

Curs 2

Futex-uri – mecanisme de sincronizare în user space

Hubertus Franke, Rusty Russel, Ulrich Drepper

Operating Systems Practical

9 Octombrie 2013

Sincronizare

Linux Fast Userspace Locking

Implementare

Resurse

Keywords

Intrebari

Sincronizare

Linux Fast Userspace Locking

Implementare

Resurse

Keywords

Intrebari

- ▶ mai multe entități de execuție (procese, thread-uri)
- ▶ acces concurent la o resursă
- ▶ asigurarea consistenței
- ▶ sincronizare
 - ▶ acces exclusiv la resursă
 - ▶ realizarea unei succesiuni de evenimente (un thread așteaptă alt thread)
- ▶ mecanisme de sincronizare
 - ▶ mutex-uri (MUTual Exclusion) (semafoare binare)
 - ▶ semafoare (numărătoare)
 - ▶ monitoare
 - ▶ cozi de așteptare, evenimente
 - ▶ spinlock-uri

► Sincronizare în Linux

► fcntl

```

► fl.l_type = F_WRLCK;
  fl.l_whence = SEEK_SET;
  fl.l_start = 100;
  fl.l_len = 10;
  fcntl(fd, F_SETLK, &fl)

```

► System V semaphores:

```

► sop.sem_num = 0;
  sop.sem_op = -1;
  sop.sem_flg = 0;
  semop(semId, &sop, 1);

```

► Userspace locking

- alternativă la mecanismele “heavyweight” de mai sus
- excluderea se realizează prin operații atomice
- numai cazul în care lock-ul este achiziționat (lock contention) necesită intervenția nucleului
 - este nevoie să se replanifice altă entitate de execuție
- Linux futexes (începând cu Linux 2.5.7)

Sincronizare

Linux Fast Userspace Locking

Implementare

Resurse

Keywords

Intrebari

- ▶ echitate (fairness) și performanță (throughput)
- ▶ fair locking
 - ▶ se trezește procesul care a așteptat cel mai mult
 - ▶ apare 'convoy problem'
 - ▶ prelucrarea se face la viteza celui mai încet proces
- ▶ random fairness
 - ▶ sunt trezite toate procesele
 - ▶ procesele concurează pentru achiziționarea lock-ului
 - ▶ apare 'thundering herd problem'
- ▶ greedy locking
 - ▶ se trezește un singur thread
 - ▶ se poate întâmpla să fie vorba de thread-ul care tocmai a eliberat lock-ul -> starvation
- ▶ este nevoie de o echilibrare a schemelor de mai sus

- ▶ două scopuri principale
 - ▶ evitarea apelurilor de sistem
 - ▶ evitarea schimbărilor de context
- ▶ pentru atingerea scopurilor se stabilesc două contexte
 - ▶ lock-ul nu este achiziționat de alt thread (uncontended case)
 - ▶ nu este nevoie de apel de sistem
 - ▶ lock-ul este ținut într-o zonă de memorie partajată
 - ▶ asupra lock-ului se efectuează operații atomice
 - ▶ lock-ul este achiziționat de alt thread (contended case)
 - ▶ se face apel de sistem pentru blocarea thread-ului și replanificare
 - ▶ o coadă de așteptare în kernel

Sincronizare

Linux Fast Userspace Locking

Implementare

Resurse

Keywords

Intrebari

- un tip de date opac care definește lock-ul

```
typedef struct ulock_t {
    long status;
} ulock_t;

static inline int usema_down (ulock_t *ulock)
{
    if (!__ulock_down (ulock))
        return 0;
    return sys_ulock_wait (ulock);
}
```

- `__ulock_down` – operație atomică de decrementare
 - întoarce 0 pentru uncontended case
 - diferit de 0 pentru contended case (apel de sistem)
- `status < 0` înseamnă contention
- condițiile de cursă sunt rezolvate de nucleu

- ▶ alocare explicită a unui obiect în kernel (coadă de așteptare + semnătură de securitate)
 - ▶ adresa este exportată în spațiul utilizator
 - ▶ la fiecare apel de sistem se verifică semnătura
 - ▶ probleme
 - ▶ apel explicit de creare/eliberare a obiectului
 - ▶ securitatea este limitată de dimensiunea cheii
- ▶ ulocks – lock word + număr de cozi de așteptare
 - ▶ lock word-ul nu este accesat de kernel
 - ▶ descrie starea lock-ului și numărul de thread-uri care așteaptă
 - ▶ cozi de așteptare folosind struct semaphore
 - ▶ lock-ul poate fi plasat la diverse adrese virtuale în spațiul de fiecărui proces/thread
 - ▶ probleme la căutarea obiectului kernel pe baza adresei

- ▶ identificator unic pentru fiecare futex
 - ▶ pointer la o structură struct page
 - ▶ offset-ul în acea structură
- ▶ structura din kernel asociată unui proces este plasată într-o tabelă hash
- ▶ denumirea de fast userspace mutex a fost condensată la futex

- ▶ apelul de sistem
 - ▶ `sys_futex (struct futex *, int op);`
 - ▶ `op` poate fi `FUTEX_UP` sau `FUTEX_DOWN`
- ▶ pași în codul kernel
 - ▶ se verifică adresa din userspace
 - ▶ pagina este 'pinned'; se incrementează referința structurii `struct page` pentru a nu fi swappata
 - ▶ `struct page + offset` dă adresa `futex`-ului (căutare în tabela hash de `futex-uri`)
- ▶ se efectuează operații în funcție de `op`
- ▶ dacă `op` este invalid se întoarce eroare
- ▶ pagina este 'unpinned'

- ▶ `op = FUTEX_DOWN`
 - ▶ procesul este marcat `INTERRUPTIBLE`
 - ▶ se încearcă o decrementare a valorii de la adresa asociată `futex`-ului
 - ▶ dacă nu se decrementează contorul la 0 se planifică procesul/thread-ul (schedule)
 - ▶ dacă se decrementează contorul la 0
 - ▶ procesul este marcat `RUNNING`
 - ▶ se trezește alt proces care așteaptă pentru a decrementa `futex`-ul la -1; astfel va indica așteptare
- ▶ `op = FUTEX_UP`
 - ▶ contorul `futex`-ului este pus pe 1
 - ▶ se trezește primul proces care așteaptă

- ▶ nu există o implementare directă a `pthread_cond_timedwait`
 - ▶ operația necesită un timeout (timer)
- ▶ primitiva `pthread_cond_broadcast` trezește toate procesele care așteaptă
 - ▶ în implementare un proces iese din kernel dacă obține futex-ul sau dacă primește un semnal
- ▶ în implementări de thread-uri N:M este nevoie de o interfață asincronă pentru informații despre un futex (un proces poate avea mai multe thread-uri)
- ▶ poate apărea starvation
 - ▶ un proces renunță la lock și apoi dorește achiziționarea lui îl poate reachiziționa rapid

- ▶ `sys_futex (struct futex *, int op, int val, struct timespec *reltime);`
- ▶ `op = FUTEX_WAIT`
 - ▶ marcarea proces ca INTERRUPTIBLE
 - ▶ citire valoare futex
 - ▶ valoare citită $\neq$ val $\rightarrow$ valoarea de retur = `EWouldBlock`
 - ▶ altfel, sleep reltime sau nedefinit
 - ▶ procesul va fi trezit la o operație `FUTEX_WAKE`
- ▶ `op = FUTEX_WAKE`
 - ▶ la fel ca `FUTEX_UP` dar nu mai alterează valoarea futex-ului
 - ▶ numărul de procese trezite este controlat de val
- ▶ `op = FUTEX_AWAIT`
 - ▶ procesul este notificat asincron când futexul își schimbă valoarea

- ▶ Pentium III 500 Mhz, 256 MB, Ulockflex
- ▶ fiecare thread
 - ▶ calculează două numere aleatoare $nlht$, lht $[0.5 \dots 1.5]$
 - ▶ achiziționează lock-ul, lucrează timp de lht
 - ▶ dă drumul la lock, lucrează timp de $nhtl$
 - ▶ reia ciclul
- ▶ se raportează numărul de cicluri efectuat de fiecare thread (throughput)
- ▶ două mecanisme
 - ▶ fair wakeup
 - ▶ regular wakeup
- ▶ se folosește și o implementare cu spinlock-uri
 - ▶ înainte de blocare un thread așteaptă $x \mu s$
- ▶ se folosesc configurații ($nlht$, lht)

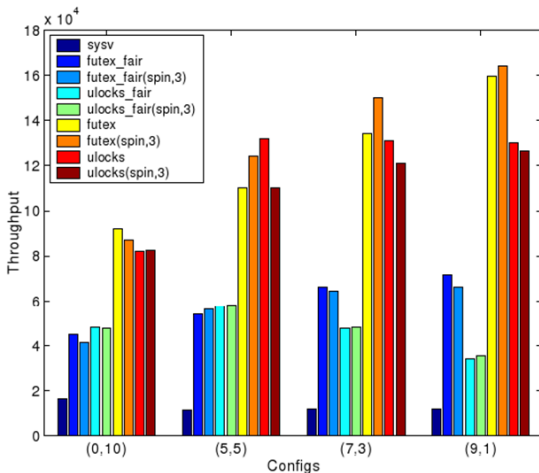


Figure 4: Throughput for various lock types for 100 tasks, 1 lock and 4 configurations

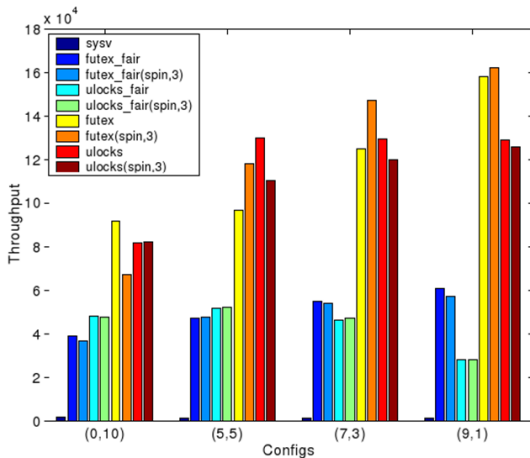


Figure 5: Throughput for various lock types for 1000 tasks, 1 lock and 4 configurations

```
[/usr/include/linux/kernel/futex.c]
```

```
asmlinkage long sys_futex(u32 __user *uaddr, int op, u32 val,  
                          struct timespec __user *utime,  
                          u32 __user *uaddr2, u32 val3)
```

- ▶ uaddr -> adresa futexului în userpace
- ▶ op -> operația de efectuat (multiplexor)
- ▶ val -> valoare folosită la operații
- ▶ utime -> timeout
- ▶ addr2, val3 -> valori folosite de unele operații

▶ FUTEX_WAIT

- ▶ thread-ul așteaptă până când este trezit
- ▶ dacă val nu corespunde valorii futexului se întoarce cu EWOULDBLOCK
- ▶ după trezire apelul se întoarce cu 0
- ▶ timespec specifică timeout-ul (nedefinit la NULL)

▶ FUTEX_WAKE

- ▶ trezirea unuia sau a mai multor thread-uri
- ▶ val spune numărul de thread-uri care se dorește a fi trezite
- ▶ de obicei este 1 sau INT_MAX
- ▶ se întoarce numărul de thread-uri trezite

▶ FUTEX_WAKE_OP

- ▶ folosită pentru implementarea variabilelor condiție
- ▶ folosește și argumentele addr2 și val3

▶ FUTEX_CMP_REQUEUE, FUTEX_REQUEUE, FUTEX_FD

```
class event {
public:
    event (): val (0) { }
    void ev_signal () {
        ++val;
        futex_wake (&val , INT_MAX);
    }
    void ev_wait () {
        futex_wait (&val , val);
    }
private:
    int val;
};
```

► initial

- $val = 0$; mutex-ul este liber

► lock ()

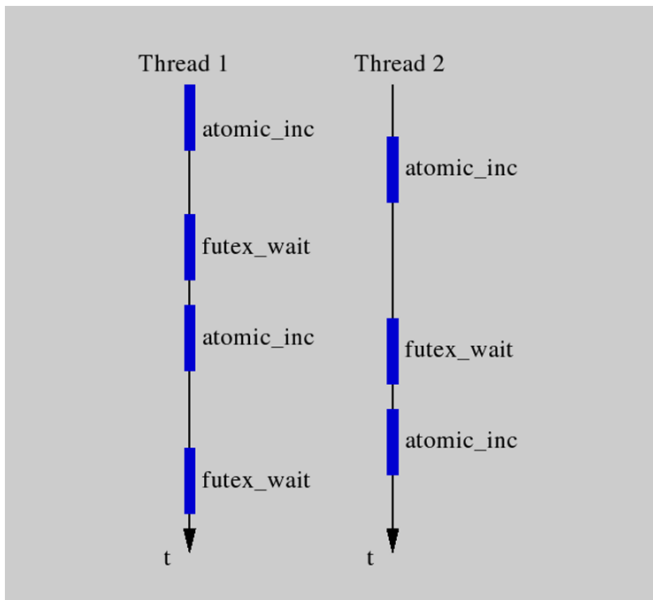
- `atomic_inc` incrementeaza atomic `val` si intoarce fosta valoare
- daca valoarea nu este 0 (lock-ul este ocupat) se apeleaza `futex_wait`
- ciclu `while` (thread-ul poate fi trezit de semnale)
- `futex_wait` are ca argument $c + 1$ ($== val$)

► unlock ()

- valoarea 0 este stocată în `val` (operație atomică) – lock liber
- se trezește un thread

```
class mutex {
public:
    mutex () : val (0) { }
    void lock () {
        int c;
        while ((c = atomic_inc (val)) != 0)
            futex_wait (&val, c + 1);
    }
    void unlock () {
        val = 0;
        futex_wake (&val, 1);
    }
private:
    int val;
};
```

- ▶ ori de câte ori se apelează unlock se face apel de sistem
 - ▶ dacă nu avem thread-uri care așteaptă nu este nevoie de apel de sistem
 - ▶ se creează o nouă stare: locked and no waiters
- ▶ valoarea val poate cauza overflow (2^{32})
 - ▶ se poate întâmpla cu un singur thread întrerupt de un semnal
- ▶ poate apărea o condiție de cursă (race) între apelul atomic_inc și futex_wake
 - ▶ apelul futex_wake se poate întoarce cu EWOULDBLOCK



- ▶ fără overflow
- ▶ fără livelock cauzat de incrementarea valorii futexului
- ▶ trebuie evitate apelurile de sistem `futex_wake` cand nu există thread-uri care așteaptă – 3 stări:
 - ▶ 0 – unlocked
 - ▶ 1 – locked, no waiters
 - ▶ 2 – locked, one or more waiters
 - ▶ nu mai putem folosi `atomic_inc`
 - ▶ putem folosi `cmpxchg` (compare-and-exchange)

```
class mutex2 {
public:
    mutex () : val (0) { }
    void lock () {
        int c;
        if ((c = cmpxchg (val , 0 , 1)) != 0)
            do {
                if (c == 2 || cmpxchg (val , 1 , 2) != 0)
                    futex_wait (&val , 2);
            } while ((c = cmpxchg (val , 0 , 2)) != 0);
    }
    void unlock () {
        if (atomic_dec (val) != 1) {
            val = 0;
            futex_wake (&val , 1);
        }
    }
private:
    int val;
};
```

▶ uncontended

▶ mutex1

▶ lock:

- ▶ 1 op atomică
- ▶ 0 apeluri de sistem

▶ unlock:

- ▶ 0 op atomice
- ▶ 1 apel de sistem

▶ mutex2

▶ lock

- ▶ 1 op atomică
- ▶ 0 apeluri de sistem

▶ unlock

- ▶ 1 op atomică
- ▶ 0 apeluri de sistem

▶ contended

▶ mutex1

▶ lock:

- ▶ 1 (+1) op atomice
- ▶ 1 (+1) apeluri de sistem

▶ unlock

- ▶ 0 op atomice
- ▶ 1 apel de sistem

▶ mutex2

▶ lock

- ▶ 2 (+1) sau 3 (+2) op atomice
- ▶ 1 (+1) apeluri de sistem

▶ unlock

- ▶ 1 op atomică
- ▶ 1 apel de sistem

- ▶ Mutex2 este mai costisitor decât Mutex1 pentru cazul contended
- ▶ codul pentru Mutex2 este corect și optimizează uncontended case
- ▶ unele arhitecturi dețin o instrucțiune atomică xchg (fără cmp)

```
void lock () {
    int c;
    if (c == cmpxchg (val, 0, 1)) != 0) {
        if (c != 2)
            c = xchg (val, 2);
        while (c != 0) {
            futex_wait (&val, 2);
            c = xchg (val, 2);
        }
    }
}
```

- ▶ `/usr/src/glibc/nptl/pthread_mutex_lock.c`
 - ▶ `pthread_mutex_lock -> _lll_mutex_lock`
- ▶ `/usr/src/glibc/nptl/sysdeps/unix/sysv/linux/i386/lowlevellock.h`
 - ▶ `_lll_mutex_lock -> __lll_mutex_lock`
- ▶ `/usr/src/glibc/nptl/sysdeps/unix/sysv/linux/i386/lowlevellock.h`
 - ▶ `__lll_mutex_lock`

```
#define _lll_mutex_lock(futex) \
    (void) ({ int ignore1, ignore2; \
        __asm __volatile (LOCK_INSTR "cmpxchgl %1, %2\n\t" \
            "jnz _L_mutex_lock_%= \n\t" \
            ".subsection 1\n\t" \
            ".type _L_mutex_lock_%=,@function\n" \
            "_L_mutex_lock_%=:\n\t" \
            "leal %2, %%ecx\n\t" \
            "call __lll_mutex_lock_wait\n\t" \
            "jmp 1f\n\t" \
            ".size _L_mutex_lock_%=,.-_L_mutex_lock_%= \n" \
            ".previous\n" \
            "1:" \
            : "=a" (ignore1), "=c" (ignore2), "=m" (futex) \
            : "0" (0), "1" (1), "m" (futex) \
            : "memory"); })
```


Sincronizare

Linux Fast Userspace Locking

Implementare

Resurse

Keywords

Intrebari

- ▶ <http://www.kernel.org/doc/ols/2002/ols2002-pages-479-495.pdf>
- ▶ <http://people.redhat.com/drepper/futex.pdf>
- ▶ <http://people.redhat.com/drepper/nptl-design.pdf>
- ▶ [http://en.wikipedia.org/wiki/Lock_\(computer_science\)](http://en.wikipedia.org/wiki/Lock_(computer_science))
- ▶ Maurice Herlihy, J. Eliot B. Moss – Transactional Memory: Architectural Support for Lock-Free Data Structures
- ▶ Paul. E McKenney, Maged M. Michael, Josh Triplett, Jonathan Walpole – Why the Grass May Not Be Greener On The Other Side: A Comparison of Locking vs. Transactional Memory

Sincronizare

Linux Fast Userspace Locking

Implementare

Resurse

Keywords

Intrebari

- ▶ futex
- ▶ concurrency
- ▶ mutex
- ▶ contention
- ▶ thread
- ▶ sincronizare

Sincronizare

Linux Fast Userspace Locking

Implementare

Resurse

Keywords

Intrebari

?