PB173 Linux

08 Synchronizácia vláken

Roman Lacko xlacko1@fi.muni.cz

2022-11-11

Obsah

Súbeh

Súbeh

A Knock knock

A Race conditions

B Who is there?

Súbeh

Stav, kedy správanie systému závisí na časovaní akcií v súbežných úlohách.

$\alpha = 1$	$\tau = 0$	$\tau = 1$	$\tau = 2$	$\tau = 3$
t_0 :	$x_0 \leftarrow \alpha$	$x_1 \leftarrow x_0 - 1$	$x_2 \leftarrow \alpha + x_1$	$\alpha \leftarrow x_2$
t_1 :	$y_0 \leftarrow \alpha$	$\alpha \leftarrow y_0 + 5$		

Jazykové okienko

Súbežnosť (*concurrency*) je schopnosť systému vykonávať v jednom čase viac úloh.

Súbeh (*race condition*) je stav popísaný vyššie.

Súbeh: Terminológia

Kritická sekcia

Kód pracujúci nad *zdieľanými zdrojmi* [1], ktorý musí byť vykonaný ako celok.

Vzájomné vylúčenie

V každom čase vykonáva kritickú sekciu nanajvýš jedna úloha.

Problém kritickej sekcie

Zaručenie vzájomného vylúčenia v kritickej sekcii s minimalizáciou patologických stavov.

---

[1] Zdieľaná pamäť sa niekedy v literatúre volá *kritická oblasť*

Súbeh: Riešenie kritickej sekcie

Riešenie kritickej sekcie vyžaduje:

Vzájomné vylúčenie

Postup

Ak na vstup do voľnej sekcie čakajú nejaké procesy, musí byť vstup jednému z nich povolený.

Obmedzené čakanie

Existuje konečný horný limit dĺžky čakania na vstup do sekcie.

Súbeh: Riešenie kritickej sekcie

Typické riešenia

- podpora hardware (atomické inštrukcie, Dijkstra),
- algoritmy (napr. Dekker, Peterson, Filter,),
- služby OS (napr. `futex()`)
- knižnice (napr. POSIX Threads Library, stdc od C11)

! Linux a pthreads

Systém poskytuje nástroje, ale správne použitie je na programátoroch!

Linux

```c
#include <linux/futex.h>
#include <sys/syscall.h>

long syscall(SYS_futex, uint32_t *uaddr, int futex_op,
        uint32_t val, const struct timespec *timeout, …);

#define FUTEX_WAIT /* … */
#define FUTEX_WAKE /* … */
…
```

⚠ futex bolí

Použitie surového systémového volania je nepohodlné, preferujte nadstavby v POSIX Threads.

Zámky

Zámky

Zámky: Mutex

Mutex (*Mutual Exclusion Lock*)

Základný objekt pre vzájomné vylúčenie.

- Zamknutie ≈ vstup do kritickej sekcie
- Odomknutie ≈ výstup z kritickej sekcie

⚠ Falošný pocit bezpečia

Mutex sa *dá* použiť na implementáciu kritickej sekcie,
ale sám korektnosť negarantuje.

Zámky: Mutex

```
#include <pthread.h>

int pthread_mutex_init(pthread_mutex_t *mutex,
        pthread_mutexattr_t *attr);
int pthread_mutex_destroy(pthread_mutex_t *mutex);
#define PTHREAD_MUTEX_INITIALIZER /* … */

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

int pthread_mutex_trylock(pthread_mutex_t *mutex);
int pthread_mutex_timedlock(pthread_mutex_t *mutex,
        const struct timespec *timeout);
```

Zámky: Mutex: Atribúty

Atribúty podobne ako pre `pthread_create()`

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

int pthread_mutexattr_set…(&attr, …);
int pthread_mutexattr_get…(&attr, …);
```

Zámky: Mutex

Typické chyby pri používaní mutexu:

Dvojnásobné zamknutie

U bežného mutexu spôsobí uviaznutie.
Možným riešením je *rekurzívny zámok*.

Zabudnuté odomknutie

Spôsobí uviaznutie. Nemá jednoduché riešenie.
Môže pomôcť *robustný mutex*.

Práca so zdrojmi

Pri práci so zdrojmi môžu pomôcť *Sentry Functions*
viď `patterns/13-resources.c`

Zámky: Mutex

```
int pthread_mutexattr_settype(… *attr, int type);
#define PTHREAD_MUTEX_NORMAL /* … */
#define PTHREAD_MUTEX_ERRORCHECK /* … */
#define PTHREAD_MUTEX_RECURSIVE /* … */
#define PTHREAD_MUTEX_DEFAULT /* … */

int pthread_mutexattr_setrobust(… *attr, int robustness);
#define PTHREAD_MUTEX_STALLED /* … */
#define PTHREAD_MUTEX_ROBUST /* … */

int pthread_mutex_consistent(pthread_mutex_t *mutex);
```

Zámky: Spin Lock

Spin Lock

- Aktívne čakanie.
- Efektívny pre krátke kritické sekcie.
- Vyťažuje procesor.

⚠ Všeobecne preferujte Mutex

Ak pri profilovaní identifikujete kritickú sekciu, pri ktorej uspanie trvá dlhšie než vykonanie, môžete zvážiť úpravu.

Zámky: Spin Lock

```
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);

int pthread_spin_trylock(pthread_spinlock_t *lock);
```

Zámky: Read-Write Lock

Read-Write Lock

- Vstup do kritickej sekcie podľa typu operácie
- **Read** (tiež *shared*) zámok pre **viacerých** čitateľov
- **Write** (tiež *exclusive*) zámok pre jedného zapisovateľa
- V každom čase maximálne jeden *typ* zámku

Zámky: Read-Write Lock

```
int pthread_rwlock_init(pthread_rwlock_t *rwlock,
        const pthread_rwlockattr_t *attr);
int pthread_rwlock_destroy(pthread_rwlock_t *rwlock);
#define PTHREAD_RWLOCK_INITIALIZER /* … */

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

/* ζ ∈ {rd, wr} */
int pthread_rwlock_tryζlock(pthread_rwlock_t *rwlock);
int pthread_rwlock_timedζlock(pthread_rwlock_t *rwlock,
        const struct *timeout);
```

Zámky: Read-Write Lock

Atribúty

```
int pthread_rwlockattr_init(pthread_rwlockattr_t *attr);
int pthread_rwlockattr_destroy(pthread_rwlockattr_t *attr);

int pthread_rwlockattr_set…(&attr, …);
int pthread_rwlockattr_get…(&attr, …);
```

Zámky: Read-Write Lock

Pozor na neintuitívne problémy:

1. Čítajúci zámok je rekurzívny, zápisový **nie je**.
2. Pri čakaní na oba zámky POSIX nedefinuje preferenciu, hrozí porušenie podmienok riešenia kritickej sekcie!

GNU NPTL umožňuje nastaviť preferenciu:

```
pthread_rwlockattr_setkind_np(… *attr, int pref);

#define PTHREAD_RWLOCK_PREFER_READER_NP /* … */
#define PTHREAD_RWLOCK_PREFER_WRITER_NP /* … */
#define PTHREAD_RWLOCK_PREFER_WRITER_NONRECURSIVE_NP /* … */
```

Zámky: Read-Write Zámok

Preferenciu vláken môže upraviť **Thread Execution Scheduling**:

```
#include <pthread.h>
#include <sched.h>

int pthread_setschedparam(pthread_t thread, int policy,
        const struct sched_param *param);
#define SCHED_OTHER /* … */
#define SCHED_/* { RR, FIFO, … } */ /* … */

struct sched_param {
    int sched_priority;
    /* Others are ignored, see sched.h */
};

int pthread_schedprio(pthread_t thread, int prio);
```

Podmienková premenná

Podmienková premenná

Condition Variable

- Čakanie na splnenie nejakej podmienky
- Nutné použiť spolu s Mutexom

Podmienková premenná

```
int pthread_cond_init(pthread_cont_t *cond,
        const pthread_condattr_t *attr);
int pthread_cond_destroy(pthread_cont_t *cond);
#define PTHREAD_COND_INITIALIZER /* … */

int pthread_cond_wait(… *cond, pthread_mutex_t *mutex);
int pthread_cond_timedwait(… *cond, pthread_mutex_t *mutex,
        const struct timespec *timeout);

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

Podmienková premenná

Typický vzor použitia

```
pthread_mutex_lock(&mutex);
while (!CONDITION) {
    pthread_cond_wait(&cond, &mutex);
}

/* At this point,
 * 1. The CONDITION is true.
 * 2. We are in a critical section (the mutex is locked). */

pthread_mutex_unlock(&mutex);
```

Čakanie bez podmienky

Zriedka sa podmienková premenná použije bez podmienky, napr. ak nás len zaujíma "signál" od iného vlákna.

Ostatné mechanizmy

Bariéra

Thread Barrier

- Synchronizácia určitého počtu vláken v jednom bode

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

int pthread_barrier_wait(pthread_barrier_t *barrier);
#define PTHREAD_BARRIER_SERIAL_THREAD /* … */
```

Dynamická inicializácia

Dynamic Package Initialization

- Zaručene spustí inicializačnú funkciu len raz
  aj pri volaní z viacerých vláken
- Používa sa obvykle na lenivú inicializáciu

```
#define PTHREAD_ONCE_INIT /* … */
int pthread_once(pthread_once_t *once, void (*init)(void));
```

Poznámky

Helgrind

Helgrind je jeden z nástrojov Valgrindu, umožňuje odhaliť niektoré typy chýb v synchronizácii.

```
$ valgrind --tool=helgrind PROGRAM [ARGUMENT…]
```

Reentrantné volania

Reentrantná procedúra je procedúra, ktorá nespôsobí súbeh pri súbežnom vykonávaní.

- Štandardné funkcie obvykle reentrantné **nie sú**.
- K `φ()` niekedy existuje reentrantný variant `φ_r()`.

Je to problém?

```
/* t₁ */ struct passwd pw = getpwuid(1000)… /* unfinished… */
/* t₂ */ struct passwd pw = getpwuid(0);
/* t₁ */ /* …continued */ …getpwuid(1000);
/* t₁ */ printf("%s\n", pw->pw_login);      /* root?! */
```

Synchronizácia procesov

Je možné POSIX Threads mechanizmy použiť medzi procesmi?

Príbuzné procesy

- Áno, ak vzniknú pred `fork()`
- Nutné použitie *zdieľanej pamäte*
- Zdieľanie treba explicitne povoliť
  napr. `pthread_mutexattr_setpshared()`

Nepríbuzné procesy

- Nie
- Alternatívy: SysV IPC, FS Locks

Štandard C

Štandard C

Niektoré mechanizmy dostupné od C11

Pozor na možnú rozdielnu sémantiku

- `pthread_mutex_α()` ? `mtx_α()`
- `pthread_cond_α()` ? `cnd_α()`
- `pthread_once()` ? `call_once()`

Spin Lock, Read-Write Lock ani bariéra sa do C11 nedostali.

Záver

Zdroje

- POSIX Thread Libraries
- Scheduling Policies
- Process CPU Priority and Scheduling