Synchronizační nástroje Linuxu určené pro vícevláknové aplikace

2

V přechozím cvičení jsme si ukázali, jak lze s pomocí atomických operací implementovat vlastní zámky, které zajistí korektní synchronizaci vláken. V praxi bychom se k takovému řešení měli vyhnout, protože není efektivní a můžeme snadno vytvořit špatně odhalitelnou chybu. Čisté řešení je použít některý z nástrojů, který pro synchronizaci nabízí daný operační systém. V tomto cvičení si představíme nástroje pro synchronizaci vláken, které jsou k dispozici v Linuxu.

1 Tradiční zámky

Základní nástrojem, který knihovna `pthreads` nabízí je zámek. Ten je označovaný jako mutex.[1] Práce s mutexy je přímočará. Mutex je reprezentován datovým typem `pthread_mutex_t`. Mutex je možné inicializovat dvěma způsoby, buď pomocí funkce `pthread_mutex_init`:

```
int pthread_mutex_init(pthread_mutex_t *mutex, pthread_mutexattr_t *attr);
```

Použití je následovné:

```
pthread_mutex_t lock;
pthread_mutex_init(&lock, NULL); // NULL => vychozi vlastnosti
```

Případně můžeme použít inicializační makro `PTHREAD_MUTEX_INITIALIZER`, které inicializuje mutex s výchozími vlastnostmi, což je ve většině případů žádoucí.[2]

```
pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;
```

S mutexy se pracuje pomocí tří základních operací:

```
int pthread_mutex_lock(pthread_mutex_t *mutex);
int pthread_mutex_trylock(pthread_mutex_t *mutex);
int pthread_mutex_unlock(pthread_mutex_t *mutex);
```

Funkce `pthread_mutex_lock` provede uzamčení zámku, případně čeká, než bude zámek odemčen. Funkce `pthread_mutex_trylock` se pokusí uzamčít zámek stejně jako by to udělala předchozí funkce, avšak pokud je zámek držen jiným vláknem, nečeká a vrátí hodnotu `EBUSY`. Funkce `pthread_mutex_unlock` zámek odemče.

[1] Z anglického *mutual exclusion*, vzájemné vyloučení.

[2] Pomocí atributů zámku lze nastavit, zda zámek může být sdílen mezi procesy nebo jak se mají zámky chovat ve specifických případech, např. pokud je ukončeno vlákno držící daný zámek, pokud je zámek zamčen vícekrát jedním vláknem apod.

Po skončení práce se zámkem bychom se měli postarat o jeho zrušení.

```
int pthread_mutexattr_destroy(pthread_mutexattr_t *attr);
```

Úkol 1: Upravte zdrojové kódy z minulého cvičení tak, aby vedle námi vytvořených mechanismů zamykání šlo použít i mutexy z knihovny `pthreads`.

Poznámka: Při práci se synchronizačními nástroji bychom si měli dát pozor na nestandardní situace, např. ukončení vlákna držícího zámek, opakované zamykání apod. V řadě situací může dojít k nedefinovanému chování[3] nebo může dojít k uváznutí (deadlocku). Pokud chceme dělat cokoliv jiného než právě jednou zamčít a právě jednou odemčít zámek, je na místě pročíst podrobně dokumentaci.

2 Semafory

O něco univerzálnějším mechanismem pro synchronizaci jsou semafory. Práce s nimi je velice podobná práci se zámky. Máme datový typ `sem_t` reprezentující semafor, inicializační funkci `sem_init`, funkce pracující se semaforem, `sem_wait`, `sem_trywait` a `sem_post` a funkci pro uvolnění semaforu `sem_destroy`. Pozor, deklarace typu a prototypů funkcí se nachází v hlavičkovém souboru `samaphore.h`.

```
#include <semaphore.h>
int sem_init(sem_t *sem, int pshared, unsigned int value);
int sem_wait(sem_t *sem);
int sem_trywait(sem_t *sem);
int sem_post(sem_t *sem);
int sem_destroy(sem_t *sem);
```

Funkce `sem_init` akceptuje jako své argumenty počáteční hodnotu semaforu (argument `value`), případně je možné nastavit příznak, že bude semafor sdílen mezi procesy (argument `pshared`).[4] Pokud máme semafor pro synchronizaci vláken, použijeme hodnotu `0`. Funkce pojmenované `wait` snižují (popř. zkusí snížit) hodnotu semaforu, zamykají semafor, a odpovídají tedy operaci `P`. Funkce `sem_post` zvyšuje hodnotu semaforu, odemyká jej, a odpovídá operaci `V`.

Úkol 2: Rozšiřte úkol z minulého cvičení tak, aby k zamykání šlo použít binární semafor.

3 Bariéry

Vedle zamykání se při programování vícevláknových aplikací často setkáváme s požadavkem na to, aby se vlákna v určitém bodě zastavila do té doby, než ostatní vlákna dokončí svou činnost. Jeden takový scénař by mohl vypadat následovně:

(1) Vlákna odděleně provádí inicializaci výpočtu.

[3] Program může spadnout, být v nekonzistentním stavu, případně se to může projevit tak, že v našem testovacím prostředí vše bude fungovat správně, ale u uživatelů se objeví chyba.

[4] Toho je možné docílit s využitím sdílené paměti.

— Čeká se až všechna vlákna dokončí inicializaci.

(2) Každé vlákno provede výpočet se svým daty.

— Čeká se až všechna vlákna dokončí výpočet.

(3) Zpracují se výsledky ze všech vláken.

Pro tento typ úloh nabízí knihovna `pthreads` synchronizační nástroj označený jako *bariéra*. Tento objekt slouží k tomu, aby se vlákno programu zastavilo v určitém místě programu do doby, než daného místa v programu dosáhnou ostatní vlákna. Objekt bariéry je reprezentován typem `pthread_barrier_t` a k jeho inicializaci slouží funkce:

```
int pthread_barrier_init(pthread_barrier_t *barrier, pthread_barrierattr_t *attr,
                         unsigned count);
```

Tato funkce akceptuje jako své argumenty objekt bariéry, atributy[5] a počet vláken, které se mají na dané bariéře zastavit. K zastavení na bariéře slouží funkce:

```
int pthread_barrier_wait(pthread_barrier_t *barrier);
```

Tato funkce zastaví běh vlákna do doby, než celkem `count` vláken zavolá `pthread_barrier_wait`.

Ke zrušení bariéry slouží funkce:

```
int pthread_barrier_destroy(pthread_barrier_t *barrier);
```

Poznámka: Bariéry jsou při některých nastaveních překladače nedostupné. Abychom je měli k dispozici, je potřeba definovat standard, pro který je program vytvořený. Toho dosáhneme tím, že před direktivy `#include` vložíme odpovídající makro. V našem případě je potřeba zadat:

```
#define _POSIX_C_SOURCE 200112L /* nebo vyssi */
```

Úkol č. 3: Rozšiřte úkol z předchozího cvičení tak, aby se vlákna producentů při dosažení poloviny vygenerovaných čísel se synchronizovala. Tj., aby vlákno, které dosáhne poloviny vygenerováných čísel, počkalo, dokud druhé vlákno nevygeneruje taktéž polovinu čísel.

4 Podmínková proměnná

Běžně se stává, že potřebujeme uspat vlákno do doby, než bude splněna nějaká podmínka. Pro tyto účely nabízí knihovna `pthreads` objekt označovaný jako *podmínková proměnná*.[6] Což je objekt, který umožní uspat vlákno do doby, než je vyslán signál, že došlo ke změně a potenciálně byla daná podmínka splněna.

Podmínková proměnná hodnota je typu `pthread_cond_t`, která je inicializována a rušena podobně jako ostatní synchronizační nástroje.

[5] Pokud použijeme `NULL`, použije se výchozí nastavení.

[6] Anglicky *condition variable*.

```
int pthread_cond_init(pthread_cond_t *cond, pthread_condattr_t *attr);
pthread_cond_t cond = PTHREAD_COND_INITIALIZER;
int pthread_cond_destroy(pthread_cond_t *cond);
```

Práce s podmínkovou proměnnou je komplikovanější, protože vyžaduje kooperaci s mutexem, který chrání podmínku, na jejíž splnění čekáme. Jinými slovy, mutex chrání proměnnou na jejíž změnu čekáme. Pro práci podmínkovou proměnnou máme tři funkce.

```
int pthread_cond_wait(pthread_cond_t *cond, pthread_mutex_t *mutex);
int pthread_cond_signal(pthread_cond_t *cond);
int pthread_cond_broadcast(pthread_cond_t *cond);
```

Funkce `pthread_cond_wait` zastaví vlákno v daném bodě a odemče zámek `mutex`, který by měl být před zavoláním této funkce ve stavu zamčeno. Funkce `pthread_cond_signal`, probudí jedno z vláken, které na dané podmínkové proměnné čekají, a zamkne mutex, který byl předaný funkci `pthread_cond_wait`. Funkce `pthread_cond_broadcast` se chová obdobně, avšak probudí všechna vlákna čekající na dané podmínkové proměnné.

Scénář použití by měl vypadat přibližně následovně.

```
1  // globalni hodnota predstavujici nejakou podminku
2  volatile int podminka = 0;
3
4  // zamek chranici podminku
5  pthread_mutex_t lock = PTHTREAD_MUTEX_INITIALIZER;
6
7  // podminkova promenna
8  pthread_cont_t cond = PTHREAD_COND_INITIALIZER;
9
10 // vlakno A:
11 pthread_mutex_lock(&lock);
12 while (!podminka) {
13    pthread_cond_wait(&cond, &lock);
14 }
15 pthread_mutex_unlock(&lock);
16
17 // vlakno B:
18 pthread_mutex_lock(&lock);
19 podminka = 1;
20 pthread_mutex_unlock(&lock);
21 pthread_cond_signal(&cond);
```

Na řádcích 1 až 8 máme inicializaci. Zámek `lock` slouží k ochraně globální proměnné `podminka`.

Vlákno (A) na řádku 11 zamkne tento zámek a na řádku 12 otestujeme, zda je podmínka splněna. Pokud není, funkce `pthread_cond_wait` na řádku 13 uspí dané vlákna a odemče zámek `lock`. To je zásadní z toho důvodu, aby jiné vlákno mohlo změnit hodnotu proměnné `podminka`.

Vlákno (B) na řádcích 18 až 20 změní stav globální proměnné. Zavoláním funkce `pthread_cond_signal` na řádku 21 probudíme jedno vlákno, které čeká na podmínkové proměnné `cond`. V našem případě je to vlákno označené (A).

Vlákno (A) pokračuje řádkem 13, kdy dojde k probuzení vlákna a současně zamčení zámku `lock`, a můžeme otestovat, zda již byla podmínka splněna, či nikoliv. Pokud podmínka nebyla splněna, opět čekáme na řádku 13, jinak se smyčka ukončí a můžeme pokračovat dalším kódem.

Úkol č. 4: Ukázkový příklad se dvěma producenty a jedním konzumentem by šel vylepšit, protože vlákna neustále soupeří o zámek chránící frontu. A to i v případě, že je fronta plná. Řešení lze vylepšit tak, že pokud je fronta plná, vlákna producenta uspíme s pomocí podmínkové proměnné, a když se místo uvolní, probudíme jej zasláním signálu.

Poznámka: Úlohu zkuste vyřešit samostatně, případně můžete nahlédnout do přiloženého kódu.

5 Další nástroje

5.1 Spinlock

Implementace mutexů v Linuxu kombinuje aktivní a pasivní čekání, tj. na začátku proběhne pokus získat zámek pomocí spinlocku, a pokud vlákno neuspěje, je uspáno, což na jednu stranu šetří procesorový čas (pokud by mělo vlákno delší dobu čekat), ale zároveň má svou režii (přepnutí vláken). Kombinace aktivního a pasivního čekání dáva smysl pro spoustu běžných úloh. Mohou nastat situace, kdy ale tato kombinaci přináší horší výsledky, zejména, trváli zamčení krátkou dobu. V takovém případě se může hodit použít explicitně spinlock. Pro tyto případy knihovna `pthreads` nabízí implementaci spinlocku, která má podobné rozhraní jako mutexy.

Pro reprezentaci tohoto typu zámku máme datový typ `pthread_spinlock_t`, a pracujeme s nimi pomocí operací.

```
int pthread_spin_init(pthread_spinlock_t *lock, int pshared);
int pthread_spin_lock(pthread_spinlock_t *lock);
int pthread_spin_trylock(pthread_spinlock_t *lock);
int pthread_spin_unlock(pthread_spinlock_t *lock);
int pthread_spin_destroy(pthread_spinlock_t *lock);
```

Úkol č. 5: Rozšiřte úkol z minulého cvičení tak, aby k zamykání šlo použít spinlocky z knihovny `pthreads` a srovnejte jejich rychlost s předchozími řešeními.

5.2 Zámky pro čtení a zápis

Pro situace, kdy máme data, která mohou být v jeden okamžik čtena více vlákny, ale zápis je realizován v jeden okamžik právě jedním vláknem, má knihovna `pthreads` samostatný typ zámků `pthread_rwlock_t`.

Pracuje se s ním podobně jako s mutexy nebo spinlocky, máme inicializační funkci `pthread_rwlock_init`, funkci pro uvolnění zámku `pthread_rwlock_destroy` a funkce pracující se zámkem `pthread_rwlock_rdlock`, `pthread_rwlock_wrlock`, `pthread_rwlock_unlock`. Přirozeně musíme mít dva typy zamykacích funkcí, které se liší tím, jestli chceme zámek použít v režimu pro čtení nebo pro zápis.