

# Примитивы синхронизации

# МЬЮТЕКСЫ

## ***На примере boost:***

- mutex
- recursive\_mutex
- timed\_mutex
- recursive\_timed\_mutex
- shared\_mutex
- spin\_mutex

# 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();`
  - *Попытаться 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);
}
```

# Замки

- lock\_guard
- unique\_lock
- shared\_lock
- upgrade\_lock
- upgrade\_to\_unique\_lock



# lock\_guard

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

# 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

- Предназначены 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

- Предназначены 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

- Предназначены 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();`

# Применение замков

- 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);
```

# 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+):  
`std::atomic::compare_exchange_strong`

# 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;
    }
}
```

# Futex

*Futex - 'Fast Userspace muTexes'*

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

# Реализация futex

```
174 static void
175 futexunlock(Lock *l)
176 {
177     uint32 v;
178
179     v = runtime·xchg(&l->key, MUTEX_UNLOCKED);
180     if(v == MUTEX_UNLOCKED)
181         runtime·throw("unlock of unlocked lock");
182     if(v == MUTEX_SLEEPING)
183         futexwakeupt(&l->key, 1);
184 }
```

# Критическая секция

- Разница с мьютексом во многом терминологическая
- Критическая секция — **не** объект ядра ОС
- Использование аналогична `pthread_mutex_t`
  - `InitializeCriticalSection`
  - `::EnterCriticalSection(&m_lock);`
  - `::LeaveCriticalSection(&m_lock);`
- Для удобства также как и с мьютексами используются замки: `CScopeLock...`

# Interlocked-функции

- Тоже работают в пространстве пользователя, не переводя процесс в режим ожидания, так как основаны на CAS-операциях
- Примеры:
  - `InterlockedIncrement(&var)`
  - `InterlockedExchange`
  - `InterlockedCompareExchange`
  - ...

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

# Графические библиотеки

- Имеют цикл обработки событий
- Физически || исполнения нет — обход синхронизации
- Если графика не нагружена и есть всего 1 вычислительно сложный поток — можно использовать «частые события». На Qt, например:
  - `Qtimer timer;`
  - `connect(&timer,SIGNAL(timeout()),this,SLOT(step()));`
  - `timer.start(0);`



# Оптимальное число потоков

Boost:

*`boost::thread::hardware_concurrency()`*

Java:

*`Runtime.getRuntime().availableProcessors()`*