

# IB109 Návrh a implementace paralelních systémů

## POSIX Threads – pokračování

Jiří Barnat

## **Správa vláken**

- Vytváření, oddělování a spojování vláken
- Funkce na nastavení a zjištění stavu vlákna

## **Vzájemná vyloučení (mutexes)**

- Vytváření, ničení, zamykání a odemykání mutexů
- Funkce na nastavení a zjištění atributů spojených s mutexy

## **Podmínkové/podmíněné proměnné (conditional variable)**

- Slouží pro komunikaci/synchronizaci vláken
- Funkce na vytváření, ničení, “čekání na” a “signalizování při” specifické hodnotě podmínkové proměnné
- Funkce na nastavení a zjištění atributů proměnných

## Motivace

- Víceru vláken provádí následující kód  
`if (my_cost < best_cost) best_cost = my_cost;`
- Nedeterministický výsledek pro 2 vlákna a hodnoty:  
`best_cost==100, my_cost@1==50, my_cost@2==75`

## Řešení

- Umístění kódu do kritické sekce
- `pthread_mutex_t`

## Inicializace mutexu

```
int pthread_mutex_init (  
    pthread_mutex_t *mutex_lock,  
    pthread_mutexattr_t *attribute)
```

- Parametr `attribute` specifikuje vlastnosti zámku
- `NULL` znamená výchozí (přednastavené) nastavení

```
int pthread_mutex_lock (pthread_mutex_t *mutex_lock)
```

- Volání této funkce zamyká `mutex_lock`
- Volání je blokující, dokud se nepodaří mutex zamknout
- Zamknout mutex se podaří pouze jednou jednomu vláknou
- Vláknou, kterému se podaří mutex zamknout je v kritické sekci
- Při opuštění kritické sekce, je vláknou "povinné" mutex odemknout
- Teprve po odemknutí je možné mutex znovu zamknout
- Kód provedený v rámci kritické sekce je po odemčení mutexu globálně viditelný (paměťová bariéra)

```
int pthread_mutex_unlock (pthread_mutex_t *mutex_lock)
```

- Odemyká `mutex_lock`

## Pozorování

- Velké kritické sekce snižují výkon aplikace
- Příliš času se tráví v blokujícím volání funkce `pthread_mutex_lock`

```
int pthread_mutex_trylock (pthread_mutex_t *mutex_lock)
```

- Pokusí se zamknout mutex
- V případě úspěchu vrátí 0
- V případě neúspěchu EBUSY
- Smysluplné využití komplikuje návrh programu
- Implementace bývá rychlejší než `pthread_mutex_lock` (nemusí se manipulovat s frontami čekajících procesů)
- Má smysl aktivně čekat opakovaným volání `trylock`

# Vlastnosti (atributy) mutexů

## Normální mutex

- Pouze jedno vlákno může jedenkrát zamknout mutex
- Pokud se vlákno, které má zamčený mutex, pokusí znovu zamknout stejný mutex, dojde k uváznutí

## Rekurzivní mutex

- Dovoluje jednomu vláknu zamknout mutex opakovaně
- K mutexu je asociován čítač zamknutí
- Nenulový čítač značí zamknutý mutex
- Pro odemknutí je nutno zavolat `unlock` tolikrát, kolikrát bylo voláno `lock`

## Normální mutex s kontrolou chyby

- Chová se jako normální mutex, akorát při pokusu o druhé zamknutí ohlásí chybu
- Pomalejší, typicky používán dočasně po dobu vývoje aplikace, pak nahrazen normálním mutexem

```
int pthread_mutexattr_settype_np (  
    pthread_mutexattr_t *attribute,  
    int type)
```

- Nastavení typu mutexu
- Typ určen hodnotou proměnné `type`
- Hodnota `type` může být
  - `PTHREAD_MUTEX_NORMAL_NP`
  - `PTHREAD_MUTEX_RECURSIVE_NP`
  - `PTHREAD_MUTEX_ERRORCHECK_NP`

## Motivace

- Často na jednu kritickou sekci čeká vícero vláken
- Aktivní čekání – permanentní zátěž CPU
- Uspávání s timeoutem – netriviální režie, omezená frekvence dotazování se na možnost vstupu do kritické sekce

## Řešení

- Uspání vlákna, pokud vlákno musí čekat
- Vzbuzení vlákna v okamžiku, kdy je možné pokračovat

## Realizace v POSIX Threads

- Mechanismus označovaný jako **Podmínkové proměnné**
- Podmínková proměnná vyžaduje použití mutexu
- Po získání mutexu se vlákno může dočasně vzdát tohoto mutexu a uspat se (v rámci dané podmínkové proměnné)
- Probuzení musí být explicitně signalizováno jiným vláknem

```
int pthread_cond_init (  
    pthread_cond_t *cond,  
    pthread_cond_attr_t *attr)
```

- Inicializuje podmínkovou proměnnou
- Má-li `attr` hodnotu `NULL`, podmínková proměnná má výchozí chování

```
int pthread_cond_destroy (  
    pthread_cond_t *cond)
```

- Zničí nepoužívanou podmínkovou proměnnou a související datové struktury

```
int pthread_cond_wait (  
    pthread_cond_t *cond,  
    pthread_mutex_t *mutex_lock)
```

- Uvolní mutex `mutex_lock` a zablokuje vlákno ve spojení s podmínkovou proměnou `cond`
- Po návratu vlákno opět vlastní mutex `mutex_lock`
- Před použitím musí být `mutex_lock` inicializován a zamčen volajícím vláknem

```
int pthread_cond_signal (  
    pthread_cond_t *cond)
```

- Signalizuje probuzení jednoho z vláken, uspaných nad podmínkovou proměnou `cond`

```
int pthread_cond_broadcast (  
    pthread_cond_t *cond)
```

- Signalizuje probuzení všem vláknům čekajících nad podmínkovou proměnnou `cond`

```
int pthread_cond_timedwait (  
    pthread_cond_t *cond,  
    pthread_mutex_t *mutex_lock,  
    const struct timespec *abstime)
```

- Vlákno buď vzbuzeno signálem, nebo vzbuzeno po uplynutí času specifikovaném v `abstime`
- Při vzbuzení z důvodu uplynutí času, vrací chybu `ETIMEDOUT`, a neimplikuje znovu získání `mutex_lock`

## Podmínkové proměnné – typické použití

```
12  pthread_cond_t is_empty;
13  pthread_mutex_t mutex;

432 pthread_mutex_lock(&mutex);
433 while ( size > 0 )
434     pthread_cond_wait(&is_empty,&mutex);
    ...
456 pthread_mutex_unlock(&mutex);

715 [pthread_mutex_lock(&mutex);]
    ...
721 size=0;
722 pthread_cond_signal(&is_empty);
723 [pthread_mutex_unlock(&mutex);]
```

## Problém

- Vzhledem k požadavkům vytváření reentrantních a thread-safe funkcí se programátorům zakazuje používat globální data.
- Případné použití globálních proměnných musí být bezstavové a prováděno v kritické sekci.
- Klade omezení na programátory.

## Řešení

- Thread specific data (TSD)
- Globální proměnné, které mohou mít pro každé vlákno jinou hodnotu.

## Standardní řešení

- Pole indexované jednoznačným identifikátorem vlákna.
- Vlákna musí mít rozumně velké identifikátory.
- Snadný přístup k datům patřící jiným vláknům – potenciální riziko nekorektního kódu.

## Řešení POSIX standardu

- Identifikátor (klíč) a asociovaná hodnota.
- Identifikátor je globální, asociovaná hodnota lokální proměnná.
- Klíč – `pthread_key_t`.
- Asociovaná hodnota – univerzální ukazatel, tj. `void *`.

```
int pthread_key_create (  
    pthread_key_t *key,  
    void (*destructor)(void*))
```

- Vytvoří nový klíč (jedná se o globální proměnnou).
- Hodnota asociovaného ukazatele je nastavena na NULL pro všechna vlákna.
- Parametr destructor – funkce, která bude nad asociovanou hodnotou vlákna volána v okamžiku ukončení vlákna, pokud bude asociovaná hodnota nenulový ukazatel.
- Parametr destructor je nepovinný, lze nahradit NULL.

## Zničení klíče a asociovaných ukazatelů

- `int pthread_key_delete (pthread_key_t key)`
- Nevolá žádné destructor funkce.
- Programátor je zodpovědný za dealokaci objektů před zničením klíče.

## Funkce na získání a nastavení hodnoty asociovaného ukazatele

- `void * pthread_getspecific (pthread_key_t key)`
- `int pthread_setspecific (pthread_key_t key, const void *value)`

```
pthread_t pthread_self ()
```

- Vrací unikátní systémový identifikátor vlákna

```
int pthread_equal (pthread_t thread1,  
                  pthread_t thread2)
```

- Vrací nenula při identitě vláken thread1 a thread2

```
pthread_once_t once_control = PTHREAD_ONCE_INIT;  
int pthread_once(pthread_once_t *once_control,  
                 void (*init_routine)(void));
```

- První volání této funkce z jakéhokoliv vlákna způsobí provedení kódu `init_routine`. Další volání nemají žádný efekt.

## Plánování (scheduling) vláken

- Není definováno, většinou je výchozí politika dostačující.
- POSIX Threads poskytují funkce na definici vlastní politiky a priorit vláken.
- Není požadována implementace této části API.

## Správa priorit mutexů

## Sdílení podmínkových proměnných mezi procesy

## Vlákna a obsluha POSIX signálů

## Typické konstrukce

## Specifikace problému

- Vlákna aplikace často čtou hodnotu, která je relativně méně často modifikována. (Write-Rarely-Read-Many)
- Je žádoucí, aby čtení hodnoty mohlo probíhat souběžně.

## Možné problémy

- Souběžný přístup dvou vláken-písařů, může vyústit v nekonzistentní data nebo mít nežádoucí vedlejší efekt, například memory leak.
- Souběžný přístup vlákna-písaře v okamžiku čtení hodnoty jiným vláknem-čtenářem může vyústit v čtení neplatných, nekonzistentních dat.

## Řešení s použitím POSIX Threads

- Čtení a modifikace dat bude probíhat v kritické sekci.
- Přístup do kritické sekce bude řízen s pomocí funkcí `pthread_*`.

## Další požadavky

- Vlákno-čtenář může vstoupit do kritické sekce, pokud v ní není nebo na ní nečeká žádné vlákno-písař.
- Vlákno-čtenář může vstoupit do kritické sekce, pokud v ní jsou jiná vlákna-čtenáři.
- Přístupy vláken-písařů jsou serializovány a mají přednost před přístupy vláken-čtenářů.

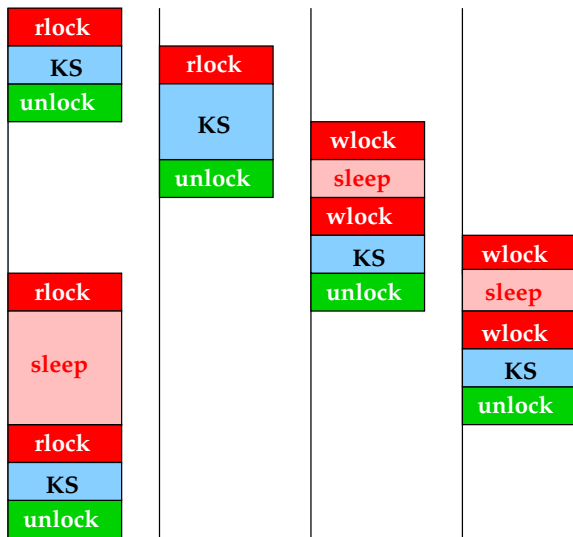
## Jednoduché řešení

- Použít jeden `pthread_mutex_t` pro kritickou sekci.
- Vylučuje souběžný přístup vláken-čtenářů.

## Lepší řešení

- Nový typ zámku – `rwlock_t`
- Funkce pracující s novým zámkem
  - `rwlock_rlock(rwlock_t *l)` – vstup vlákna-čtenáře
  - `rwlock_wlock(rwlock_t *l)` – vstup vlákna-písaře
  - `rwlock_unlock(rwlock_t *l)` – opuštění libovolným vláknem
- Funkce `rwlock` implementovány s pomocí POSIX Thread API

# Čtenáři a písaři – Implementace



```
1  typedef struct {
2      int readers;
3      int writer;
4      pthread_cond_t readers_proceed;
5      pthread_cond_t writer_proceed;
6      int pending_writers;
7      pthread_mutex_t lock;
8  } rwlock_t;
9
10 void rwlock_init (rwlock_t *l) {
11     l->readers = l->writer = l->pending_writers = 0;
12     pthread_mutex_init(&(l->lock), NULL);
13     pthread_cond_init(&(l->readers_proceed), NULL);
14     pthread_cond_init(&(l->writer_proceed), NULL);
15 }
```

```
16
17     void rwlock_rlock (rwlock_t *l) {
18         pthread_mutex_lock(&(l->lock));
19         while (l->pending_writers>0 || (l->writer>0)) {
20             pthread_cond_wait(&(l->readers_proceed), &(l->lock));
21         }
22         l->readers++;
23         pthread_mutex_unlock(&(l->lock));
24     }
25
```

```
26
27     void rwlock_wlock (rwlock_t *l) {
28         pthread_mutex_lock(&(l->lock));
29         while (l->writer>0 || (l->readers>0)) {
30             l->pending_writers++;
31             pthread_cond_wait(&(l->writer_proceed), &(l->lock));
32             l->pending_writers --;
33         }
34         l->writer++;
35         pthread_mutex_unlock(&(l->lock));
36     }
37
```

```
38
39 void rwlock_unlock (rwlock_t *l) {
40     pthread_mutex_lock(&(l->lock));
41     if (l->writer>0)
42         l->writer=0;
43     else if (l->readers>0)
44         l->readers--;
45     pthread_mutex_unlock(&(l->lock));
46     if ( l->readers == 0 && l->pending_writers >0)
47         pthread_cond_signal( &(l->writer_proceed) );
48     else if (l->readers>0)
49         pthread_cond_broadcast( &(l->readers_proceed) )
50 }
51
```

- Počítání minima

```
2122    ...
2123    rwlock_rlock(&rw_lock);
2124    if (my_min < global_min) {
2125        rwlock_unlock(&rw_lock);
2126        rwlock_wlock(&rw_lock);
2127        if (my_min < global_min) {
2128            global_min = my_min;
2129        }
2130    }
2131    rwlock_unlock(&rw_lock);
2132    ...
```

- Hašovací tabulky

- ...

## Specifikace problému

- Synchronizační primitivum
- Vláknu je dovoleno pokračovat po *bariéře* až když ostatní vlákna dosáhly bariéry.
- Naivní implementace přes mutexy vyžaduje aktivní čekání (nemusí být vždy efektivní).

## Lepší řešení

- Implementace bariéry s použitím podmínkové proměnné a počítadla.
- Každé vlákno, které dosáhlo bariéry zvýší počítadlo.
- Pokud není dosaženo počtu vláken, podmíněné čekání.

```
1  typedef struct {
2      pthread_mutex_t count_lock;
3      pthread_cond_t ok_to_proceed;
4      int count;
5  } barrier_t;
6
7  void barrier_init (barrier_t *b) {
8      b->count = 0;
9      pthread_mutex_init(&(b->count_lock), NULL);
10     pthread_cond_init(&(b->ok_to_proceed), NULL);
11 }
```

```
12 void barrier (barrier_t *b, int nr_threads) {
13     pthread_mutex_lock(&(b->count_lock));
14     b->count ++;
15     if (b->count == nr_threads) {
16         b->count = 0;
17         pthread_cond_broadcast(&(b->ok_to_proceed));
18     }
19     else
20         while (pthread_cond_wait(&(b->ok_to_proceed),
21             &(b->count_lock)) != 0);
22     pthread_mutex_unlock(&(b->count_lock));
23 }
```

## Problém

- Po dosažení bariéry všemi vlákny, je mutex `count_lock` postupně zamčen pro všechny vlákna
- Dolní odhad na dobu běhu bariéry je tedy  $O(n)$ , kde  $n$  je počet vláken participujících na bariéře

## Možné řešení

- Implementace bariéry metodou binárního půlení
- Teoretický dolní odhad na bariéru je  $O(n/p + \log p)$ , kde  $p$  je počet procesorů

## Cvičení

- Implementujte bariéru využívající binárního půlení
- Měřte dopad počtu participujících vláken na dobu trvání lineární a logaritmické bariéry na vámi zvoleném paralelním systému

## Typické chyby – situace 1

- Vlákno V1 vytváří vlákno V2
- V2 požaduje data od V1
- V1 plní data až po vytvoření V2
- V2 použije neinicializovaná data

## Typické chyby – situace 2

- Vlákno V1 vytváří vlákno V2
- V1 předá V2 pointer na lokální data V1
- V2 přistupuje k datům asynchronně
- V2 použije data, která už neexistují (V1 skončilo)

## Typické chyby – situace 3

- V1 má vyšší prioritu než V2, čtou stejná data
- Není garantováno, že V1 přistupuje k datům před V2
- Pokud V2 má destruktivní čtení, V1 použije neplatné data

## Valgrind

- Simulace běhu programu.
- Analýza jednoho běhu programu.

## Nástroje valgrindu

- Memcheck – detekce nekonzistentního použití paměti.
- Callgrind – jednoduchý profiler.
  - kcachegrind – vizualizace.
- Helgrind – detekce nezamykaných přístupů ke sdíleným proměnným v POSIX Thread programech.

## Další způsoby synchronizace

## **Problém – jak synchronizovat procesy**

- Mutexy z POSIX Threads dle standardu slouží pouze pro synchronizaci vláken v rámci procesu.
- Pro realizaci kritických sekcí v různých procesech je třeba jiných synchronizačních primitiv.
- Podpora ze strany operačního systému.

## **Semaforey**

- Čítače používané ke kontrole přístupů ke sdíleným zdrojům.
- Standardy: POSIX, System V
- Lze využít i pro synchronizaci vláken.

## Semafor

- Celočíselný nezáporný čítač jehož hodnota indikuje “obsazenost” sdíleného zdroje.
  - Nula – zdroj je využíván a není k dispozici.
  - Nenula – zdroj není využíván, je k dispozici.
- `sem_init()` – inicializuje čítač zadanou výchozí hodnotou
- `sem_wait()` – sníží čítač, pokud může, a skončí, jinak blokuje
- `sem_post()` – zvýší čítač o 1, případně vzbudí čekající vlákno

## Semaforey vs. mutexy

- Mutex zamyká a odemyká totéž vlákno.
- Semafor může být spravován / manipulován různými vlákny.

## Monitor

- Synchronizační primitivum vyššího programovacího jazyka.
- Označení kódu, který může být souběžně prováděn nejvýše jedním vláknem.
- JAVA – klíčové slovo `synchronized`

## Semaforey, mutexy a monitory

- Se semaforey a mutexy je explicitně manipulováno programátorem.
- Vzájemné vyloučení realizované monitorem je implicitní, tj. explicitní zamykání skrze explicitní primitiva doplní překladač.