

Примитивы синхронизации

Калишенко Е.Л.
2017

Попытка № 1

```
class LockOne implements Lock {  
    // thread-local index, 0 or 1  
    private boolean[] flag = new boolean[2];  
  
    public void lock() {  
        int i = ThreadID.get();  
        int j = 1 - i;  
        flag[i] = true;  
        while (flag[j]) {} // wait  
    }  
  
    public void unlock() {  
        int i = ThreadID.get();  
        flag[i] = false;  
    }  
}
```

Попытка № 2

```
class LockTwo implements Lock {  
    private volatile int victim;  
  
    public void lock() {  
        int i = ThreadID.get();  
        victim = i;  
        // let the other go first  
        while (victim == i) {} // wait  
    }  
  
    public void unlock() {}  
}
```

Mutex для 2 потоков

```
class Peterson implements Lock {  
    // thread-local index, 0 or 1  
    private volatile boolean[] flag = new boolean[2];  
    private volatile int victim;  
  
    public void lock() {  
        int i = ThreadID.get();  
        int j = 1 - i;  
        flag[i] = true; // I'm interested  
        victim = i; // you go first  
        while (flag[j] && victim == i) {}; // wait  
    }  
  
    public void unlock() {  
        int i = ThreadID.get();  
        flag[i] = false; // I'm not interested  
    }  
}
```

Алгоритм булочника

```
class Bakery implements Lock {
    boolean[] flag;
    Label[] label;
    public Bakery (int n) {
        flag = new boolean[n];
        label = new Label[n];
        for (int i = 0; i < n; i++) {
            flag[i] = false; label[i] = 0;
        }
    }
    public void lock() {
        int i = ThreadID.get();
        flag[i] = true;
        label[i] = max(label[0], ..., label[n-1]) + 1;
        while (( $\exists k \neq i$ )(flag[k] && (label[k], k) <<
(label[i], i))) {}
    }
    public void unlock() {
        flag[ThreadID.get()] = false;
    }
}
```

POSIX mutex

Тип: `pthread_mutex_t`

Жизнь:

- инициализация:
`pthread_mutex_init` или `PTHREAD_MUTEX_INITIALIZER`
- запрос на монопольное использование:
`pthread_mutex_lock` или
`pthread_mutex_timedlock`
- отказ от монопольного использования:
`pthread_mutex_unlock`
- тестирование монопольного использования:
`pthread_mutex_trylock`
- разрушение:
`pthread_mutex_destroy`

mutex && recursive_mutex

Функции:

- *Захват:* `void lock();`
- *Попытаться захватить:* `bool try_lock();`
- *Освободить:* `void unlock();`

timed_mutex && recursive_timed_mutex

Функции:

- То же, что у *mutex* && *recursive_mutex*
 - Захват: `void lock();`
 - Попытаться захватить: `bool try_lock();`
 - Освободить: `void unlock();`
- + Захват с ограничением: `timed_lock(...);`

shared_mutex

Функции:

- То же, что у *[recursive_]timed_mutex*
 - Захват: `void lock();`
 - Попытааться захватить: `bool try_lock();`
 - Освободить: `void unlock();`
 - Захват с ограничением: `void timed_lock(...);`
- + Захват на чтение: `void [timed_]lock_shared();`
- + Захват на чтение с возможностью «дозахвата» на запись: `void lock_upgrade();`

spin_mutex

Функции:

- То же, что у *timed_mutex*
 - Захват: `void lock();`
 - Попытаться захватить: `bool try_lock();`
 - Освободить: `void unlock();`
 - Захват с ограничением: `void timed_lock(...);`

Отличие:

- Активное ожидание на захвате

Релизация spin_mutex

```
inline void spin_mutex::lock(void)
{
    do{
        boost::uint32_t prev_s =
            ipcdetail::atomic_cas32(const_cast<boost::uint32_t*>(&m_s), 1, 0);

        if (m_s == 1 && prev_s == 0){
            break;
        }
        // relinquish current timeslice
        ipcdetail::thread_yield();
    }while (true);
}
```

CAS-операции

CAS — compare-and-set, compare-and-swap

`bool compare_and_set(`

- `int*` `<адрес переменной>`,
- `int` `<старое значение>`,
- `int` `<новое значение>`)

- ✓ Возвращает признак успешности операции установки значения
- ✓ Атомарна на уровне процессора (CPU: i486+):

`cmpxchg`

Преимущества CAS

- Является аппаратным примитивом
- Возможность продолжения захвата примитива без обязательного перехода в режим «ожидания»
- Меньше вероятность возникновения блокировки из-за более мелкой операции
- Более быстрая (правда не в условиях жёсткой конкуренции)

Пример CAS инкремента

```
/**
 * Atomically increments by one the current value.
 * @return the updated value
 */
public final int incrementAndGet() {
    for (;;) {
        int current = get();
        int next = current + 1;
        if (compareAndSet(current, next))
            return next;
    }
}
```

Мьютексы

На примере boost:

- mutex
- recursive_mutex
- timed_mutex
- recursive_timed_mutex
- shared_mutex
- spin_mutex

lock_guard

- Захват в конструкторе
- Освобождение в деструкторе
- Нельзя освободить до разрушения
- Используются методы мьютексов
 - *Захват:* `void lock();`
 - *Освободить:* `void unlock();`

Замки

- lock_guard
- unique_lock
- shared_lock
- upgrade_lock
- upgrade_to_unique_lock

RAII

- Мьютекс имеет свой `typedef` на `scoped_lock`:

- mutex:

```
typedef unique_lock<mutex> scoped_lock;
```

- recursive_mutex:

```
typedef unique_lock<recursive_mutex> scoped_lock;
```

- timed_mutex:

```
typedef unique_lock<timed_mutex> scoped_timed_lock;  
typedef scoped_timed_lock scoped_lock;
```

- Удобнее захватывать:

```
boost::mutex::scoped_lock l(m);
```

unique_lock

- То же, что lock_guard
- + Используются методы мьютексов
 - *Попытаться захватить:* `bool try_lock();`
 - *Захват с ограничением:* `void timed_lock(...);`
- + Дополнительные функции получения мьютекса, проверки «захваченности»...

shared_lock

- Предназначени для работы с shared_mutex
- Захват на чтение
- Освобождение в деструкторе
- Используются методы мьютексов
 - *Захват:* `void [timed_]lock_shared();`
 - *Освободить:* `void unlock_shared();`

upgrade_lock

- Предназначени для работы с shared_mutex
- Захват на чтение с возможностью «дозахвата» на запись
- Освобождение в деструкторе
- Используются методы мьютексов
 - *Захват:* `void lock_upgrade();`
 - *Освободить:* `void unlock_upgrade();`

upgrade_to_unique_lock

- Предназначени для работы с upgrade_lock
- Захват на запись после захвата на чтение
- Освобождение в деструкторе
- Используются методы мьютексов
 - *Захват:* `void unlock_upgrade_and_lock();`
 - *Освободить:* `void unlock_and_lock_upgrade();`

Futex

Futex - 'Fast Userspace muTexes'

- Применяются для реализации POSIX mutex
- Доступен с ядра 2.5.40
- В реализации используется с CAS — некоторые операции проводятся в пространстве пользователя

Реализация futex

```
1355     raw_spin_lock(&pi_state->owner->pi_lock);
1356     WARN_ON(list_empty(&pi_state->list));
1357     list_del_init(&pi_state->list);
1358     raw_spin_unlock(&pi_state->owner->pi_lock);
1359
1360     raw_spin_lock(&new_owner->pi_lock);
1361     WARN_ON(!list_empty(&pi_state->list));
1362     list_add(&pi_state->list, &new_owner->pi_state_list);
1363     pi_state->owner = new_owner;
1364     raw_spin_unlock(&new_owner->pi_lock);
1365
1366     raw_spin_unlock_irq(&pi_state->pi_mutex.wait_lock);
1367
1368     deboost = rt_mutex_futex_unlock(&pi_state->pi_mutex, &wake_q);
1369
1370     /*
1371     * First unlock HB so the waiter does not spin on it once he got woken
1372     * up. Second wake up the waiter before the priority is adjusted. If we
1373     * deboost first (and lose our higher priority), then the task might get
1374     * scheduled away before the wake up can take place.
1375     */
1376     spin_unlock(&hb->lock);
1377     wake_up_q(&wake_q);
```


Барьер для N потоков

```
#define SYNC_MAX_COUNT 10
```

```
void SynchronizationPoint() {  
    static mutex_t sync_lock = PTHREAD_MUTEX_INITIALIZER;  
    static cond_t sync_cond = PTHREAD_COND_INITIALIZER;  
    static int sync_count = 0;  
  
    /* блокировка доступа к счетчику */  
    pthread_mutex_lock(&sync_lock);  
    sync_count++;  
  
    /* проверка: следует ли продолжать ожидание */  
    if (sync_count < SYNC_MAX_COUNT)  
        pthread_cond_wait(&sync_cond, &sync_lock);  
    else  
        /* оповестить о достижении данной точки всеми */  
        pthread_cond_broadcast(&sync_cond);  
  
    /* активизация взаимной блокировки - в противном случае  
       из процедуры сможет выйти только одна нить! */  
    pthread_mutex_unlock(&sync_lock);  
}
```

Условные переменные

- Нужны как механизм взаимодействия потоков, в отличие от мьютексов
- Всегда используется с мьютексом
- Предназначена для уведомления события
- Атомарно освобождает мьютекс при `wait()`
- Хорошо подходит для задач типа «производитель-потребитель»
- Для boost: `boost::condition`

Пример использования

```
public void prepareData() {  
    synchronized (monitor) {  
        System.out.println("Data prepared");  
        ready = true;  
        monitor.notifyAll();  
    }  
}
```

```
public void sendData() {  
    synchronized (monitor) {  
        System.out.println("Waiting for data...");  
        while (!ready) {  
            try {  
                monitor.wait();  
            } catch (InterruptedException e) {  
                e.printStackTrace();  
            }  
        }  
        System.out.println("Sending data...");  
    }  
}
```

Число ядер в системе

Boost:

boost::thread::hardware_concurrency()

Java:

Runtime.getRuntime().availableProcessors()