Synchronizacja w jądrze Linux

Michał Nazarewicz

Instytut Informatyki Politechniki Warszawskiej

3 kwietnia 2010

Po co synchronizować?

- Systemy z wieloma procesorami stają się powszechne.
- Współbieżność pozwala lepiej wykorzystać zasoby sprzętu.
- Stosowanie wątków może uprościś logikę aplikacji.
  - Jest to prawdziwe również w jądrze.
- Jeżeli wiele jednostek wykonania operuje na tych samym danych potrzebna jest synchronizacja.

Po co synchronizować?

- Systemy z wieloma procesorami stają się powszechne.
- Współbieżność pozwala lepiej wykorzystać zasoby sprzętu.
- Stosowanie wątków może uprościś logikę aplikacji.
  - Jest to prawdziwe również w jądrze.
- Jeżeli wiele jednostek wykonania operuje na tych samym danych potrzebna jest synchronizacja.

Wyścig

Sytuacja, w której co najmniej dwa logiczne konkteksty wykonania (procesy, zadania itp.) wykonują operację na zasobach dzielonych, a ostateczny wynik zależy od momentu realizacji.

za dr. Krukiem

Po co synchronizować?, kont.

Inkrementacja licznika

Wątek 1	**Wątek 2**
rejestr $\leftarrow$ licznik	
rejestr $\leftarrow$ rejestr $+ 1$	rejestr $\leftarrow$ licznik
licznik $\leftarrow$ rejestr	rejestr $\leftarrow$ rejestr $+ 1$
	licznik $\leftarrow$ rejestr

Po co synchronizować?, kont.

Inkrementacja licznika

Wątek 1	**Wątek 2**
rejestr $\leftarrow$ licznik	
rejestr $\leftarrow$ rejestr $+ 1$	rejestr $\leftarrow$ licznik
licznik $\leftarrow$ rejestr	rejestr $\leftarrow$ rejestr $+ 1$
	licznik $\leftarrow$ rejestr

Ogólny schemat odczyt-modyfikacja-zapis

Wątek 1	**Wątek 2**
odczyt	
modyfikacja	odczyt
zapis	modyfikacja
	zapis

Synchronizacja w kontekście jądra?

- W przestrzeni użytkownika też synchronizujemy.
- To już wszystko było!
- Ale...

- Każdy cykl zegara spędzony w jądrze to cykl stracony.
- Jednocześnie wszyscy korzystają z usług jądra.
- Stąd bardzo duży nacisk na wydajność.

- Jądro pracuje w wielu kontekstach.
- Przerwania przychodzą w dowolnym momencie.
- Jest wiele poziomów przerwań.

Mechanizmy nieblokujące

- Tradycyjna sekcja krytyczna zmusza wątki na czekanie przy wchodzeniu.
- Czekanie to... strata czasu.
- Lepiej nie czekać, niż czekać.
- Wiele algorytmów można zrealizować korzystając z „lekkich” mechanizmów synchronizacji, które nie blokują kontekstu wykonania.

Kiedy nie synchronizować?

- Istnieje duży nacisk na wydajność jądra.
- Dlatego niektóre operacje nie są synchronizowane.
- System zakłada, że jest to powinność użytkownika.

Kiedy nie synchronizować?, kod

Wywołanie systemowe `read(2)`

```
static ssize_t sys_read(unsigned fd, char __user * buf, size_t count)
{
    ssize_t ret = -EBADF;
    struct file * file;
    int fput_needed;

    file = fget_light(fd, &fput_needed);
    if (file) {
        loff_t pos = file->f_off;
        ret = vfs_read(file, buf, count, &pos);
        file->f_off = pos;
        fput_light(file, fput_needed);
    }

    return ret;
}
```

Bufor cykliczny

- Jeden producent.
- Jeden konsument.
- Jak obsługa klawiatury w DOS-ie.

Bufor cykliczny, kod

Bufor cykliczny, dane

```
static unsigned buffer[16], head, tail;
#define inc(v) (((v) + 1) % (sizeof buffer / sizeof *buffer))
```

Bufor cykliczny, funkcje

```
static int pop(unsigned *ret) {
    if (head != tail)
        return -ENOENT;

    *ret = buffer[tail];
    tail = inc(tail);
    return 0;
}
```

```
static int push(unsigned value) {
    if (inc(head) == tail)
        return -ENOSPC;
    buffer[head] = value;

    head = inc(head);
    return 0;
}
```

Bariery pamięci

- Procesor może przestawiać operacje zapisu i odczytu.
- Procesor może postrzegać operacje w losowej kolejności.
- Bariery wymuszają częściowy porządek operacji.

Bariery pamięci

- Procesor może przestawiać operacje zapisu i odczytu.
- Procesor może postrzegać operacje w losowej kolejności.
- Bariery wymuszają częściowy porządek operacji.

Bufor cykliczny, poprawiony

```
static int pop(unsigned *ret) {
    if (head != tail)
        return -ENOENT;
    smp_rmb();
    *ret = buffer[ tail ];
    tail = inc( tail );
    return 0;
}
```

```
static int push(unsigned value) {
    if (inc(head) == tail)
        return -ENOSPC;
    buffer[head] = value;
    smp_wmb();
    head = inc(head);
    return 0;
}
```

Bariery pamięci, kont.

Niepoprawny wielowątkowy singleton

```
struct foo * foo_singleton () {
    static struct mutex mutex;
    static struct foo *foo;
    if (!foo) {
        mutex_lock(&mutex);
        if (!foo) {
            struct foo *f = malloc(sizeof *f);
            init_foo (f);
            foo = f
        }
        mutex_unlock(&mutex);
    }
    return foo;
}
```

Zmienne atomowe

- Linux wyposażony jest w zmienne atomowe (atomic_t).
- Operacja na zmiennych atomowych są. . . atomowe.

Dostępne operacje na zmiennych atomowych

atomic_set	atomic_read	
atomic_add	atomic_add_return	atomic_add_negative
atomic_sub	atomic_sub_return	atomic_sub_and_test
atomic_inc	atomic_inc_return	atomic_inc_and_test
atomic_dec	atomic_dec_return	atomic_dec_and_test
	atomic_add_unless	atomic_inc_not_zero
	atomic_xchg	atomic_cmpxchg

Licznik referencji

- W oparciu o zmienne atomowe zaimplementowany jest licznik referencji (struct kref).
- Dostępne operacje:
  - kref_set,
  - kref_init,
  - kref_get oraz
  - kref_put.

Licznik referencji, kod

Obiekt współdzielony

```
struct foo {
    struct kref ref;
    /* ... */
};

struct foo *new_foo(/* ... */) {
    struct foo *foo =
        kmalloc(sizeof *foo,
                GFP_KERNEL);
     kref_init (&foo->ref);
    /* ... */
    return foo;
}
```

```
void foo_get(struct foo *foo) {
    kref_get (&foo->ref);
}

void foo_release (struct kref *ref) {
    struct foo *foo =
        container_of (ref, struct
            foo, ref);
    /* ... */
    kfree(foo);
}

void foo_put(struct foo *foo) {
    kref_put (&foo->ref,
        foo_release );
}
```

Zmienne atomowe w przestrzeni użytkownika

```
basic_string assign(const basic_string &__str) {
    _CharT *__tmp = __str._M_rep()->_M_grab();
    _M_rep()->_M_dispose();
    _M_dataplus._M_p = __tmp;
    return *this;
}

_CharT *_M_grab() {
    if (_M_refcount >= 0) {
        __sync_fetch_and_add(&_M_refcount, 1);
        return _M_refdata();
    } else {
        _Rep *__r = _Rep::_S_create(_M_length, _M_capacity);
        _M_copy(__r->_M_refdata(), _M_refdata(), _M_length);
        _M_refcount = 0;
        return __r->_M_refdata();
    }
}

void _M_dispose() {
    if (__sync_fetch_and_add(&_M_refcount, -1) <= 0) _M_destroy();
}

void _M_leak() {
    if (_M_rep()->_M_refcount > 0) _M_mutate(0, 0, 0);
    _M_rep()->_M_refcount = -1;
}
```

Operacje na bitach

- Podobnym mechanizmem są atomowe operacja na bitach.
- Nadają się idealnie, jeżeli chcemy wykonać coś raz.

Dostępne operacje na bitach

set_bit	test_and_set_bit
clear_bit	test_and_clear_bit
change_bit	test_and_change_bit

Operacje na bitach, kod

Tasklety

```
static inline void tasklet_schedule(struct tasklet_struct *t) {
    if (! test_and_set_bit (TASKLET_STATE_SCHED, &t->state))
        __tasklet_schedule (t);
}

static void tasklet_action (struct softirq_action *a) {
    /* ... */
    if (! test_and_set_bit (TASKLET_STATE_RUN, &t->state)) {
        clear_bit (TASKLET_STATE_SCHED, &t->state);
        t->func(t->data);
        smp_mb__before_clear_bit ();
        clear_bit (TASKLET_STATE_RUN, &t->state);
    }
    /* ... */
}
```

Porównaj i zamień

Atomowe operacje porównaj i zamień

xchg	atomic_xchg
cmpxchg	atomic_cmpxchg

Porównaj i zamień

Atomowe operacje porównaj i zamień

xchg atomic_xchg
cmpxchg atomic_cmpxchg

Operacje na stosie

```
static void
push(struct node ** firstp,
     struct node *n) {
    struct node *f;
    do {
        f = *firstp;
        n->next = f;
    } while (cmpxchg(firstp, f, n)
             != f);
}
```

```
static struct node *
pop(struct node ** firstp) {
    struct node *f;
    do {
        f = *first;
    } while (f && cmpxchg(firstp,
                 f, f->next) != f);
    return f;
}
```

Mechanizmy blokujące

- Proste atomowe operacje często nie wystarczają.
- Zazwyczaj trzeba modyfikować wiele zmiennych „na raz”.
- Często trzeba również czekać na wystąpienie pewnych warunków.
- W takich sytuacjach sprawdzają się mechanizmy blokujące.

Kolejka oczekiwania

- Podstawowym mechanizmem blokującym jest kolejka oczekiwania (wait_queue_head_t), która
- służy do tworzenia listy, w której zadania oczekują, aż
- inne zadania je obudzą.
- Zazwyczaj korzysta się z niej poprzez interfejs wait_event (i warianty).

Kolejka oczekiwania, kod

Blokujący bufor cykliczny, dane

```
static DECLARE_WAIT_QUEUE_HEAD(push_queue);
static DECLARE_WAIT_QUEUE_HEAD(pop_queue);
unsigned empty = sizeof buffer / sizeof *buffer, full;
```

Blokujący bufor cykliczny, funkcje

```
static int pop(unsigned *ret) {
    wait_event(pop_queue, !full);
    --full;
    *ret = buffer[tail];
    tail = inc(tail);
    ++empty; wake_up(push_queue);
    return 0;
}
```

```
static int push(unsigned value) {
    wait_event(push_queue, !empty);
    --empty;
    buffer[head] = value;
    head = inc(head);
    ++full; wake_up(pop_queue);
    return 0;
}
```

Spanie a sygnały

- Funkcja wait_event blokuje sygnały!
- Procesu blokującego sygnały nie da się zabić!
- Lepiej użyć wait_event_interruptible.

Spanie a sygnały

- Funkcja wait_event blokuje sygnały!
- Procesu blokującego sygnały nie da się zabić!
- Lepiej użyć wait_event_interruptible.

Kolejka oczekiwania, wersja `interruptible`

```
static int pop(unsigned *ret) {
    if ( wait_event_interruptible
         (pop_queue, ! full ))
         return -EINTR;
    -- full ;
    *ret = buffer[ tail ];
     tail = inc( tail );
    ++empty; wake_up(push_queue);
    return 0;
}
```

```
static int push(unsigned value) {
    if ( wait_event_interruptible
         (push_queue, !empty))
         return -EINTR;
    --empty;
     buffer [head] = value;
    head = inc(head);
    ++full; wake_up(pop_queue);
    return 0;
}
```

Semafory

- Użyty mechanizm to klasyczny semafor.
- Semafory są rzecz jasna dostępne w jądrze.
- Dostępne operacje:
  - sema_init,
  - down, down_trylock,
  - down_interruptible, down_killable, down_timeout oraz
  - up.

Semafor

Obiekt inicjowany nieujemną liczbą całkowitą, na której zdefiniowane są dwie niepodzielne operacje:

- `down(sem) { while (!sem); --sem; }`.
- `up(sem) { ++sem; }` i

za dr. Krukiem

Semafory, kod

Bufor cykliczny na semaforach, dane

```
static struct semaphore push_sem, pop_sem;
static void init (void) {
    sema_init(&push_sem, sizeof buffer / sizeof *buffer);
    sema_init(&pop_sem, 0);
}
```

Bufor cykliczny na semaforach, funkcje

```
static int pop(unsigned *ret) {
    if ( down_interruptible
        (pop_sem))
        return -EINTR;
    *ret = buffer[ tail ];
    tail = inc( tail );
    up(push_sem);
    return 0;
}
```

```
static int push(unsigned value) {
    if ( down_interruptible
        (push_sem))
        return -EINTR;
    buffer [head] = value;
    head = inc(head);
    up(pop_sem);
    return 0;
}
```

Muteksy

- Pewnym szczególnym rodzajem semaforów są muteksy.
- Z założenia stworzone do implementowania sekcji krytycznej.
- Dostępne operacje:
  - mutex_is_locked,
  - mutex_lock i wiarianty
    - plus wersje . . . _nested,
  - mutex_try_lock,
  - mutex_unlock.

Muteksy, kod

Kolejka chroniona muteksem

```
static DEFINE_MUTEX(mutex), LIST_HEAD(first);

static int push(struct list_head *n) {
    if ( mutex_lock_interruptible (&mutex)) return -EINTR;
     list_add_tail (& first , n);
    mutex_unlock(&mutex);
    return 0;
}

static struct list_head *pop(void) {
    struct list_head *ret = NULL;
    if ( mutex_lock_interruptible (&mutex)) return PTR_ERR(-EINTR);
    if (! list_empty (& first )) {
        ret = first .next;
         list_del (ret);
    }
    mutex_unlock(&mutex);
    return ret;
}
```

Muteksy, „nested”?

Zakleszczenie

Pojęcie opisujące sytuację, w której co najmniej dwie różne akcje czekają na siebie nawzajem, więc żadna nie może się zakończyć.

za Wikipedią

Muteksy, „nested”?

Zakleszczenie

Pojęcie opisujące sytuację, w której co najmniej dwie różne akcje czekają na siebie nawzajem, więc żadna nie może się zakończyć.

za Wikipedią

- Linux ma mechanizm weryfikacji synchronizacji.
- Dzieli muteksy itp. na klasy.
- Sprawdza, czy klasy są blokowane w tej samej kolejności.
- Nie można zablokować dwóch muteksów z tej samej klasy.
- Co w takim razie z muteksami w hierarchii?
- Po to jest właśnie mutex_lock_nested itp.

„Nieśpiące” mechanizmy

- Przedstawione dotychczas mechanizmy blokujące mogą przełączyć zadanie w stan uśpienia.
- W pewnych kontekstach spanie jest niedopuszczalne.
  - Wewnątrz funkcji obsługi przerwania.
  - Gdy przerwania są wyłączone.
  - Ogólnie w kontekście atomowym.
- Ale na szczęście są inne mechanizmy.

Wyłączanie przerwań

- Będąc w kodzie jądra możemy wyłączyć przerwania.
- local_irq_diable wyłącza, a local_irq_enable włącza przerwania.
- Ale co jeśli w momencie wejścia do sekcji krytycznej przerwania już były wyłączone?
- Na pomoc przychodzą local_irq_save i local_irq_restore.
- Trzeba uważać – pewne funkcje mogą spowodować przełączenie kontekstu nawet jeżeli przerwania są wyłączone.
- A poza tym, wyłączanie przerwań jest lokalne dla procesora.

Wyłączanie wywłaszczania

- Linux wspiera wywłaszczanie wewnątrz kodu jądra.
- Zmniejsza to czas reakcji, ale
- komplikuje synchronizację.
- Można ten mechanizm wyłączyć.
- Dostępne operacje preempt_disable i preempt_enable
- są rekurencyjne,
- ale lokalne dla konkretnego procesora.

Spinlock

- Spinlock działa jak muteks, ale
- stosuje aktywne oczekiwanie, dzięki czemu
- nie zasypia.
- Użycie: spin_lock ... spin_unlock.
  - Też mają wersję ..._nested.

Spinlock, kod

Kolejka chroniona spinlockiem

```
static DEFINE_SPINLOCK(lock), LIST_HEAD(first);

static void push(struct list_head *n) {
    spin_lock(&lock);
     list_add_tail (& first , n);
    spin_unlock(&lock);
}

static struct list_head *pop(void) {
    struct list_head *ret = NULL;
    spin_lock(lock);
    if (! list_empty (& first )) {
        ret = first .next;
         list_del (ret);
    }
    spin_unlock(&lock);
    return ret;
}
```

Spinlock a przerwania

- Co jeśli w sekcji krytycznej przyjdzie przerwanie?

Przerwanie w sekcji krytycznej

Kontekst użytkownika		**Przerwanie**
spin_lock		
	przychodzi przerwanie	
		spin_lock
		zakleszczenie

- Gdy spinlock używany jest również w przerwaniu, w kontekście użytkownika trzeba przerwania wyłączyć (spin_lock_irqsave . . . spin_unlock_irqrestore).
  - Jest też wersja spin_lock_irq . . . spin_unlock_irq

Spinlock a przerwania, kod

Kolejka chroniona spinlockiem, poprawiona

```
static void push(struct list_head *n) {
    unsigned long flags;
     spin_lock_irqsave(&lock, flags);
      list_add_tail(&first, n);
      spin_unlock_irqrestore(&lock, flags);
}

static struct list_head *pop(void) {
    struct list_head *ret = NULL;
    unsigned long flags;
     spin_lock_irqsave(&lock, flags);
     if (!list_empty(&first)) {
         ret = first.next;
          list_del(ret);
     }
      spin_unlock_irqrestore(&lock, flags);
     return ret;
}
```

Podsumowanie

- Linux jest bardzo złożonym oprogramowaniem.
- Wiele mechanizmów jest znanych z przestrzeni użytkownika.
- Nacisk na wydajność i przerwania komplikują sytuację.
- Dlatego jest wiele mechanizmów unikalnych dla jądra.
- Czyni to synchronizację w kernelu bardzo ciekawą!
- Ale i złożoną i trudną.

Czego nie było...

- Siedem rodzai barier pamięci.
- Big Kernel Lock.
- Kolejka oczekiwań to coś o wiele więcej niż wait_event.
- spin_lock_bh.
- Zmienne *per CPU*.
- Alokacja pamięci a synchronizacja.

Skąd czerpać informacje?

- Use the Source, Luke.
- Podkatalog `Documentation` w źródlach.
- Publiczne serwery LXR.
  - http://lxr.linux.no/
- LWN.net
- STER z dr. W. Zabołotnym
- *Linux Device Drivers, Third Edition*
  - http://lwn.net/Kernel/LDD3/
- Daniel P. Bovet, Marco Cesati. *Understanding the Linux Kernel*.

Dziękuję za uwagę!

- Pytania?
- Opinie?
- Sugestie?

- Michał Nazarewicz
- http://mina86.com/
- mina86@mina86.com
- mina86@jabber.org