Synchronizace a řízení přístupu ke zdrojům

Tématicky zaměřený vývoj aplikací v jazyce C
skupina Systémové programování – Linux

Jiří Novosad

Fakulta informatiky
Masarykova univerzita
novosad@fi.muni.cz

Brno, 28. 10. 2014

Úvod

motivace, řízení přístupu ke zdrojům

Motivace – Problém souběhu (race conditions)

Výsledek operací prováděných nad sdíleným zdrojem různými procesy (vlákny) závisí na časování běhu procesů. Problém atomicity operací.

Příklad – inkrementace sdíleného čitače

```
int counter = 0;
void *my_thread (void* p)
{
        int i = 0;
        for (i = 0; i < 50000; i++) {
                counter++;
        }
        return NULL;
}
```

Motivace – Problém souběhu (race conditions)

Výsledek operací prováděných nad sdíleným zdrojem různými procesy (vlákny) závisí na časování běhu procesů. Problém atomicity operací.

Příklad – inkrementace sdíleného čitače

```
int counter = 0;
void *my_thread (void* p)
{
        int i = 0;
        for (i = 0; i < 50000; i++) {
                counter++;
        }
        return NULL;
}
```

→ řízení přístupu ke zdrojům

Další profláknuté příklady

- Čtenáři a písaři (en.wikipedia.org/wiki/Readers-writers_problem)
- Večeřící filosofové (en.wikipedia.org/wiki/Dining_philosophers_problem)

Pojmy

mutual exclusion – vzájemné vyloučení, když zdroj používá jedno vlákno, další k němu nesmí přistoupit (Alice a Bob mají psa a kočku a chtějí je pustit na dvorek)

kritická sekce – část kódu, kterou nemůže vykonávat více vláken najednou

futex – nízkoúrovňový synchronizační objekt v jádře Linuxu, kterým jsou implementovány ostatní synchronizační mechanismy

mutex

spinlock

semafor

bariéra

podmínková proměnná

Teorie řešení problému kritické sekce

Tři podmínky řešení problému KS

1. vzájemné vyloučení (mutual exclusion)
2. absence uváznutí (deadlock)
3. žádné stárnutí (konečné čekání – starvation)

Řešení

- algoritmy vzájemného vyloučení – Dekker, Peterson (ale přeuspořádání přístupů k paměti)
- hardwarové atomické instrukce
- → knihovny operačního systému

Linux vám poskytne nástroje, ale správné použití je na vás.

Mutexy

práce s mutexy, atributy mutexů

mutex – Mutual Exclusion lock

- Zamknout mutex může pouze 1 vlákno
- Další vlákna pak při pokusu o zamčení čekájí na odemčení
- Atomicita zamykání mutexu je garantována jádrem

```
#include <pthread.h>
int pthread_mutex_init(pthread_mutex_t *restrict mutex,
                       const pthread_mutexattr_t *restrict attr);
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
int pthread_mutex_destroy(pthread_mutex_t *mutex);
int pthread_mutex_lock(pthread_mutex_t *mutex);
int pthread_mutex_trylock(pthread_mutex_t *mutex);
int pthread_mutex_unlock(pthread_mutex_t *mutex);
```

úkol

- Projděte si soubor `soubeh.c` ve studijních materiálech
- Vyřešte problém souběhu pomocí mutexu

Problém uváznutí (deadlock)

- zapomenutý zamknutý mutex
- dvojnásobné zamčení mutexu

Problém uváznutí (deadlock)

- zapomenutý zamknutý mutex
- dvojnásobné zamčení mutexu

Typy mutexů

- rychlý mutex, defaultní v Linuxu (`PTHREAD_MUTEX_NORMAL`)
- rekurzivní mutex – povoleno násobné zamykání, ale musí následovat stejný počet odemčení (`PTHREAD_MUTEX_RECURSIVE`) – používat vyjímečně (invariant)
- kontrolovaný mutex – jádro kontroluje zamykání a při pokusu o násobné zamčení vrací `EDEADLK` (`PTHREAD_MUTEX_ERRORCHECK`)

robustní mutex – umožňuje vyrovnat se se zamknutým mutexem při ukončení vlákna – `pthread_mutexattr_getrobust(3p)`

Atributy mutexů

- nastavení vlastností mutexů – lze je ale nastavit pouze při vytvoření mutexu
- `man -k pthread_mutexattr`

Použití

1. `pthread_mutexattr_init()`
2. `pthread_mutexattr_set*`
3. použití atributu ve funkci `pthread_mutex_init()`
4. `pthread_mutexattr_destroy()`

Příklad

- `pthread_mutexattr_settype(&attr, PTHREAD_MUTEX_ERRORCHECK)`

spinlock

- aktivní čekání na odemčení zámku
- vhodný na vícejádrových systémech, když víte, že skutečně nebudete čekat dlouho
- příliš se nepoužívá (spíše uvnitř jádra, nebo bez preempce – jinak hrozí odložení procesu, který drží zámek)

```
#define _POSIX_C_SOURCE=200112L
#include <pthread.h>

int pthread_spin_init(pthread_spinlock_t *lock, int pshared);
int pthread_spin_destroy(pthread_spinlock_t *lock);

int pthread_spin_lock(pthread_spinlock_t *lock);
int pthread_spin_unlock(pthread_spinlock_t *lock);
```

Čtenáři a písaři

- občas je dobré mít různé zámky podle typu operace v kritické sekci (čtecí nebo zápisová)
- nevadí, když více vláken data pouze čte – nikdo ale nesmí zároveň zapisovat
- když už někdo v daný čas zapisuje, nesmí nikdo další ani číst, ani zapisovat

```
#include <pthread.h>
int pthread_rwlock_init(pthread_rwlock_t *restrict rwlock,
                        const pthread_rwlockattr_t *restrict attr);
int pthread_rwlock_destroy(pthread_rwlock_t *rwlock);

int pthread_rwlock_rdlock(pthread_rwlock_t *rwlock);
int pthread_rwlock_wrlock(pthread_rwlock_t *rwlock);
int pthread_rwlock_unlock(pthread_rwlock_t *rwlock);
```

Čtenáři a písaři

- občas je dobré mít různé zámky podle typu operace v kritické sekci (čtecí nebo zápisová)
- nevadí, když více vláken data pouze čte – nikdo ale nesmí zároveň zapisovat
- když už někdo v daný čas zapisuje, nesmí nikdo další ani číst, ani zapisovat

```
#include <pthread.h>
int pthread_rwlock_init(pthread_rwlock_t *restrict rwlock,
                        const pthread_rwlockattr_t *restrict attr);
int pthread_rwlock_destroy(pthread_rwlock_t *rwlock);

int pthread_rwlock_rdlock(pthread_rwlock_t *rwlock);
int pthread_rwlock_wrlock(pthread_rwlock_t *rwlock);
int pthread_rwlock_unlock(pthread_rwlock_t *rwlock);
```

- K zámkům souborů se dostaneme v některé z příštích lekcí

Semafory

práce se semafory

Semafor

- obecnější verze mutexu
- lze povolit vstup více vláken do kritické sekce až do limitu
- původně součást System V IPC (`semget`, `semop`, ...), my se budeme zabývat jednoduššími a přehlednějšími POSIX semafory (`sem_*`) → `-pthread`
- úlohy typu Producent-Konzument (problém omezeného bufferu)

Princip fungování

- nezáporný čítač s počáteční hodnotou (počet vláken, které mohou vstoupit do kritické sekce, počet zdrojů)
- inkrement čítače funkcí `sem_post`
- dekrement čítače funkcí `sem_wait`, která je blokující v případě, že má čítač hodnotu 0
- jádro garantuje atomicitu operací
- další informace: `man 7 sem_overview`

Semafory – použití

```
#include <semaphore.h>
int sem_wait(sem_t *sem);
int sem_post(sem_t *sem);
int sem_trywait(sem_t *sem);
int sem_timedwait(sem_t *sem, const struct timespec *abs_timeout);
int sem_getvalue(sem_t *sem, int *sval);
```

Pojmenované semafory

- Dostupné přes virtuální filesystém v /dev/shm/

```
sem_t *sem_open(const char *name, int oflag,
                mode_t mode, unsigned int value);
int sem_close(sem_t *sem);
int sem_unlink(const char *name);
```

Nepojmenované semafory

- Nutno alokovat paměť dostupnou všem vláknům (procesům[1])

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

[1] paměť sdílenou mezi procesy probereme později

úkol

- Upravte předchozí úlohu s mutexy tak, že fronta není naplněna na začátku, ale jedno vlákno funguje jako producent průběžně doplňující data do fronty. Ostatní vlákna pak tyto úlohy zpracovávají.
- Tip: semafor jako čítač prvků fronty

Další synchronizační objekty

bariéry a podmínkové proměnné

Bariéra

- Zarážka – synchronizace vláken na konkrétní místo v programu.
- Dokud se na bariéře nezastaví specifikovaný počet vláken, jsou všechna blokována, následně jsou všechna najednou probuzena.
- Použití – např. sběr výsledků výpočtu

```
#include <pthread.h>
int pthread_barrier_init(pthread_barrier_t *restrict barrier,
        const pthread_barrierattr_t *restrict attr, unsigned count);

int pthread_barrier_wait(pthread_barrier_t *barrier);

int pthread_barrier_destroy(pthread_barrier_t *barrier);
```

Podmínková proměnná

- Obdoba bariéry, ale nečeká se na určitý počet čekajících vláken, ale na splnění obecné podmínky.
- Při splnění podmínky mohou být spuštena všechna vlákna, nebo jen jedno.
- K zajištění atomicity využívá pomocný mutex, který chrání podmínku.
- Odstraňuje busy-waiting (`condvar.c`)

Podmínkové proměnné – funkce

```
#include <pthread.h>
int pthread_cond_init(pthread_cond_t *restrict cond,
        const pthread_condattr_t *restrict attr);
int pthread_cond_destroy(pthread_cond_t *cond);
pthread_cond_t cond = PTHREAD_COND_INITIALIZER;

int pthread_cond_wait(pthread_cond_t *restrict cond,
        pthread_mutex_t *restrict mutex);

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

Podmínková proměnná – princip

Postup – čekání na podmínku

1. vlákno zamkne mutex a přečte příznak (ověří podmínku)
2. je-li podmínka nastavena, odemkne mutex a provede práci
3. není-li podmínka nastavena, zavolá funkci `pthread_cond_wait`, která automaticky (a atomicky) odemkne mutex a uspí se
4. po probuzení se automaticky uzamkne mutex a znovu se kontroluje podmínka

Postup – nastavení podmínky

1. vlákno zamkne mutex podmínkové proměnné
2. upraví příznak/podmínku
3. zavolá `pthread_cond_signal` nebo `pthread_cond_broadcast`, čímž probudí čekající vlákno (vlákna)
4. probuzené vlákno se pokusí zamknout mutex (znovu se uspí při čekání na odemknutí mutexu)
5. vlákno, které probouzelo ostatní, odemkne mutex a uvolní tak cestu některému čekajícímu vláknu

Podmínkové proměnné – příklad

```
pthread_cond_init(&cond, NULL);
pthread_mutex_lock(&mt); /* lock condition's mutex */
while (mycond != 0) { /* test the condition */
        /* automatically unlocks mutex and falls asleep */
        /* automatically locks mutex on waking up */
        pthread_cond_wait(&cond, &mt);
}
pthread_mutex_unlock(&mt); /* unlock the mutex */

...

/* waking up */
pthread_mutex_lock(&mt); /* lock condition's mutex */
... /* change the condition */
pthread_cond_signal(&cond); /* wake up another waiting thread */
pthread_mutex_unlock(&mt);
```

úkol

Upravte `condvar.c` tak, aby nedocházelo k busy waitingu.

Závěr

shrnutí, domácí úkoly a zdroje

Synchronizace mezi procesy

- Všechny zmíněné synchronizační mechanismy lze použít i pro synchronizaci procesů.
- U funkcí z `pthread.h` je třeba pomocí atributů deklarovat použití mezi procesy.
- Většinou je ale potřeba využít sdílenou paměť – někdy příště.

Domácí úkol

Knihovna front

- implementujte knihovnu pro práci s frontami
- API podrobně v `queue.h`
- knihovna (všechny funkce) je thread-safe
- nezapomeňte ošetřit chybné použití (pořadí) funkcí
  - volání enqueue/dequeue funkcí před inicializací/po ukončení
  - vícenásobné volání inicializace
  - ...
- nezapomeňte, že dequeue je blokující, ale nesmí docházet k busy-waitingu
- před odevzdáním otestujte!

Zdroje

- computing.llnl.gov/tutorials/pthreads/
- www.yolinux.com/TUTORIALS/LinuxTutorialPosixThreads.html
- www.csc.villanova.edu/∼mdamian/threads/posixsem.html

Fronty

- System V IPC Message Passing

  beej.us/guide/bgipc/output/html/multipage/mq.html
- POSIX Message Queues

  www.users.pjwstk.edu.pl/∼jms/qnx/help/watcom/clibref/mq_overview.html