value;                /* Access protected by mutex */
} my_struct_t;
```

```
int main (int argc, char *argv[])
{
    my_struct_t *data;
    int status;

    data = malloc (sizeof (my_struct_t));
    if (data == NULL)
        errno_abort ("Allocate structure");
    status = pthread_mutex_init (&data->mutex, NULL);
    if (status != 0)
        err_abort (status, "Init mutex");
    status = pthread_mutex_destroy (&data->mutex);
    if (status != 0)
        err_abort (status, "Destroy mutex");
    (void)free (data);
    return status;
}
```

Zámky

- Jednoduchší spôsob inicializácie je statická inicializácia:

pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;

- Nie je nutné volať inicializačnú funkciu.
- Takáto inicializácia je vhodná obzvlášť pre globálne premenné (v C++ statické členy).

Príklad statickej inicializácie

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

typedef struct my_struct_tag {
    pthread_mutex_t   mutex;  /* Protects access to value */
    int  value;                         /* Access protected by mutex */
} my_struct_t;
my_struct_t data = {PTHREAD_MUTEX_INITIALIZER, 0};

int main (int argc, char *argv[])
{
    return 0;
}
```

Zámky

- Zámka sa uzamkne volaním funkcie pthread_mutex_lock(pthread_mutex_t*)
- Ak zámka nebola už uzamknutá uzamkne sa a funkcia okamžite ukončí návratom.
- Ak už bola uzamknutá iným vláknom výkon funkcie je zablokovaný a pokračuje len ak je zámka odomknutá iným vláknom.
- Na uzamknutej zámke môže byť blokovaných aj viac vlákien.
- Keď je zámka odomknutá je odblokované len jedno vlákno (nepredikovateľne), ostatné ostanú zablokované.
- Vlákno je odblokované volaním pthread_mutex_unlock(pthread_mutex_t*). Funkciu môže volať len to vlákno, ktoré zámku uzamklo.

Zámky-príklad

Job Queue Thread Function, Protected by a Mutex

```
#include <malloc.h>
#include <pthread.h>
struct job {
  /* Link field for linked list. */
  struct job* next;
  /* Other fields describing work to be done... */
};
/* A linked list of pending jobs. */
struct job* job_queue;
/* A mutex protecting job_queue. */
pthread_mutex_t job_queue_mutex =
    PTHREAD_MUTEX_INITIALIZER;
```

```
void* thread_function (void* arg)
{
  while (1) {
    struct job* next_job;
    /* Lock the mutex on the job queue. */
    pthread_mutex_lock (&job_queue_mutex);
    /* Now it's safe to check if the queue is empty. */
    if (job_queue == NULL)
      next_job = NULL;
    else {
      /* Get the next available job. */
      next_job = job_queue;
      /* Remove this job from the list. */
      job_queue = job_queue->next;
   }
```

```
    /* Unlock the mutex on the job queue because we're done
       with the queue for now. */
    pthread_mutex_unlock (&job_queue_mutex);
    /* Was the queue empty? If so, end the thread. */
    if (next_job == NULL)
      break;
    /* Carry out the work. */
    process_job (next_job);
    /* Clean up. */
    free (next_job);
  }
  return NULL;
}
```

Poznámky

- Prístup k zdieľanej fronte úloh job_queue je riadený zámkou job_queue_mutex a je len medzi volaniami funkcií pthread_mutex_lock a pthread_mutex_unlock
- Prístup k premennej next_job je aj mimo tejto oblasti, keďže úloha už bola z fronty vybraná a teda je iným vláknam nedostupná.
- Ak je fronta prázdna zo slučky nemožno vyskočiť okamžite ale len po odomknutí zámky, inak by sa bránilo znovu pristupovať ostatným vláknam k fronte úloh.

- Nasledovný príklad ukazuje ďalšie použitie zámky pri vkladaní úloh do fronty:

```
void enqueue_job (struct job* new_job)
{
  pthread_mutex_lock (&job_queue_mutex);
  new_job->next = job_queue;
  job_queue = new_job;
  pthread_mutex_unlock (&job_queue_mutex);
}
```

Príklad prístupu vlákien ku globálnej premennej

- Nasledovné funkcie používajú zámku pri prístupe k premennej *count*
- Funkcia, ktorá premennú inkrementuje používa zámku aby operácia inkrementovania globálnej premennej bola atomická
- Funkcia, ktorá globálnu premennú vracia používa zámku na to aby operácia čítania bola atomická, keďže premenná *count* je typu long long čo je 64-bitov a na 32-bitovej architektúre sa vykoná na 2-krát

Príklad

```
#include <pthread.h>
pthread_mutex_t count_mutex;
long long count;
void increment_count( ){
    pthread_mutex_lock(&count_mutex);
   count = count + 1;
   pthread_mutex_unlock(&count_mutex);
}
long long get_count( ){
   long long c;
   pthread_mutex_lock(&count_mutex);
   c = count;
   pthread_mutex_unlock(&count_mutex);
   return (c);
}
```

Typy zámok

- *Rýchly mutex* (default) môže spôsobiť deadlock v prípade ak sa ho pokúša znovu zamknuť to isté vlákno a je na zámke tiež blokované a nemôže ho teda odomknúť.
- *Rekurzívny mutex* nespôsobuje deadlock. Môže byť zamknutý niekoľko krát tým istým vláknom. Zámka si pamätá počet zamknutí. Na odomknutie sa musí toľkokrát volať funkcia pthread_mutex_unlock.
- *Error-checking mutex* nespôsobí deadlock lebo detekuje po ďalšom uzamknutí zámky tým istým vláknom chybový stav EDEADLK.

Tvorba zámok iných typov

- Treba deklarovať atribúty zámky typu
  pthread_mutexattr_t
- Inicializovať atribúty volaním
  pthread_mutexattr_init(pthread_mutexattr_t*)
- Nastaviť atribúty funkciou
  pthread_mutexattr_setkind_np(pthread_mutexattr_t*, PTHREAD_MUTEX_RECURSIVE_NP)
  alebo s druhým argumentom PTHREAD_MUTEX_ERRORCHECK_NP
  alebo pthread_mutexattr_settype() s tými istými argumentami
- Inicializovať zámku volaním funkcie
  pthread_mutex_init(pthread_mutex_t*, pthread_mutexattr_t*)
- Atribúty sa zrušia volaním funkcie
  pthread_mutexattr_destroy(pthread_mutexattr_t*)

Tvorba zámok iných typov

- Nasledovný kód ilustruje použitie error-checking mutex:

```
pthread_mutexattr_t attr;
pthread_mutex_t mutex;
pthread_mutexattr_init (&attr);
pthread_mutexattr_setkind_np (&attr,
    PTHREAD_MUTEX_ERRORCHECK_NP);
pthread_mutex_init (&mutex, &attr);
pthread_mutexattr_destroy (&attr);
```

Hierarchia zámok a deadlocky

- deadlock je stav keď sú všetky vlákna blokované a nemôžu pokračovať
- problém deadlocku môže nastať ak vlákna uzamkýnajú viac zámiek bez poradia uzamknutia
- technika, ktorá deadlock rieši sa volá *hierarchia uzamkýnania zámok* a spočíva v tom, že sa musí **dodržať poradie uzamkýnania** zámok každým vláknom

Príklad1 deadlocku

Thread 1

```
/* use resource 1 */
pthread_mutex_lock(&m1);
/* NOW use resources 2 + 1 */
pthread_mutex_lock(&m2);
pthread_mutex_unlock(&m1);
```

Thread 2

```
/* use resource 2 */
pthread_mutex_lock(&m2);
/* NOW use resources 1 + 2 */
pthread_mutex_lock(&m1);
pthread_mutex_unlock(&m2);
```

- Nastane deadlock, keďže vlákna uzamkýnajú zámky v rôznom poradí

Príklad2 –Resource deadlock

```
struct {
pthread_mutex_t mutex;
char *buf;
} A;
struct {
pthread_mutex_t mutex;
char *buf;
} B;
struct {
pthread_mutex_t mutex;
char *buf;
} C;
```

```
use_all_buffers() {
pthread_mutex_lock(&A.mutex);
/* use buffer A */
pthread_mutex_lock(&B.mutex);
/* use buffers B */
pthread_mutex_lock(&C.mutex);
/* use buffer C */
/* All done */
pthread_mutex_unlock(&C.mutex);
pthread_mutex_unlock(&B.mutex);
pthread_mutex_unlock(&A.mutex);
}
```

```
use_buffer_a() {
pthread_mutex_lock(&A.mutex);
/* use buffer A */
pthread_mutex_unlock(&A.mutex);
}
functionB() {
pthread_mutex_lock(&B.mutex);
/* use buffer B */
  if (..some condition) {
  use_buffer_a();
  }
pthread_mutex_unlock(&B.mutex);
}
```

```
/* Thread A */                  /* Thread B */
use_all_buffers();                 functionB();
```

- Ak najprv štartuje vlákno B uzamkne B.mutex a vlákno A zostane čakať na jeho získanie (vo funkcii use_all_buffers() s uzamknutým A.mutex).

  Keď je splnená podmienka vo funkcii functionB() vlákno B sa pokusi získať A.mutex, ktorý je už uzamknutý vláknom A - výsledkom je deadlock.

Uzamknutie zámky bez blokovania

- Používa sa na predchádzanie deadlocku
- Vlákno môže vykonávať inú činnosť namiesto čakania na odomknutie zámku.
- Zabezpečí to funkcia *pthread_mutex_trylock*
- Ak je zámka uzamknutá iným vláknom vlákno na tejto funkcii nezostane zablokované stáť ale vracia chybový kód EBUSY (errno.h) a môže pokračovať v inej činnosti. Neskôr sa môže opäť pokúsiť zamknúť zámku.
- Ak je zámka neuzamknutá funkcia vracia 0.

Uzamknutie zámky bez blokovania

Príklad

Thread 1:

```
pthread_mutex_lock(&m1);
pthread_mutex_lock(&m2);
/* no processing */
pthread_mutex_unlock(&m2);
pthread_mutex_unlock(&m1);
```

Thread 2:

```
for (; ;) {
  pthread_mutex_lock(&m2);
  if(pthread_mutex_trylock(&m1)==0)
  /* got it! */
  break;
  /* didn't get it */
  pthread_mutex_unlock(&m2);
}
/* get locks; no processing */
pthread_mutex_unlock(&m1);
pthread_mutex_unlock(&m2);
```

Uzamknutie zámky bez blokovania

- V príklade vlákno1 uzamkýna zámky m1, m2 v danom poradí, zatiaľ čo vlákno2 mimo poradia. Aby nenastal deadlock musí vlákno2 zamkýnať zámku m1 volaním ...trylock() .
- Deadlock by nastal ak by bolo vlákno1 po uzamknutí m1 prerušené vláknom2 a to by miesto trylock použilo na uzamknutie m1 funkciu pthread_mutex_lock.
- Deadlock nenastane ak vlákno2 použije pthread_mutex_trylock pre uzamknutie m1, ak je totiž m1 už zamknuté, vlákno2 je nie blokované ale môže uvolniť m2, aby ho mohlo vlákno1 uzamknuť a následne uvolniť obe zámky m1 a m2

Semafóry

- Uvažujme opäť zreťazený zoznam úloh, ktoré vykonávajú vlákna. Program pracuje ak je fronta úloh vytvorená predom alebo ak prichádzajú nové úlohy do fronty aspoň tak rýchlo ako sú spracovávané vláknami. Avšak ak vlákna pracujú veľmi rýchlo fronta úloh bude prázdna a vlákna budú zrušené. Ak budú neskôr prichádzať do fronty ďalšie úlohy už nebude vlákien na ich spracovanie.
- Je treba mechamizmus, ktorý bude vlákna v prípade prázdnej fronty blokovať pokiaľ nebudú vo fronte nové úlohy.

Semafóry

- Semafóry sú vhodné prostriedky na synchronizáciu viacerých vlákien.
- Semafór obsahuje počítadlo hodnoty, ktorého sú nezáporné čísla.
- Obsahuje tiež dve základné operácie:

  wait – ak je hodnota 0 vlákno je na semafóre blokované, pokiaľ nebude jeho hodnota kladná a hodnota semafóra sa dekrementuje. Ak je hodnota semafóra > 0 dekrementuje sa a funkcia končí návratom

  post – inkrementuje hodnotu semafóra o 1. Ak bola hodnota predtým 0 a vlákna boli blokované na wait operácii jedno z nich bude odblokované (wait operácia sa skončí a semafór sa dekrementuje opäť na nulu)

Semafóry (GNU/Linux)

- Semafór sa deklaruje ako premenná typu sem_t
- Inicializuje sa funkciou sem_init(sem_t*,0, int), kde
  prvý argument je smerník na premennú semafóra
  druhý argument musí byť pre GNU/Linux 0
  tretí argument je typu int a je počiatočnou hodnotou semafóra.
- Semafór sa zruší funkciou sem_destroy(sem_t*).
- Funkcia sem_wait(sem_t*) je wait operácia.
- Funkcia sem_post(sem_t*) je post operácia.
- Funkcia sem_trywait(sem_t*) miesto blokovania vracia chybovú hodnotu EAGAIN.
- Funkcia sem_getvalue(sem_t*, int*) vloží do svojho druhého argumentu aktuálnu hodnotu semafóra.

Príklad semafóra

Job Queue Controlled by a Semaphore

```
#include <malloc.h>
#include <pthread.h>
#include <semaphore.h>
struct job {
  /* Link field for linked list. */
  struct job* next;
  /* Other fields describing work to be done... */
};
/* A linked list of pending jobs. */
struct job* job_queue;
/* A mutex protecting job_queue. */
pthread_mutex_t job_queue_mutex =
    PTHREAD_MUTEX_INITIALIZER
```

```
/* A semaphore counting the number of jobs in the queue.*/
sem_t job_queue_count;
/* Perform one-time initialization of the job queue. */
void initialize_job_queue ()
{
  /* The queue is initially empty. */
  job_queue = NULL;
  /* Initialize the semaphore which counts jobs in the queue.
     Its initial value should be zero. */
  sem_init (&job_queue_count, 0, 0);
}
```

```
/* Process queued jobs until the queue is empty. */
void* thread_function (void* arg)
{
  while (1) {
    struct job* next_job;
    /* Wait on the job queue semaphore. If its value is positive,
    indicating that the queue is not empty, decrement the count by 1. If
    the queue is empty, block until a new job is enqueued. */
    sem_wait (&job_queue_count);
    /* Lock the mutex on the job queue. */
    pthread_mutex_lock (&job_queue_mutex);
    /* Because of the semaphore, we know the queue is not empty. Get
    the next available job. */
    next_job = job_queue;
```

```
    /* Remove this job from the list. */
    job_queue = job_queue->next;
    /* Unlock the mutex on the job queue because we're done with the
    queue for now. */
    pthread_mutex_unlock (&job_queue_mutex);
    /* Carry out the work. */
    process_job (next_job);
    /* Clean up. */
    free (next_job);
  }
  return NULL;
}
```

```
/* Add a new job to the front of the job queue. */
void enqueue_job (/* Pass job-specific data here... */)
{
  struct job* new_job;
  /* Allocate a new job object. */
  new_job = (struct job*) malloc (sizeof (struct job));
  /* Set the other fields of the job struct here... */
  /* Lock the mutex on the job queue before accessing it. */
  pthread_mutex_lock (&job_queue_mutex);
  /* Place the new job at the head of the queue. */
  new_job->next = job_queue;
  job_queue = new_job;
```

```
  /* Post to the semaphore to indicate that another job is
     available. If threads are blocked, waiting on the
     semaphore, one will become unblocked so it can
     process the job. */
  sem_post (&job_queue_count);
  /* Unlock the job queue mutex. */
  pthread_mutex_unlock (&job_queue_mutex);
}
```

Komentár k programu

- Pred tým než sa z fronty vyberie úloha musí každé vlákno čakať na semafóre; ak je jeho hodnota 0 (fronta je prázdna) vlákno je blokované pokiaľ je nie hodnota semafóra >0 (úloha bola vložená do fronty).
- Funkcia enqueue_job vkladá úlohy do fronty, ktorú musí najprv pred vložením úlohy uzamknuť a po vložení inkrementuje semafór a frontu odomkne.

Podmienkové premenné (PP)

- Používajú sa na blokovanie vlákien pokiaľ je nie špecifikovaná podmienka pravdivá
- Vždy sa používajú spolu so zámkou, pretože keď sa PP testuje musí byť uzamknutá.
- Keď je podmienka false vlákno je na PP blokované uvolní zámku a čaká na zmenu podmienky.
- Ak iné vlákno zmení podmienku, signalizuje túto zmenu, čakajúce vlákna sa zobudia, zamknú PP a znovu odtestujú podmienku.

Príklad

- Predpokladajme, že máme vlákno, ktoré vo svojej funkcii vykonáva nekonečný cyklus a v každej iterácii vykoná nejakú činnosť len ak je nastavený flag ako globálna premenná, ktorá je dostupná všetkým vláknam
- Vlákno musí v každej iterácii testovať flag – činné čakanie na jeho nastavenie (zbytočné mrhanie časom procesora)
- flag musí byť chránený zámkou

```
#include <pthread.h>
int thread_flag;
pthread_mutex_t thread_flag_mutex;
void initialize_flag (){
  pthread_mutex_init (&thread_flag_mutex, NULL);
  thread_flag = 0;
}
void* thread_function (void* thread_arg){
  while (1) {
  int flag_is_set;
  /* Protect the flag with a mutex lock. */
  pthread_mutex_lock (&thread_flag_mutex);
  flag_is_set = thread_flag;
  pthread_mutex_unlock (&thread_flag_mutex);
  if (flag_is_set)                    // busy waiting
  do_work ();
  /* Else don't do anything. Just loop again. */
  }
  return NULL;
}
```

```
/* Function sets the value of the thread flag to
   FLAG_VALUE. */
void set_thread_flag (int flag_value){
/* Protect the flag with a mutex lock. */
  pthread_mutex_lock (&thread_flag_mutex);
  thread_flag = flag_value;
  pthread_mutex_unlock (&thread_flag_mutex);
}
```

Efektívnejšie riešenie bez busy waiting

- Čo skutočne chceme je:

  - ak flag nie je nastavený uspať vlákno

  - po zmene flagu vlákno prebudiť aby mohlo vykonávať činnosť

- Riešením je použiť podmienkovú premennú

  - hodnota flagu bude podmienka, na ktorú vlákno vo funkcii vlákna čaká

  - vo funkcii set_thread_flag iné vlákno podmienku (flag) nastaví a signalizuje blokovanému vláknu, že sa podmienka zmenila

Podmienkové premenné

- Podmienková premenná je typu pthread_cond_t
- Podmienková premená umožňuje implementovať podmienku, ktorá umožní aby sa vlákno vykonávalo a opačne podmienku, na ktorej bude vlákno blokované
- Podmienka je zdieľaná všetkými vláknami a preto musí byť chránená zámkou a tá istá zámka musí byť použitá aj s podmienkovou premennou
- Podmienková premenná na rozdiel od semafórov nemá žiadne počítadlo alebo pamäť

Podmienkové premenné

- Nasledovné funkcie pracujú s podmienkovou premennou:

  int pthread_cond_init (pthread_cond_t*,NULL) - inicializačná funkcia PP

  int pthread_cond_signal (pthread_cond_t*) – vyšle signál vláknu blokovanému na PP, ktorý ho odblokuje. Ak nie je blokované žiadne vlákno je signál ignorovaný.

  int pthread_cond_broadcast (pthread_cond_t*) – odblokuje všetky vlákna, blokované na PP

  int pthread_cond_wait(pthread_cond_t*,pthread_mutex_t*) – blokuje volajúce vlákno na PP pokiaľ nepríde signál

  int pthread_cond_destroy(pthread_cond_t*) – zruší PP

Ako funguje pthread_cond_wait

- Druhým argumentom pthread_cond_wait() musí byť tá istá zámka, ktorá sa používa pri testovaní podmienky spojenej s PP

```
thread_function(){
…
pthread_mutex_lock(&mutex);
while(!flag)
pthread_cond_wait (&flag_cond, &mutex);
pthread_mutex_unlock(&mutex);
…
}
```

Ako funguje pthread_cond_wait

- Ak je podmienka false funkcia automaticky odomkne zámku a vlákno bude blokované pokiaľ nepríde signál o zmene podmienky
- Po signále bude vlákno volajúce pthread_cond_wait odblokované automaticky uzamkne zámku a opakovane otestuje podmienku
- Návrat z blokovania totiž automaticky neznamená, že sa podmienka zmenila a preto sa **odporúča** dať pthread_cond_wait do while slučky (po if condition sa podmienka už neotestuje)
- Ďalší dôvod prečo dať pthread_cond_wait do while slučky je ten, že vlákno môže poslať signál prv než iné naň čaká (signál sa potom stratí) a ak by tam while slučka nebola vlákno bude čakať na signál, ktorý už nemusí vzniknuť. Táto situácia nenastane ak vlákno najprv odtestuje podmienku či vôbec má na signál čakať.

Ako funguje pthread_cond_signal

- Vždy keď vlákno urobí inštrukciu, ktorá môže zmeniť podmienku je treba vykonať tieto kroky:
- Uzamknúť zámku pridruženú k PP aby malo k podmienke exkluzívny prístup
- Vykonať inštrukciu, ktorá zmení podmienku
- Vyslať signál PP, že nastala zmena podmienky
- Odomknúť zámku pridruženú k PP

```
thread_function(){
…
pthread_mutex_lock(&mutex);

  flag = !flag;
  pthread_cond_signal (&flag_cond);

pthread_mutex_unlock(&mutex);
…
}
```

Statická inicializácia PP

```
#include <pthread.h>
#include "errors.h"
typedef struct my_struct_tag {
    pthread_mutex_t    mutex;  /* Protects access to value */
    pthread_cond_t     cond;    /* Signals change to value */
    int  value;                          /* Access protected by mutex */
} my_struct_t;
my_struct_t data = {
    PTHREAD_MUTEX_INITIALIZER, PTHREAD_COND_INITIALIZER, 0};
int main (int argc, char *argv[])
{
    return 0;
}
```

Dynamická inicializácia PP

```
#include <pthread.h>
/*
 * Define a structure, with a mutex and condition variable.
 */
typedef struct my_struct_tag {
    pthread_mutex_t     mutex;  /* Protects access to value */
    pthread_cond_t      cond;    /* Signals change to value */
    int   value;              /* Access protected by mutex */
} my_struct_t;
```

```
int main (int argc, char *argv[]){
    my_struct_t *data;
    int status;
    data = malloc (sizeof (my_struct_t));
    status = pthread_mutex_init (&data->mutex, NULL);
    status = pthread_cond_init (&data->cond, NULL);
    status = pthread_cond_destroy (&data->cond);
    status = pthread_mutex_destroy (&data->mutex);
    (void)free (data);
    return status;
}
```

Podmienkové premenné - príklad

- V nasledovnom príklade sa PP používa na ochranu flagu (podmienka)
- V thread funkcii sa pred testovaním flagu vždy uzamkne zámka
- Zámka je automaticky odomknutá pred blokovaním vlákna na funkcii ..._wait a automaticky opäť uzamknutá po signále pre PP
- Zámka sa tiež používa vo funkcii, ktorá mení hodnoty flagu

Podmienkové premenné - príklad

Control a Thread Using a Condition Variable

```
#include <pthread.h>
int thread_flag;
pthread_cond_t thread_flag_cv;          /*condition variable*/
pthread_mutex_t thread_flag_mutex;    /*mutex*/
void initialize_flag (){                    /* initialization function*/
  /* Initialize the mutex and condition variable. */
  pthread_mutex_init (&thread_flag_mutex, NULL);
  pthread_cond_init (&thread_flag_cv, NULL);
  /* Initialize the flag value. */
  thread_flag = 0;
}
```

```
void* thread_function (void* thread_arg){
  while (1) {
     /* Lock the mutex before accessing the flag value. */
     pthread_mutex_lock (&thread_flag_mutex);
     while (!thread_flag)
     /* The flag is clear. Wait for a signal on the condition
    variable, indicating that the flag value has changed. When
    the signal arrives and this thread unblocks, loop and check
    the flag again. */
     pthread_cond_wait (&thread_flag_cv, &thread_flag_mutex);
    /* When we've gotten here, we know the flag must be set.
    Unlock the mutex. */
     pthread_mutex_unlock (&thread_flag_mutex);
    do_work ();
  }
  return NULL;
}
```

```
/* Sets the value of the thread flag to FLAG_VALUE. */
void set_thread_flag (int flag_value){
  /* Lock the mutex before accessing the flag value. */
  pthread_mutex_lock (&thread_flag_mutex);
  /* Set the flag value, and then signal in case
     thread_function is blocked, waiting for the flag to become
     set. */
    thread_flag = flag_value;
    pthread_cond_signal (&thread_flag_cv);

   /* Unlock the mutex. */
  pthread_mutex_unlock (&thread_flag_mutex);
}
```

Príklad 2

- V nasledovnom programe budú dva vlákna inkrementovať globálnu premennú count
- Program zaručuje, že ak hodnota count bude z daného intervalu bude count inkrementovať vlákno 2 - všetko ostatné je náhodné.

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

pthread_mutex_t condition_mutex =
PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t condition_cond =
PTHREAD_COND_INITIALIZER;

void *functionCount1();
void *functionCount2();
int count = 0;
#define COUNT_DONE 10
#define COUNT_ST1 3
#define COUNT_ST2 6
```

```
main()
{
pthread_t thread1, thread2;

pthread_create( &thread1, NULL, &functionCount1,
NULL);
pthread_create( &thread2, NULL, &functionCount2,
NULL);
pthread_join( thread1, NULL);
pthread_join( thread2, NULL);
exit(0);
}
```

```
// Write numbers 1-3 and 8-10 as permitted by  functionCount2()
void *functionCount1(){
  for(;;){
   // Lock mutex and then wait for signal to relase mutex
    pthread_mutex_lock( &condition_mutex );
    while( count >= COUNT_ST1 && count <=  COUNT_ST2 )
        // Wait while functionCount2() operates on count
        // mutex unlocked if condition variable in functionCount2() signaled.
      pthread_cond_wait( &condition_cond, &condition_mutex );
    count++;
    printf("Counter value functionCount1: %d\n",count);
    pthread_mutex_unlock( &condition_mutex );
    if(count >= COUNT_DONE)
       return (NULL);
  }
}
```

```
  // Write numbers 4-7
void *functionCount2(){
  for(;;){
    pthread_mutex_lock( &condition_mutex );
    if( count < COUNT_ST1 || count > COUNT_ST2 )
       // Signal to free waiting thread by freeing the mutex.
       // Note: functionCount1() is now permitted to modify "count".
        pthread_cond_signal( &condition_cond );
    else {
    count++;
    printf("Counter value functionCount2: %d\n",count);
    }
    pthread_mutex_unlock( &condition_mutex );
    if(count >= COUNT_DONE)
    return(NULL);
  }
}
```

Výstup

```
Counter value functionCount1: 1
Counter value functionCount1: 2
Counter value functionCount1: 3
Counter value functionCount2: 4
Counter value functionCount2: 5
Counter value functionCount2: 6
Counter value functionCount2: 7
Counter value functionCount1: 8
Counter value functionCount1: 9
Counter value functionCount1: 10
Counter value functionCount2: 11
```

Príklad 3

- Nasledovný príklad demonštruje riadený výber a vkladanie úloh z a do fronty úloh vláknami s použitím podmienkovej premennej

Príklad 3

Job Queue Controlled by a condition variable

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

struct job {
  /* Link field for linked list. */
  struct job* next;
};
/* A linked list of pending jobs. */
struct job* job_queue;
/* A mutex protecting job_queue. */
pthread_mutex_t job_queue_mutex
 = PTHREAD_MUTEX_INITIALIZER
```

```
/* A count counting the number of jobs in the queue.*/
int count;
pthread_cond_t cond= PTHREAD_COND_INITIALIZER;
/* Perform one-time initialization of the job queue. */
void initialize_job_queue ()
{
  /* The queue is initially empty. */
  job_queue = NULL;
  /* Initialize the count which counts jobs in the queue. Its
     initial value should be zero. */
  count = 0;
}
```

```
/* Process queued jobs until the queue is empty. */
void* thread_function (void* arg)
{
  while (1) {
    struct job* next_job;
    pthread_mutex_lock (&job_queue_mutex);
    while(count = 0)
      pthread_cond_wait(&cond, &job_queue_mutex);

/* Because of the count value, we know the queue is
   not empty. Get the next available job. */
     next_job = job_queue;
```

```
    /* Remove this job from the list. */
      job_queue = job_queue->next;
      count--;
      pthread_mutex_unlock (&job_queue_mutex);
     /* Carry out the work. */
     process_job (next_job);
     /* Clean up. */
     free (next_job);
   }
   return NULL;
}
```

```
/* Add a new job to the front of the job queue. */
void enqueue_job (/* Pass job-specific data here... */)
{
  struct job* new_job;
  /* Allocate a new job object. */
  new_job = (struct job*) malloc (sizeof (struct job));
  /* Set the other fields of the job struct here... */

  /* Lock the mutex on the job queue before accessing it. */
  pthread_mutex_lock (&job_queue_mutex);
  /* Place the new job at the head of the queue. */
  new_job->next = job_queue;
  job_queue = new_job;
```

```
  if (count = 0){
    count++;
    pthread_cond_signal(&cond);
  }
  else
  count++;
  /* Unlock the mutex on the job queue */
  pthread_mutex_unlock (&job_queue_mutex);
}
```

Komentár k programu

- Pred tým než sa z fronty vyberie úloha musí každé vlákno testovať premennú count ak je nula vlákno je na podm. premennej blokované (fronta je prázdna) pokiaľ nedostane cond_signal že úloha bola vložená do fronty iným vláknom.
- Funkcia vlákna enqueue_job(), ktorá úlohy vkladá do fronty, musí po vložení úlohy zaslať cond_signal vláknam len v prípade ak naň čakajú.
- Prístup k premennej count musí byť chránený zámkou