Sisteme de operare avansate

Futex-uri
(mecanisme de sincronizare în spațiul utilizator)

Hubertus Franke, Rusty Russel; Ulrich Drepper

Sincronizare

- mai multe entităţi de execuţie (procese, thread-uri)
- acces concurent la o resursă
- asigurarea consistenţei
- sincronizare
  - acces exclusiv la resursă
  - realizarea unei succesiuni de evenimente (un thread aşteaptă alt thread)
- mecanisme de sincronizare
  - mutex-uri (MUTual Exclusion) (semafoare binare)
  - semafoare (numărătoare)
  - monitoare
  - cozi de aşteptare, evenimente

Sincronizare în Linux

- 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 “heavyweigth” de mai sus
  - excluderea se realizează prin operaţii atomice
  - numai cazul în care lock-ul este achiziţionat (lock contention) necesită intervenţia nucleului
    - → este nevoie să se replanifice altă entitate de execuţie
  - Linux futexes (începând cu Linux 2.5.7)

Linux Fast Userspace Locking - Cerinţe

- echitate (fairness) şi performanţă (throughput)
- fair locking
  - se trezeşte procesul care a aşteptat cel mai mult
  - apare 'convoy problem'
    - prelucrarea se face la viteza celui mai încet proces
- random fairness
  - sunt trezite toate procesele
  - procesele concurează pentru achiziţionarea lock-ului
  - apare 'thundering herd problem'
- greedy locking
  - se trezeşte un singur thread
  - se poate întâmpla să fie vorba de thread-ul care tocmai a eliberat

Linux Fast Userspace Locking

- două scopuri (conflictuale)
  - evitarea apelurilor de sistem
  - evitarea schimbărilor de context
- pentru atingerea scopurilor se stabilesc două contexte
  - lock-ul nu este achiziţionat de alt thread (uncontended case)
    - nu este nevoie de apel de sistem
    - lock-ul este ţinut într-o zonă de memorie partajată
    - asupra lock-ului se efectuează operaţii atomice
  - lock-ul este achiziţionat de alt thread (contended case)
    - se face apel de sistem pentru blocarea thread-ului şi replanificare
    - o coadă de aşteptare în kernel

Schemă de implementare

- un tip de date opac care defineşte 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 – operaţie atomică de decrementare
  - întoarce 0 pentru uncontended case
  - diferit de 0 pentru contended case (apel de sistem)
- status < 0 înseamnă contention
- condiţiile de cursă sunt rezolvate de nucleu

Implementări posibile

- alocare explicită a unui obiect în kernel (coadă de aşteptare + semnătură de securitate)
  - adresa este exportată în spaţiul 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 aşteptare
  - lock word-ul nu este accesat de kernel
    - descrie starea lock-ului şi numărul de thread-uri care aşteaptă
  - cozi de aşteptare folosind struct semaphore
  - lock-ul poate fi plasat la diverse adrese virtuale în spaţiul de fiecărui proces/thread
    - probleme la căutarea obiectului kernel pe baza adresei

Futex-uri - specificaţii

- 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 condensata la futex

Implementare 2.5.7

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

Implementare 2.5.7 - op

- 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 trezeşte alt proces care aşteaptă pentru a decrementa futex-ul la -1; astfel va indica aşteptare
- op = FUTEX_UP
  - contorul futex-ului este pus pe 1
  - se trezeşte primul proces care aşteaptă

Probleme la implementarea 2.5.7

- nu există o implementare directă a pthread_cond_timedwait
  - operaţia necesită un timeout (timer)
- primitiva pthread_cond_broadcast trezeşte toate procesele care aşteaptă
  - în implementare un proces iese din kernel dacă obţine futex-ul sau dacă primeşte un semnal
- în implementări de thread-uri N:M este nevoie de o interfaţă asincronă pentru informaţii despre un futex (un proces poate avea mai multe thread-uri)
- poate apărea starvation
  - un proces renunţă la lock şi apoi doreşte achiziţionarea lui îl poate reachiziţiona rapid

Noua implementare

- sys_futex (struct futex *, int op, int val, struct timespec *reltime);
- op = FUTEX_WAIT
  - marcare proces ca INTERRUPTIBLE
  - citire valoare futex
  - valoare citită != val -> valoarea de retur = EWOULDBLOCK
  - altfel, sleep reltime sau nedefinit
  - procesul va fi trezit la o operaţie FUTEX_WAKE
- op = FUTEX_WAIT
  - 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 se îşi schimbă valoarea

Benchmarks

- Pentium III 500 Mhz, 256 MB, Ulockflex
- fiecare thread
  - calculează două numere aleatoare nlht, lht [0.5 .. 1.5]
  - achiziţionează 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 foloseşte şi o implementare cu spinlock-uri

Benchmarks (2)

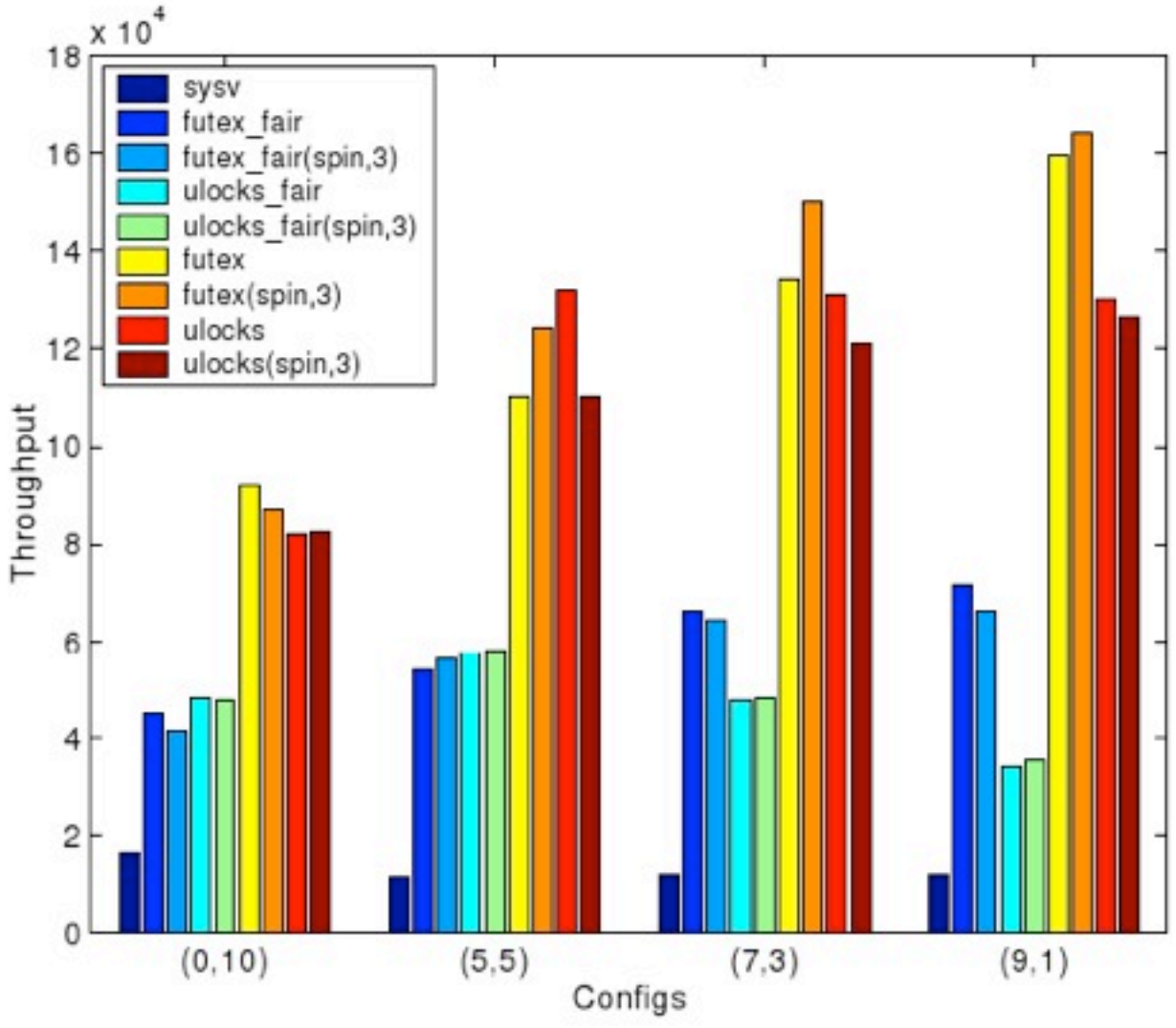


Figure 4: Throughput for various lock types for 100 tasks, 1 lock and 4 configurations

Benchmarks (3)

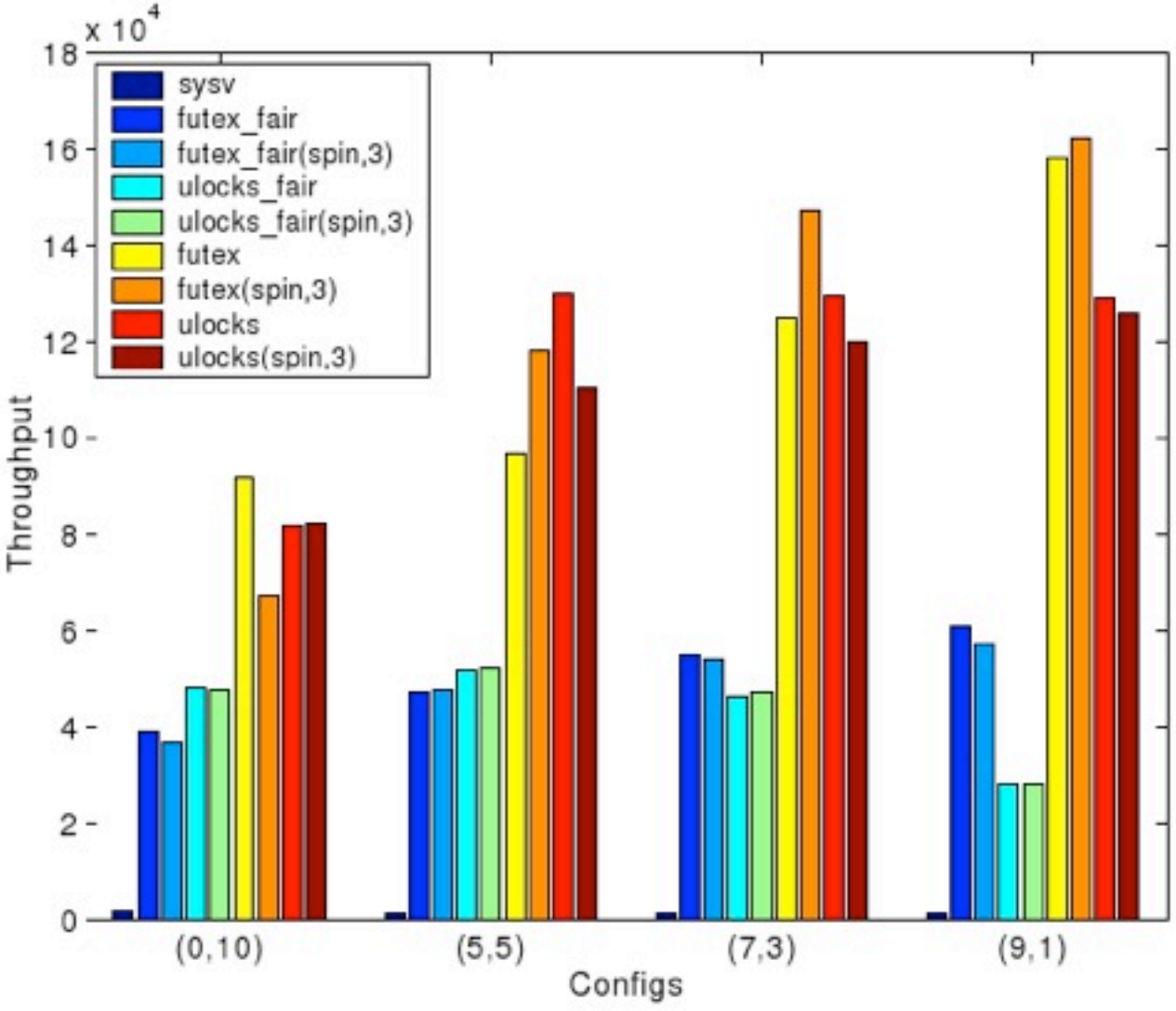


Figure 5: Throughput for various lock types for 1000 tasks, 1 lock and 4 configurations

Perspectiva userspace actuală

[/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 -> operaţia de efectuat (multiplexor)
- val -> valoare folosită la operaţii
- utime -> timeout
- addr2, val3 -> valori folosite de unele operaţii

Operaţii actuale

- FUTEX_WAIT
  - thread-ul aşteaptă 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 doreşte a fi trezite
  - de obicei este 1 sau INT_MAX
  - se întoarce numărul de thread-uri trezite
- FUTEX_WAKE_OP
  - folosită pentru implementarea variabilelor condiţie

Exemplu event

```
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;
};
```

Mutex1

```
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;
};
```

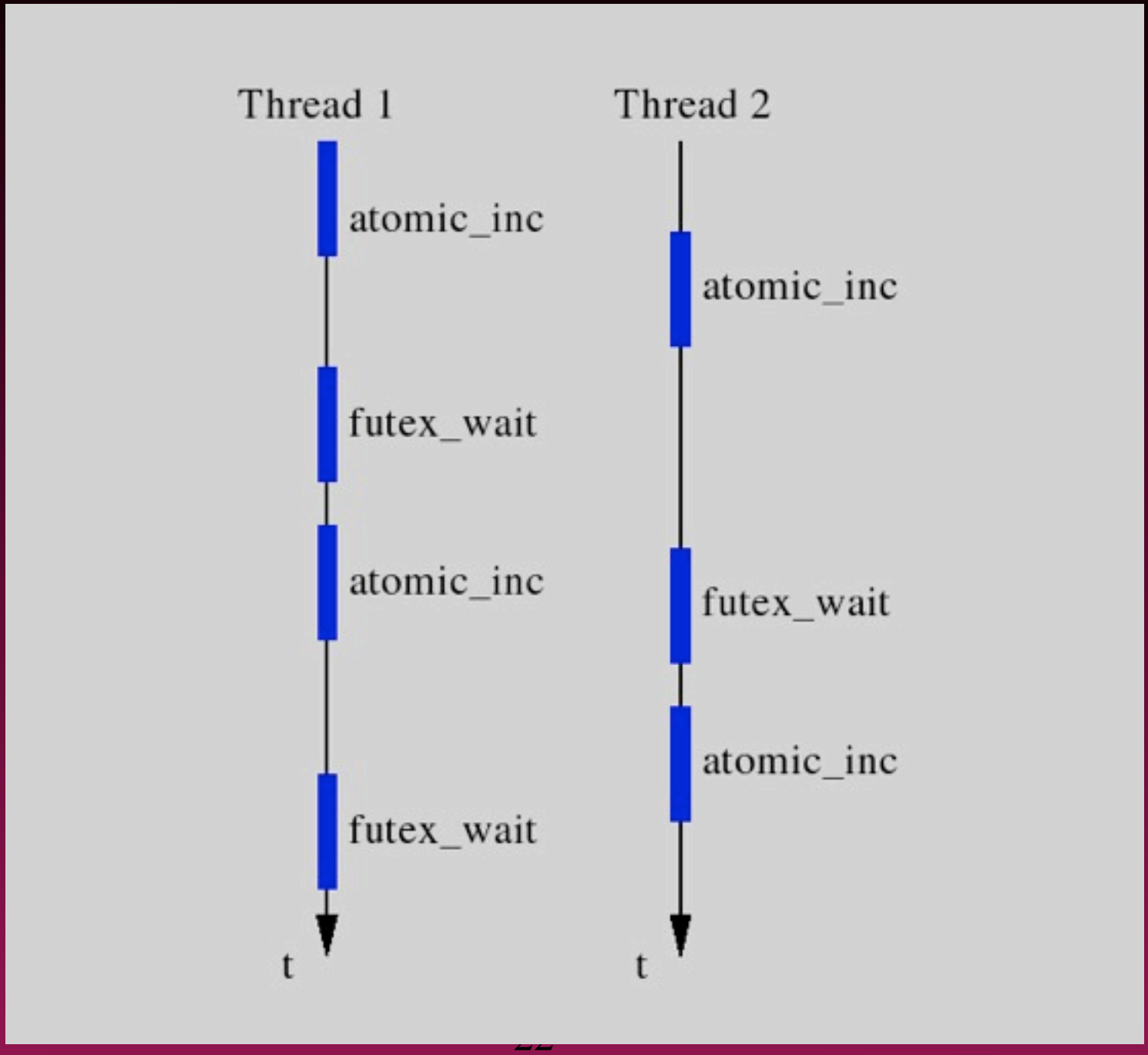
Mutex1

- 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 (operaţie atomică) – lock liber
  - se trezeşte un thread

Mutex1 (probleme)

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

Mutex1 (bug)


Thread 1
atomic_inc
futex_wait
atomic_inc
futex_wait
t
Thread 2
atomic_inc
futex_wait
atomic_inc
t

Mutex2

- fără overflow
- fără livelock cauzat de incrementarea valorii futexului
- trebuie evitate apelurile de sistem futex_wake cand nu există thread-uri care aşteaptă – 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)

Mutex2 (design)

```
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;
};
```

Contended/uncontended case

- 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

- contended
- mutex1
  - lock:
    - 1 (+1) op atomice
    - 1 (+1) syscalls
  - unlock
    - 0 op atomice
    - 1 apel de sistem
- mutex2
  - lock
    - 2 (+1) sau 3 (+2) op atomice
    - 1 (+1) syscalls
  - unlock
    - 1 op atomică
    - 1 apel de sistem

Mutex3

- Mutex2 este mai costisitor decât Mutex1 pentru cazul contended
- codul pentru Mutex2 este corect şi optimizează uncontended case
- unele arhitecturi deţin o instrucţiune atomică xchg (fără cmp)

Mutex3 - design

```
...
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);
    }
  }
}
...
```

Implementarea curentă

- [/usr/src/glibc/nptl/phtread_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

Implementare curentă

```
#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"); })
```

Resurse utile

- 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)