

IB109 Návrh a implementace paralelních systémů

Programování v prostředí se sdílenou pamětí

Jiří Barnat

Rizika spojená se sdílenou pamětí

Pozorování

- Paralelní programy mohou při opakovaném spuštění zdánlivě náhodně vykazovat různá chování.
- Výsledek provedení programu může záviset na absolutním pořadí provedení instrukcí programu, tj. na proložení instrukcí zúčastněných procesů/vláken.

Race condition

- Nedokonalost paralelního programu, která se projevuje takovýmto nedeterministickým chováním se označuje jako **race condition**, (zkráceně race).

Příklad

```
*myStructure p;
```

```
P0 {  
    p = new myStructure;  
    p -> data = 1;  
    cout <<(p->data)<<endl;  
}
```

```
P1 {  
    p = new myStructure;  
    p -> data = 2;  
    cout <<(p->data)<<endl;  
}
```

Pozorování

- Jednoduchý příkaz ve vyšším programovacím jazyce neodpovídá nutně jedné instrukci procesoru.
- V moderních operačních systémech je každé vlákno podrobena plánovacímu procesu.
- Vykonání posloupnosti instrukcí procesoru odpovídající jednomu příkazu vyššího programovacího jazyka může být přerušeno a proloženo vykonáním instrukcí jiného vlákna.

Příklad

- Přičtení čísla do proměnné efektivně může znamenat načtení proměnné do registru, provedení aritmetické operace, a uložení výsledku do paměti.
- Při vhodném souběhu následujících procesů, se může efekt jednoho přiřazení do sdílené globální proměnné zcela vytratit

```
volatile int a=0;
```

```
P0 {
```

```
    a = a + 10;
```

```
}
```

```
P1 {
```

```
    a = a + 20;
```

```
}
```

- Demonstrujte proložení instrukcí, které vyústí v jinou hodnotu, než 30.

Pozorování

- Nelze spoléhat na současný souběh vláken, potažmo relativní rychlost výpočtu jednotlivých vláken.

Příklad

```
volatile int a=0;
```

```
P0 {  
    usleep 200;  
    a = 0;  
}
```

```
P1 {  
    a = 1;  
    usleep 200;  
}
```

- Po skončení obou vláken (současně spuštěných) bude mít sdílená proměnná ve většině případů hodnotu 0. Není to však ničím garantováno, tj. může nastat situace, kdy bude mít hodnotu 1.

Uváznutí (Deadlock)

- Pokud mají vlákna inkrementální požadavky na unikátní sdílené zdroje, může dojít k tzv. uváznutí, tj. nemožnosti pokračování ve výpočtu.

Příklad

```
P0 {
    zamkni A;
    zamkni B;
    ...
}
P1 {
    zamkni B;
    zamkni A;
    ...
}
```


Hladovění, Stárnutí, Neprogrese (Livelock)

- Jev, kdy alespoň jedno vlákno není schopné vzhledem k paralelnímu souběhu s jiným vláknem pokročit ve výpočtu za danou hranici.

Příklad

- ```
volatile int a=0;
P0 {
 while (true) {;
 a++; a-;
 }...
}
P1 {
 while (a == 0) { };
 ...;
}
```

- Vlákno P1 může na vyznačeném řádku strávit mnoho času.

## Thread-safe procedura

- Označení procedury či programu, jejíž kód je bezpečné provádět (vzhledem k sémantice výstupu a stabilitě výpočtu) souběžně několika vlákný bez nutnosti vzájemné domluvy/synchronizace.
- Knihovní funkce nemusí být thread-safe!
  - `rand()` -> `rand_r()`

## Re-entrantní procedura

- Procedura, jejíž provádění může být v rámci jednoho vlákna přerušeno, kód kompletně vykonán od začátku do konce v rámci téže úlohy, a poté obnoveno/dokončeno přerušené vykonávání kódu.
- Termín pochází z dob, kdy nebyly multitaskingové operační systémy.

## Neporovnatelné

- Re-entrantní procedura nemusí být thread-safe.  
viz [http://en.wikipedia.org/wiki/Reentrancy\\_\(computing\)](http://en.wikipedia.org/wiki/Reentrancy_(computing))
- Thread-safe procedura nemusí být re-entrantní.  
(Problémem je například používání globálních zámků.)

## Příklad

- Thread-safe procedura, která není re-entrantní:

```
WC {
 je-li odemčeno, vejdi a zamkni, jinak čekej
 ...
 odemkni a opusť onu místnost
}
```

## Nebezpečné akce vzhledem k paralelnímu zpracování

- Nekontrolovaný přístup ke globálním proměnným a haldě.
- Uchovávání stavu procedury do globálních proměnných.
- Alokace, dealokace zdrojů globálního rozsahu (soubory, ...).
- Nepřímý přístup k datům skrze odkazy nebo ukazatele.
- Viditelný vedlejší efekt (modifikace nestálých proměnných).

## Bezpečná strategie

- Přístup pouze k lokálním proměnným (zásobník).
- Kód je závislý pouze na argumentech dané funkce.
- Veškeré volané podprocedury a funkce jsou thread-safe.

```
*myStructure p;
```

```
*myStructure function() {
 p=new myStructure;
 return p;
}
```

```
P0 {
 *myStructure x;
 x=function();
}
```

```
P1 {
 *myStructure x;
 x=function();
}
```

## Pozorování

- Přístup ke sdíleným proměnným je „kořenem všeho zla“.
- Veškeré modifikace a neatomická čtení globálních proměnných musí být **serializovány**.

## Kritická sekce

- Část kódu, jehož provedení je neproložitelné instrukcemi jiného vlákna.
- Realizace kritické sekce musí být odolná vůči plánování.

## Zamykání

- Vlákno vstupující do volné kritické sekce, svým vstupem znemožní přístup ostatním vláknům (sekcí tzv. zamkne).
- Ostatní vlákna čekají před vstupem do kritické sekce.
- Při odchodu z kritické sekce, je zámek uvolněn, získá ho **náhodně** jedno z čekajících vláken.

## Jednoduché řešení zámku

- Sdílená atomicky přístupovaná bitová proměnná, jejíž hodnota indikuje přítomnost procesu/vlákná v přidružené kritické sekci.
- Manipulována při vstupu a výstupu z/do kritické sekce.
- Vyžaduje podporu HW pro atomickou manipulaci.

## Aktivní čekání – spinlock

- Dokud neuspěje, vlákno opakovaně zkouší vstoupit do kritické sekce (tj. po tuto dobu neustále provádí tentýž kód).

## Uspávání

- Procesy/vlákná se po neúspěchu vstoupit do kritické sekce sami vzdají procesorového kvanta (uspí se).
- Jsou buzeny buď po vypršení časového limitu nebo explicitně jiným běžícím vláknem.

## Rizika

- Uvážnutí, stárnutí, snížení výkonnosti.

## Přístup ke sdíleným globálním proměnným

- Manipulace se zámkem vynucuje vylití cache paměti.
- Mnoho přístupů k zamykaným proměnným může být úzkým místem výkonu aplikace, z principu nelze odstranit.

## Petersonův algoritmus (spinlock, user-space)

- Spravedlivý algoritmus pro řízení vzájemného vyloučení.
- Nezpůsobuje stárnutí ani uvážnutí.
- Vyžaduje atomické zápisy do proměnných.
- Citlivý na provádění instrukcí mimo pořadí.



# POSIX Thread API

## Historie

- SMP systémy
- Vlákna implementována jednotlivými výrobci HW
- IEEE POSIX 1003.1c standard

## IEEE POSIX 1003.1c

- Programátorský model semaforů a provádění operací v kritické sekci
- Rozhraní pro C
- POSIX threads, PThreads

## Jiné normy

- Operační systémy: NT Threads (Win32), Solaris threads, ...
- Programovací jazyky: Java threads, C++11 threading, ...

## **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

## Přes 60 API funkcí

- `#include <pthread.h>`
- Překlad s volbou `-pthread`

## Mnemotechnické předpony funkcí

- `pthread_`, `pthread_attr_`
- `pthread_mutex_`, `pthread_mutexattr_`
- `pthread_cond_`, `pthread_condattr_`
- `pthread_key_`

## Pracuje se skrytými objekty (Opaque objects)

- Objekty v paměti, o jejichž podobě programátor nic neví.
- Přístupovány výhradně pomocí odkazu (handle).
- Nedostupné objekty a neplatné (dangling) reference.

## Idea

- Vlastnosti všech vláken, mutexů i podmínkových proměnných nastavovány speciálními objekty.
- Některé vlastnosti entity musí být specifikovány již v době vzniku entity.

## Typy atributových objektů

- Vlákna: `pthread_attr_t`
- Mutexy: `pthread_mutexattr_t`
- Podmínkové proměnné: `pthread_condattr_t`

## Vznik a destrukce

- Funkce `_init` a `_destroy` s odpovídající předponou
- Parametr odkaz na odpovídající atributový objekt

## Vytváření vláken

- Každý program má jedno hlavní vlákno
- Další vlákna musí být explicitně vytvořena programem
- Každé vlákno (i vytvořené) může dále vytvářet další vlákna
- Vlákno vytvářeno funkcí `pthread_create`
- Vytvářené vlákno je ihned připraveno k provádění
- Může být plánovačem spuštěno dříve, než se dokončí volání vytvářecí funkce
- Veškerá data potřebná při spuštění vlákna, musí být připravena před voláním vytvářecí funkce
- Maximální počet vláken je závislý na implementaci

```
int pthread_create (
 pthread_t *thread_handle,
 const pthread_attr_t *attribute,
 void * (*thread_function)(void *),
 void *arg);
```

- `thread_handle` odkaz na vytvořené vlákno
- `attribute` odkaz na atributy vytvořeného vlákna (NULL pro přednastavené nastavení atributů)
- `thread_function` ukazatel na funkci nového vlákna
- `arg` ukazatel na parametry funkce `thread_function`
- Při úspěšném vytvoření vlákna vrací 0

## Ukončení vlákna nastává

- Voláním funkce `pthread_exit`
- Pokud skončí hlavní funkce rodičovského vlákna jinak než voláním `pthread_exit`
- Je-li zrušeno jiným vláknem pomocí `pthread_cancel`
- Rodičovský proces je ukončen (násilně nebo voláním `exit`)

`void pthread_exit (void *value)`

- Ukončuje běh vlákna
- Odkazy na prostředky procesu (soubory, IPC, mutexy, ...) otevřené v rámci vlákna se nezavírají
- Data patřící vlákně musí být uvolněna před ukončením vlákna (systém provede uvolnění prostředků až po skončení rodičovského procesu)
- Ukazatel `value` předán při spojení vláken

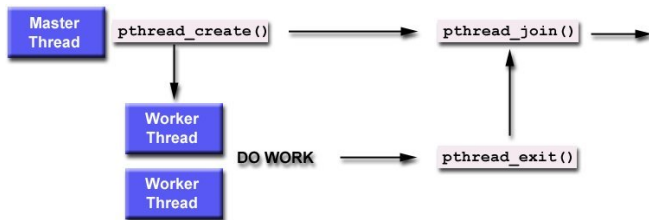


# Správa vláken – příklad

```
1 #include <pthread.h>
2 #include <stdio.h>
3 #define NUM_THREADS 5
4
5 void *PrintHello(void *threadid)
6 { printf("%d: Hello World!\n", threadid);
7 pthread_exit(NULL);
8 }
9
10 int main (int argc, char *argv[])
11 { pthread_t threads[NUM_THREADS];
12 for(int t=0; t<NUM_THREADS; t++)
13 pthread_create(&threads[t], NULL,
14 PrintHello, (void *)t);
15 pthread_exit(NULL);
16 }
```

```
int pthread_join (pthread_t thread_handle,
 void **ptr_value);
```

- Čeká na dokončení vlákna `thread_handle`
- Hodnota `ptr_value` je ukazatel na pointer specifikovaný vláknem `thread_handle` při volání `pthread_exit`
- Nutný například pokud `main` má vrátet smyslupnou návratovou hodnotu



## Nespojitelná vlákna (Detached threads)

- Nemohou být spojena voláním funkce `pthread_join`
- Šetří systémové prostředky
- Přednastavené nastavení typu vlákna není vždy zřejmé, proto je doporučeno typ vlákna explicitně nastavit

```
int pthread_detach (
 pthread_t *thread_handle)
```

```
int pthread_attr_setdetachstate(
 pthread_attr_t *attr,
 int detachstate)
```

```
int pthread_attr_getdetachstate(
 pthread_attr_t *attr,
 int *detachstate)
```

## Velikost zásobníku

- Minimální velikost zásobníku není určena
- Při velkém počtu vláken se často stává, že vyhrazené místo pro zásobník je vyčerpáno
- POSIX umožňuje zjistit a nastavit pozici a velikost místa vyhrazeného pro zásobník jednoho vlákna

```
int pthread_attr_getstacksize (
 pthread_attr_t *attribute, size_t *stacksize)
```

```
int pthread_attr_setstacksize (
 pthread_attr_t *attribute, size_t stacksize)
```

```
int pthread_attr_getstackaddr (
 pthread_attr_t *attribute, void **stackaddr)
```

```
int pthread_attr_setstackaddr (
 pthread_attr_t *attribute, void *stackaddr)
```

```
int pthread_cancel (
 pthread_t *thread_handle)
```

- Žádost o zrušení vlákna `thread_handle`
- Adresované vlákno se může a nemusí ukončit
- Vlákno může ukončit samo sebe
- Při zrušení se provádí úklid dat spojených s rušeným vláknem
- Funkce skončí po odeslání žádosti (je neblokující)
- Návrátový kód 0 značí, že adresované vlákno existuje, ne že bylo/bude zrušeno

## 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

# Vlastnosti (atributy) mutexů

```
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