Synchronizace a řízení přístupu ke zdrojům

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

Radek Krejčí

Fakulta informatiky
Masarykova univerzita
radek.krejci@mail.muni.cz

Brno, 10. litopadu 2010

Úvod

motivace, řízení přístupu ke zdrojům

Motivace – Problém souběhu (race conditions)

Problém zpracování/přístupu ke sdíleným datům/zdrojům (problém atomicity operací)

Příklad – zápis do souboru

```
void*
my_thread(void* s)
{
        int i;
        for(i = 0; i < 10; i++) {
                printf("%s", (char*)s);
        }
        printf("\n");
        return NULL;
}
```

Motivace – Problém souběhu (race conditions)

Problém zpracování/přístupu ke sdíleným datům/zdrojům (problém atomicity operací)

Příklad – zápis do souboru

```
void*
my_thread(void* s)
{
        int i;
        for(i = 0; i < 10; i++) {
                printf("%s", (char*)s);
        }
        printf("\n");
        return NULL;
}
```

→ řízení přístupu ke zdrojům

Další profláknuté příklady

- Čtenáři a písaři
- Večeřící filosofové (cs.wikipedia.org/wiki/Problém_obědvajících_filosofů)

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

mutex –

spinlock –

semafor –

bariéra –

podmínková proměnná –

Teorie řešení problému kritické sekce

3 podmínky řešení problému KS

1. vzájemné vyloučení
2. vyloučení uváznutí (deadlock)
3. vyloučení stárnutí (konečné čekáná na vstup do KS)

Algoritmy vzájemného vyloučení

- Dekker
- Peterson

Teorie řešení problému kritické sekce

3 podmínky řešení problému KS

1. vzájemné vyloučení
2. vyloučení uváznutí (deadlock)
3. vyloučení stárnutí (konečné čekáná na vstup do KS)

Algoritmy vzájemného vyloučení

- Dekker
- Peterson

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);
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 `soubeh.c` z adresáře `data/`

úkol

- Projděte si `soubeh.c` z adresáře `data/`
- 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)
- rekurzivní mutex – povoleno násobné zamykání, ale musí následovat stejný počet odemčení
- kontrolovaný mutex – jádro kontroluje zamykání a při pokusu o násobné zamčení vrací `EDEADLK`
- *robustní mutex* – vyrovná se se zamknutým mutexem při ukončení vlákna (Nepřenositelný!)

Atributy mutexů

- nastavení vlastností mutexů – lze je ale nastavit pouze před vytvořením mutexu
- `man -k pthread_mutexattr`

Použití

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

Příklady

- `pthread_attr_settype(pthread_mutexattr_t *attr, int type)`

spinlock – aktivní zámek

- aktivní čekání na odemčení zámku
- příliš se nepoužívá

spinlock – aktivní zámek

- aktivní čekání na odemčení zámku
- příliš se nepoužívá
- vhodný na vícejádrových systémech, když víte, že skutečně nebudete čekat dlouho

```
#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
- nevadí, když více vláken data pouze čte – nikdo ale nesmí zapisovat
- když už někdo zapisuje, nesmí nikdo jiný ani číst, ani zapisovat

```
#include <pthread.h>
int pthread_rwlock_destroy(pthread_rwlock_t *rwlock);
int pthread_rwlock_init(pthread_rwlock_t *restrict rwlock,
                        const pthread_rwlockattr_t *restrict attr);
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
- nevadí, když více vláken data pouze čte – nikdo ale nesmí zapisovat
- když už někdo zapisuje, nesmí nikdo jiný ani číst, ani zapisovat

```
#include <pthread.h>
int pthread_rwlock_destroy(pthread_rwlock_t *rwlock);
int pthread_rwlock_init(pthread_rwlock_t *restrict rwlock,
                        const pthread_rwlockattr_t *restrict attr);
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 vrátíme v některé z příštích lekcí

Semafory

práce se semafory

Semafor

- obecnější verze mutexu
- lze povolit vstup více vláken do kritické sekce
- původně součást System V IPC, my se budeme zabývat jednoduššími a přehlednějšími POSIX semafory
- úlohy typu Producent-Konzument
- problém omezeného bufferu

Princip fungování

- v principu nezáporný čítač s počáteční hodnotou (počet vláken, které mohou vstoupit do kritické sekce)
- 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.

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 všechny blokovány, následně jsou všechny najednou probuzeny.

```
#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);
```

Podmínkové proměnné

- Obdoba bariéry, ale nečeká se na určitý počet čekajících vláken, ale na splnění podmínky.
- Při splnění podmínky mohou být spuštěny všechny vlákna, nebo jen jedno.
- Využívá pomocný mutex

```
while(1) {
        int flag_is_set;

        /* protect accesing the flag */
        pthread_mutex_lock (&flag_mutex);
        flag_is_set = flag;
        pthread_mutex_unlock (&flag_mutex);

        if (flag_is_set) {
                /* do the job */
        }
}
```

Podmínkové proměnné

- Obdoba bariéry, ale nečeká se na určitý počet čekajících vláken, ale na splnění podmínky.
- Při splnění podmínky mohou být spuštěny všechny vlákna, nebo jen jedno.
- Využívá pomocný mutex
- Odstraňuje busy-waiting:
-

```
while(1) {
        int flag_is_set;

        /* protect accesing the flag */
        pthread_mutex_lock (&flag_mutex);
        flag_is_set = flag;
        pthread_mutex_unlock (&flag_mutex);

        if (flag_is_set) {
                /* do the job */
        }
}
```

Podmínkové proměnné – 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 vlákno `pthread_cond_wait`, ta automaticky (a atomicky) odemkne mutex a uspí se
4. po probuzení se uzamkne mutex a znovu se kontroluje podmínka

Postup – nastavení podmínky

1. vlákno zamkne mutex podmínkové proměnné
2. upraví příznak/podmínku
3. zavolá `pthread_cond_signál` 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 čekajícímu vláknu

Podmínkové proměnné – funkce

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

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 cyklus ze začátku části o podmínkových proměnných 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
- Některé nabízejí vlastní mechanismus (pojmenované semafory)
- U funkcí z `pthread.h` je třeba pomocí atributů deklarovat požutí mezi procesy
- Většinou je ale potřeba použít sdílenou paměť – příště.

Domácí úkol

Vyhledávání řetězce v souboru

- Program vyhledává v zadaném textovém souboru řetězec a vypisuje čísla řádků, kde řetězec nalezl
- Uživatel má možnost kdykoliv změnit vyhledávaný řetězec – program čte zadání ze standardního vstupu a na nově zadaný řetězec reaguje okamžitě novým vyhledáváním od začátku souboru
- Prázdný řádek znamená zastavení vyhledávání a čekání na nové zadání
- Program se ukončuje znakem konce souboru (Ctrl + D) zadaným uživatelem nebo signálem SIGTERM

Zdroje

- www.ibm.com/developerworks/aix/library/au-ipc/index.html
- computing.llnl.gov/tutorials/pthreads/
- www.yolinux.com/TUTORIALS/LinuxTutorialPosixThreads.html
- www.csc.villanova.edu/∼mdamian/threads/posixsem.html

Pro zájemce

- 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