Многопоточное программирование (часть 2)

Михаил Георгиевич Курносов

Email: mkurnosov@gmail.com

WWW: http://www.mkurnosov.net

Курс «Параллельные и распределённые вычисления»

Школа анализа данных Яндекс (Новосибирск)

Весенний семестр, 2016

Конкурентный доступ к разделяемой структуре данных

```
void handler(int& counter)
{
    for (int i = 0; i < 10000; ++i) {
        counter++;
    }
}

int main()
{
    int counter = 0;

    std::thread t1(handler, std::ref(counter));
    std::thread t2(handler, std::ref(counter));
    t1.join();
    t2.join();

    std::cout << "Counter = " << counter << std::endl;
    return 0;
}
```

Конкурентный доступ к разделяемой структуре данных

```
void handler(int& counter)
{
    for (int i = 0; i < 10000; ++i) {
        counter++;                          // Data race
    }
}

int main()
{
    int counter = 0;

    std::thread t1(handler, std::ref(counter));
    std::thread t2(handler, std::ref(counter));
    t1.join();
    t2.join();

    std::cout << "Counter = " << counter << std::endl;
    return 0;
}
```

- **Ожидаемое значение 20 000**
- **Результат запусков**
  - Counter = 19747
  - Counter = 19873
  - Counter = 19813
  - …

Организация критической секции

```
int lock = 0;

void handler(int& counter)
{
    for (int i = 0; i < 10000; ++i) {

        do {                        // Ожидаем освобождение блокировки
            if (lock == 0) {
                lock = 1;           // Захватываем блокировку
                break;
            }
        } while (1)

         counter++;

         lock = 0;                  // Освобождаем блокировку
    }
}
```

Intel 64 atomic operations (Intel ASDM, Vol.3, Ch.8)

Guaranteed atomic operations

- Reading/writing a byte
- Reading/writing a word aligned on a 16-bit boundary
- Reading/writing a doubleword aligned on a 32-bit boundary
- Reading/writing a quadword aligned on a 64-bit boundary
- 16-bit accesses to uncached memory locations that fit within a 32-bit data bus
- Unaligned 16-, 32-, and 64-bit accesses to cached memory that fit within a cache line

```
int flag;
// ...
flag = 1
```

ISO C/C++ scalar types alignment (int, char, …)
Linux ABI // http://www.x86-64.org/documentation/abi.pdf

Locked atomic operations (locking system bus, prefix #LOCK)

- The bit test and modify instructions: BTS, BTR, and BTC
- The exchange instructions: XADD, CMPXCHG, and CMPXCHG8B
- The LOCK prefix is automatically assumed for XCHG instruction
- The following single-operand arithmetic and logical instructions: INC, DEC, NOT, and NEG
- The following two-operand arithmetic and logical instructions: ADD, ADC, SUB, SBB, AND, OR, and XOR

```
// int counter; counter += val
   lock xaddl counter, val
```

Intel® 64 and IA-32 Architectures Software Developer's Manual, Vol. 3A: Chapter 8. Multiple-processor management
http://download.intel.com/products/processor/manual/325384.pdf

Cache locking (using #LOCK prefix)

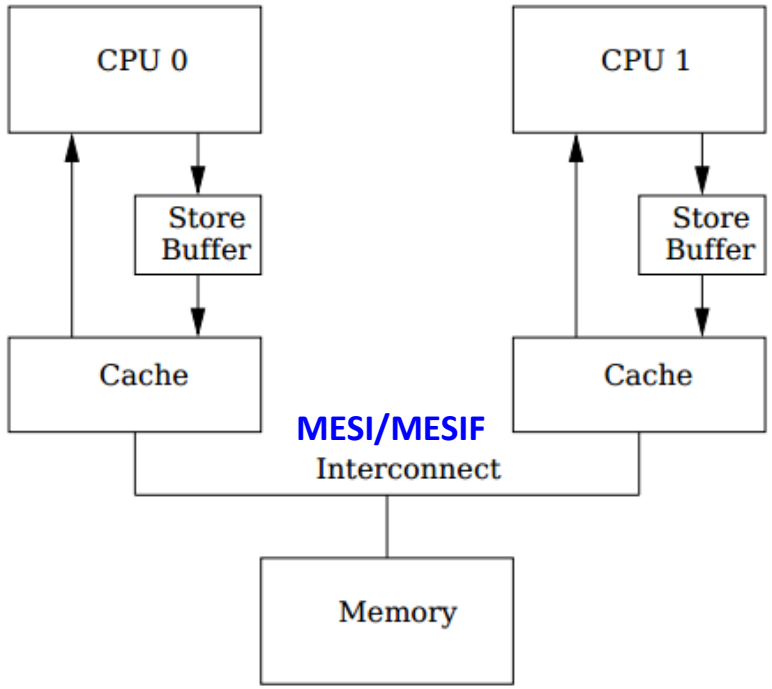
- If <area of memory is cached> then
  // No bus locking!
  Modify data in cache line
- Cache coherency mechanism ensures atomicity (MESI, MESIF)

Atomic test_and_set

```
int lock = 0;

void handler(int& counter)
{
    for (int i = 0; i < 10000; ++i) {

        while (test_and_set(&lock) == 1) // Ожидаем освобождение блокировки
            ;

        counter++;

        lock = 0;                        // Освобождаем блокировку
    }
}
```

Модель согласованности памяти

- **Операции чтения и записи данных в память могут выполняться в порядке отличном от исходного**
  - Иерархия кеш-памяти + протоколы поддержания когерентностей кешей (MESI/MESIF)
  - Store buffers – очереди записи (сокрытие латентности MESI)
  - Внеочередное выполнение команд



- Paul E. McKenney. **Memory Barriers: a Hardware View for Software Hackers** // http://www.rdrop.com/users/paulmck/scalability/paper/whymb.2010.06.07c.pdf

Модель согласованности памяти

Type	Alpha	ARMv7	POWER	SPARC PSO	x86	x86 oostore	AMD64	IA-64	zSeries
Loads reordered after loads	Y	Y	Y			Y		Y	
Loads reordered after stores	Y	Y	Y			Y		Y	
Stores reordered after stores	Y	Y	Y	Y		Y		Y	
Stores reordered after loads	Y	Y	Y	Y	Y	Y	Y	Y	Y
Atomic reordered with loads	Y	Y	Y					Y	
Atomic reordered with stores	Y	Y	Y	Y				Y	
Dependent loads reordered	Y								
Incoherent Instruction cache pipeline	Y	Y	Y	Y	Y	Y		Y	Y

❑ Paul E. McKenney. **Memory Barriers: a Hardware View for Software Hackers** // http://www.rdrop.com/users/paulmck/scalability/paper/whymb.2010.06.07c.pdf

Memory ordering Intel64 (Core 2 Duo)

Chapter 8.2 [1]: In a single-processor system for memory regions defined as write-back cacheable, the memory-ordering model respects the following principles:

- Reads are not reordered with other reads.
- Writes are not reordered with older reads.
- Writes to memory are not reordered with other writes, with the following exceptions: writes executed with the CLFLUSH instruction; streaming stores (writes) executed with the non temporal move instructions (MOVNTI, MOVNTQ, MOVNTDQ, MOVNTPS, and MOVNTPD); and string operations (see Section 8.2.4.1).
- Reads may be reordered with older writes to different locations but not with older writes to the same location.
- Reads or writes cannot be reordered with I/O instructions, locked instructions, or serializing instructions.
- Reads cannot pass earlier LFENCE and MFENCE instructions.
- Writes cannot pass earlier LFENCE, SFENCE, and MFENCE instructions.
- LFENCE instructions cannot pass earlier reads.
- SFENCE instructions cannot pass earlier writes.
- MFENCE instructions cannot pass earlier reads or writes.

[1] Intel® 64 and IA-32 Architectures Software Developer's Manual Combined Volumes 3A, 3B, and 3C: System Programming Guide // http://download.intel.com/products/processor/manual/325384.pdf

Последовательная согласованность (Sequential consistency)

- Операции с памятью (load, store) выполняются в программном порядке (исходном) – их относительный порядок не должен меняться
- Модель системы: нет кеш-памяти, нет буферов записи (store buffers)

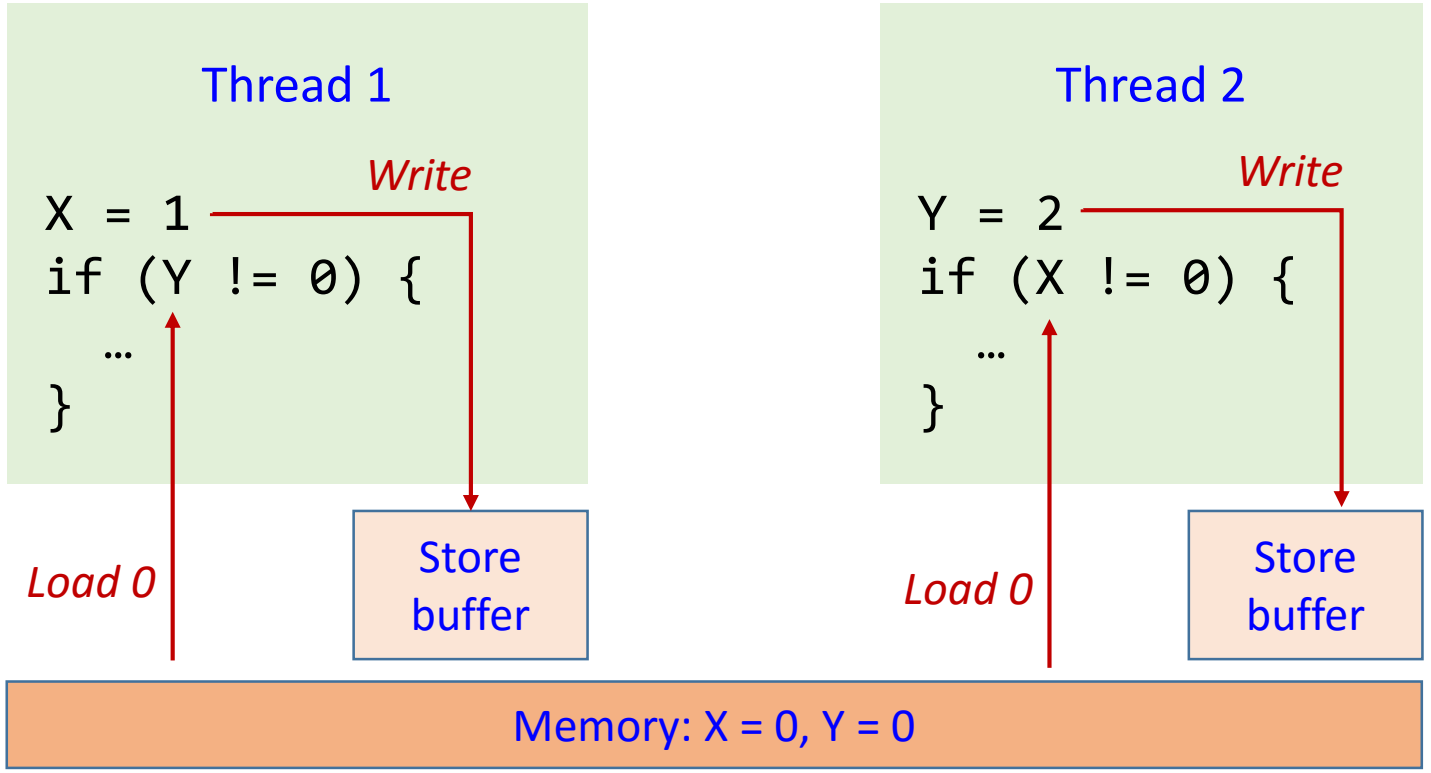
Thread 1	Thread 2
X = 5 Z = Y	C = A B = 2

Разрешены любые сценарии выполнения потоков, но недопустимо менять относительный порядок выполнения X = 5 и Z = Y, а также C = A и B = 2

Проста для понимания, ограничивает потенциальные возможности процессора

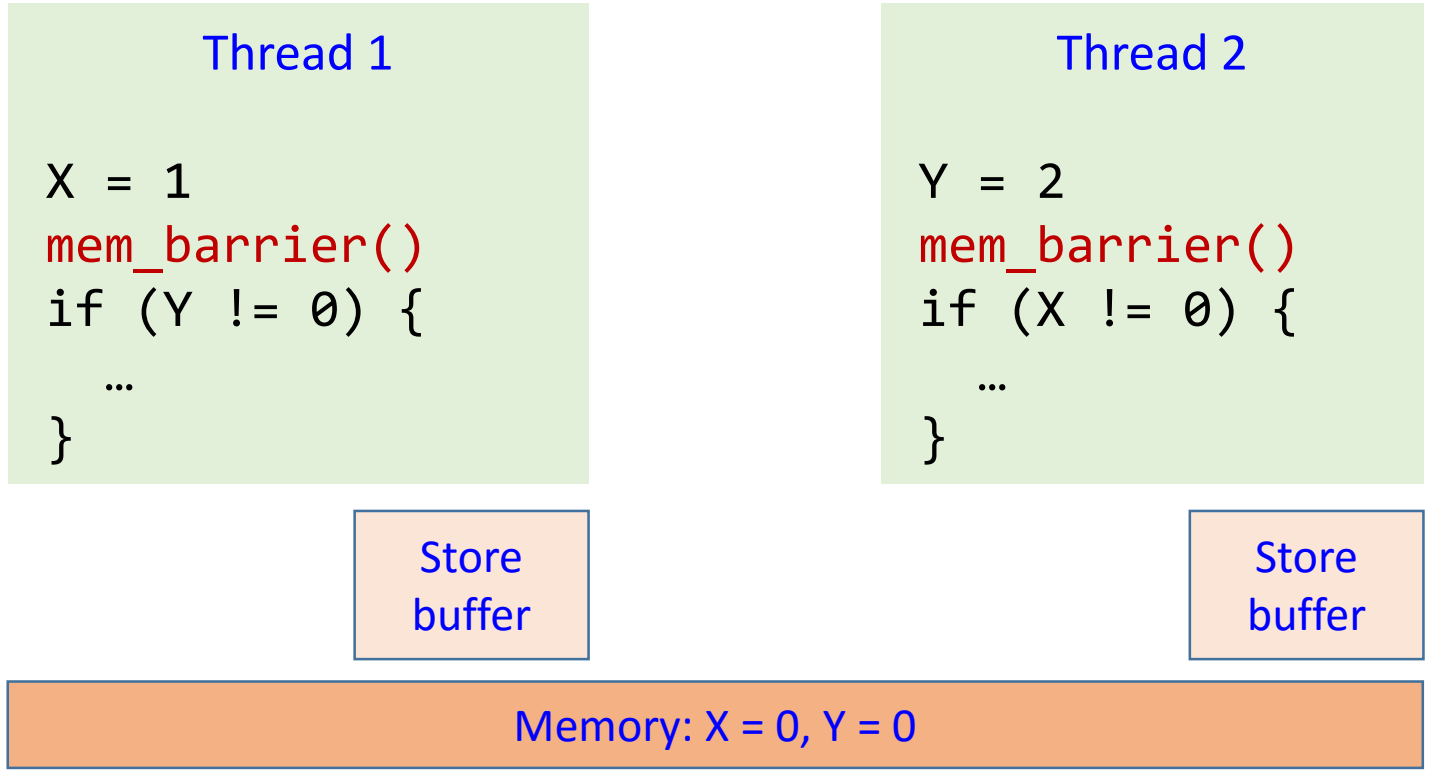
Ослабленные модели согласованности (Relaxed consistency)

- Операции с памятью (load, store) могут выполняются в порядке отличном от исходного
- Модель системы: кеш-память + поддержка когерентности кешей



Барьер памяти (memory barrier)

- **Барьер памяти** – инструкция, которая сбрасывает буфер записи/чтения
- Следующие операции работы с памятью не будут выполнены, пока не завершаться все находящиеся в очереди



Memory barrier

- **Compiler memory barrier** – предотвращает перестановку инструкций компилятором (в ходе оптимизации)

```
/* GNU inline assembler */
asm volatile("" ::: "memory");

/* Intel C++ intrinsic */
__memory_barrier();

/* Microsoft Visual C++ */
_ReadWriteBarrier()
```

- **Hardware memory barrier** – предотвращает перестановку инструкций процессором

```
/* x86, x86_64 */
void _mm_lfence();   /* lfence */
void _mm_sfence();   /* sfence */
void _mm_mfence();   /* mfence */
```

- **GCC: Built-in functions for atomic memory access //** http://gcc.gnu.org/onlinedocs/gcc-4.6.2/gcc/Atomic-Builtins.html
- **LLVM Atomic Instructions and Concurrency Guide** // http://llvm.org/docs/Atomics.html
- **Linux kernel memory barriers** // https://www.kernel.org/doc/Documentation/memory-barriers.txt

Memory barrier

```
volatile bool stopflag;
int a, b;

void run() {
    while (!stopflag);
    // Здесь нужен барьер, чтобы чтение stopflag всегда предшествовало обновлению b
    b = a;
}

int main() {
    stopflag = false;
    a = b = 0;
    std::thread mythread(run);
    std::this_thread::sleep_for(std::chrono::milliseconds(100));

    a = 1;
    // Здесь нужен барьер, чтобы a была видна всем потокам, перед обновлением stopflag,
    stopflag = true;

    mythread.join();
    return 0;
}
```

Атомарные типы

- **Атомарный тип (atomic type)** – это тип данных, который поддерживает атомарное выполнение операций
- std::atomic<bool>, std::atomic<Integral>, std::atomic<T*>
- **Методы класса std::atomic**
  - **bool is_lock_free()** – true, если реализации операций не использует блокировок
  - **T load(memory_order = std::memory_order_seq_cst)** – атомарно загружает и возвращает значение атомарной переменной
  - **void store(T desired, memory_order = std::memory_order_seq_cst)** – атомарно записывает значение в переменную
  - **…**

C++11 Atomic operations library

```
#include <atomic>

std::atomic<bool> stopflag;
int a, b;

void run()  {
    // Атомарное чтение stopflag + гарантия сохранения порядка выполнения операций
    while (!stopflag.load(/* memory_order = std::memory_order_seq_cst) */));
    b = a;
}

int main() {
    stopflag.store(false);
    a = b = 0;
    std::thread mythread(run);
    std::this_thread::sleep_for(std::chrono::milliseconds(100));

    a = 1;
    stopflag.store(true);
    mythread.join();
    return 0;
}
```

Барьерная синхронизация потоков

- **Барьерная синхронизация (Barrier)** – это примитив синхронизации, который заставляет каждый поток ожидать, пока остальные потоки не достигнут общей точки синхронизации

```
std::atomic<int> gsum;

void thread_routine(int id, int nthreads)
{
    int sum = 0;
    for (int i = 0; i < n; i += nthreads)
        sum += fx(i);
    gsum.fetch_add(sum);

    BARRIER();  // Синхронизируем все потоки, значение gsum дальше требуется всем потокам
    post_process(gsum);
}
```

Барьерная синхронизация потоков

```
void handler(int id, int n, barrier& b) {
    std::this_thread::sleep_for(std::chrono::milliseconds(rand() % 5000));
    print_time("Before barrier");
    b.wait();
    print_time("After barrier");
}

int main() {
    std::vector<std::thread> threads(10);
    barrier b(threads.size());

    for (size_t i = 0; i < threads.size(); ++i) {
        threads[i] = std::thread(handler, i, threads.size(), std::ref(b));
    }

    std::for_each(threads.begin(), threads.end(),
                  std::mem_fn(&std::thread::join));
    return 0;
}
```

Барьерная синхронизация потоков

```
class barrier {
    unsigned int const count;
    std::atomic<unsigned int> spaces;
    std::atomic<unsigned int> generation;

public:
    explicit barrier(unsigned nthreads): count(nthreads), spaces(nthreads),
                                         generation(0) { }

    void wait() {
        unsigned const my_generation = generation;
        if (!--spaces) {
            spaces = count;
            ++generation;
        } else {
            while (generation == my_generation)
                std::this_thread::yield();
        }
    }
};
```

- Maurice Herlihy, Nir Shavit. **The Art of Multiprocessor Programming, Morgan Kaufmann**, 2012, [C. 397] “17. Barriers”
- Эндрюс Г. **Основы многопоточного, параллельного и распределенного программирования**. - М.: Вильямс, 2003, [С. 103] “3.4 Барьерная синхронизация”

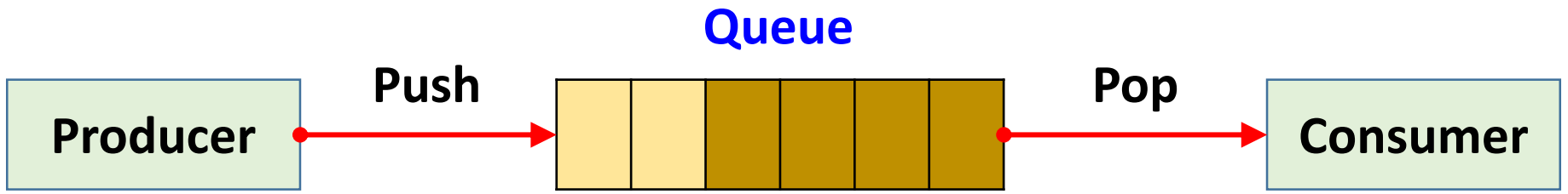
Барьерная синхронизация потоков

```
$./barrier
Thread 139857187747584: Before barrier: Wed Feb 19 11:53:51 2014
Thread 139857179354880: Before barrier: Wed Feb 19 11:53:51 2014
Thread 139857238103808: Before barrier: Wed Feb 19 11:53:51 2014
Thread 139857162569472: Before barrier: Wed Feb 19 11:53:52 2014
Thread 139857170962176: Before barrier: Wed Feb 19 11:53:52 2014
Thread 139857204532992: Before barrier: Wed Feb 19 11:53:52 2014
Thread 139857221318400: Before barrier: Wed Feb 19 11:53:53 2014
Thread 139857212925696: Before barrier: Wed Feb 19 11:53:53 2014
Thread 139857196140288: Before barrier: Wed Feb 19 11:53:54 2014
Thread 139857229711104: Before barrier: Wed Feb 19 11:53:55 2014
Thread 139857229711104: After barrier: Wed Feb 19 11:53:55 2014
Thread 139857212925696: After barrier: Wed Feb 19 11:53:55 2014
Thread 139857179354880: After barrier: Wed Feb 19 11:53:55 2014
Thread 139857238103808: After barrier: Wed Feb 19 11:53:55 2014
Thread 139857170962176: After barrier: Wed Feb 19 11:53:55 2014
Thread 139857221318400: After barrier: Wed Feb 19 11:53:55 2014
Thread 139857204532992: After barrier: Wed Feb 19 11:53:55 2014
Thread 139857162569472: After barrier: Wed Feb 19 11:53:55 2014
Thread 139857196140288: After barrier: Wed Feb 19 11:53:55 2014
Thread 139857187747584: After barrier: Wed Feb 19 11:53:55 2014
```

Условная синхронизация (Condition synchronization)

Producer–Consumer Problem

- **Производитель-потребитель** (Producer–consumer, bounded-buffer problem) – классическая задача синхронизации многопоточных программ
- Производитель помещает данные в очередь фиксированного размера, а потребитель забирает данные из нее
- Производитель не может помещать данные в заполненную очередь
- Потребитель не может забирать данные из пустой очереди



Условная синхронизация

- **Решение 1** – использовать разделяемую переменную-флаг, значение которой периодически опрашивается (поток будет непрерывно нагружать процессор)
- **Решение 2** – это решение 1 + периодически отправлять поток “спать”, второй поток успеет сменить флаг

```
bool flag;
std::mutex m;

void wait_for_flag() {
    std::unique_lock<std::mutex> l(m);
    while (!flag) {
        l.unlock();
        std::this_thread::sleep_for(std::chrono::milliseconds(100));  // Как выбирать 100? 200?
        l.lock();
    }
}
```

Условные переменные (Condition variables)

- **Условная переменная (Condition variable)** – это примитив синхронизации, позволяющий организовать ожидания наступления определенного события
- Условная переменная работает в паре с мьютексом
- #include <condition_variable>
- class condition_variable;
- **Notification**
  - void **notify_one**() – снимает блокировку с одного потока ожидающего на *this
  - void **notify_all**() – снимает блокировки со всех потоков ожидающих на *this
- **Wating**
  - void **wait**(std::unique_lock<**std::mutex**>& lock, Predicate pred) – ожидает пока предикат не примет значение истина

Решение 1 – очередь на базе std::queue<T>

```
#include <condition_variable>

std::queue<int> data_queue;
std::mutex data_mutex;
std::condition_variable data_condvar;

void producer()
{
    for (int i = 0; i < 10; ++i) {
        std::lock_guard<std::mutex> lg(data_mutex);  // Защищаем доступ к очереди
        int val = (i < 9) ? i + 1 : -1;
        data_queue.push(val);
        data_condvar.notify_one(); // Извещаем заблокированный поток о новых данных

    }  // unlock mutex
}
```

Решение 1 – очередь на базе std::queue<T>

```
void consumer()
{
    while (true) {
        std::unique_lock<std::mutex> lock(data_mutex); // Защищаем доступ к очереди
        // Wait:
        // Проверяем условие - если не выполнено, освобождаем lock и ожидаем извещения
        // Получили извещение - захватываем lock и проверяем условие
        // Условие выполнено - захватываем lock и выходим из wait
        data_condvar.wait(lock, []{ return !data_queue.empty(); });

        int val = data_queue.front();
        data_queue.pop();
        lock.unlock();

        std::cout << "Consumer " << val << "\n";
        if (val == -1)
            break;
    }
}
```

Решение 1 – очередь на базе std::queue<T>

```
int main()
{
    std::thread c(consumer);
    producer();
    //std::thread p(producer);
    c.join();
    //p.join();
    return 0;
}
```

Потокобезопасная очередь

```
template <typename T> class threadsafe_queue {
public:
    void put(T val) {
        std::lock_guard<std::mutex> lock(mutex);
        queue.push(val);
        cond.notify_one();   // в очереди появились данные
    }

    T take() {
        std::unique_lock<std::mutex> ulock(mutex);
        //while (queue.empty())
        //    cond.wait(ulock);
        cond.wait(ulock, [this]() { return !queue.empty(); });
        T val = queue.front();
        queue.pop();
        return val;
    }

private:
    std::queue<T> queue;
    std::mutex mutex;
    std::condition_variable cond;
};
```

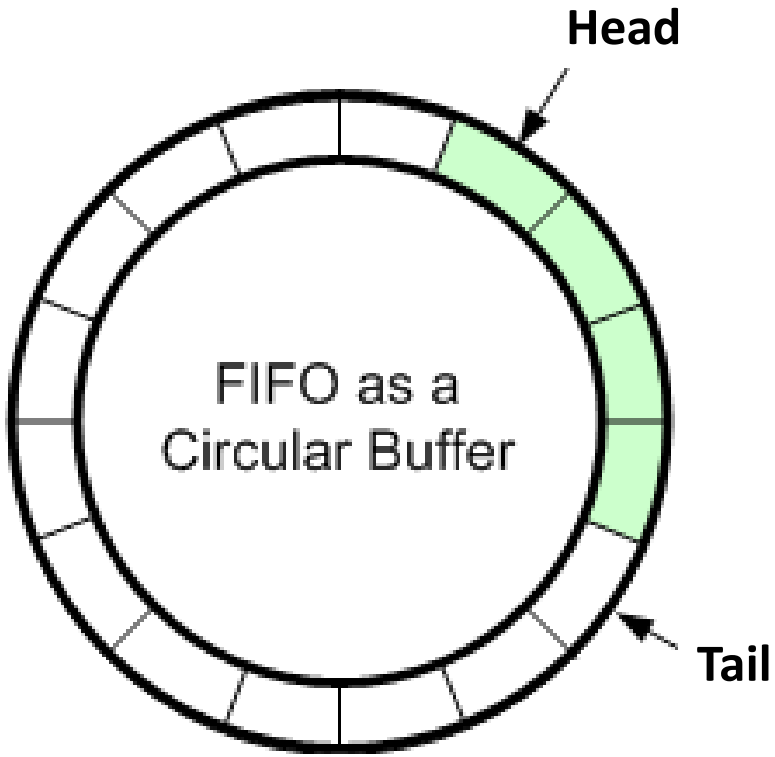
Потокобезопасная очередь

```
template <typename T>
void producer(int id, threadsafe_queue<T>& queue)
{
    for (size_t i = 0; i < 15; ++i) {
        queue.put(id + 100 + i);
        std::cout << "Producer " << id << " put " << id + 100 + i << "\n";
    }
}

template <typename T>
void consumer(int id, threadsafe_queue<T>& queue)
{
    for (size_t i = 0; i < 10; ++i) {
        T val = queue.take();
        std::cout << "Consumer " << id << " take " << val << "\n";
    }
}
```

Потокобезопасная очередь

```
int main()
{
    threadsafe_queue<int> queue;

    int nconsumers = 3;
    std::vector<std::thread> consumers;
    for (int i = 0; i < nconsumers; ++i)
        consumers.push_back(std::thread(consumer<int>, i, std::ref(queue)));

    int nproducers = 2;
    std::vector<std::thread> producers;
    for (int i = 0; i < nproducers; ++i)
        producers.push_back(std::thread(producer<int>, i, std::ref(queue)));

    for (std::thread& t : consumers)
        t.join();
    for (std::thread& t : producers)
        t.join();
    return 0;
}
```

Кольцевой буфер (Circular buffer, Bounded buffer)



Кольцевой буфер (Circular buffer, Bounded buffer)

```
template <typename T> class ringbuffer {
    T* buffer;
    int capacity;
    int head;
    int tail;
    int count;

    std::mutex mutex;
    std::condition_variable not_full;    // Сообщение - “в буфере есть свободная позиция”
    std::condition_variable not_empty;   // Сообщение - “буфер не пуст”

public:
    ringbuffer(int capacity): capacity(capacity), head(0), tail(0), count(0) {
        buffer = new T[capacity];
    }

    ~ringbuffer() {
        delete[] buffer;
    }
```

Кольцевой буфер (Circular buffer, Bounded buffer)

```
void put(T value)
{
    std::unique_lock<std::mutex> ulock(mutex);
    // Wait for free positions in the buffer
    not_full.wait(ulock, [this](){ return count != capacity; });

    buffer[tail] = value;
    tail = (tail + 1) % capacity;
    ++count;

    // Buffer has elems, notify waiting thread
    not_empty.notify_one();
}
```

Кольцевой буфер (Circular buffer, Bounded buffer)

```
    T take()
    {
        std::unique_lock<std::mutex> ulock(mutex);
        // Wait for elem in the buffer
        not_empty.wait(ulock, [this](){ return count != 0; });

        T value = buffer[head];
        head = (head + 1) % capacity;
        --count;
        // Buffer has free position now, notify waiting thread
        not_full.notify_one();
        return value;
    }

}; // class ringbuffer
```

Кольцевой буфер (Circular buffer, Bounded buffer)

```
template <typename T> void producer(int id, ringbuffer<T>& buf)
{
    for (size_t i = 0; i < 15; ++i) {
        buf.put(id + 100 + i);
        std::cout << "Producer " << id << " put " << id + 100 + i << "\n";
    }
}

template <typename T> void consumer(int id, ringbuffer<T>& buf)
{
    for (size_t i = 0; i < 10; ++i) {
        T val = buf.take();
        std::cout << "Consumer " << id << " take " << val << "\n";
    }
}
```

Кольцевой буфер (Circular buffer, Bounded buffer)

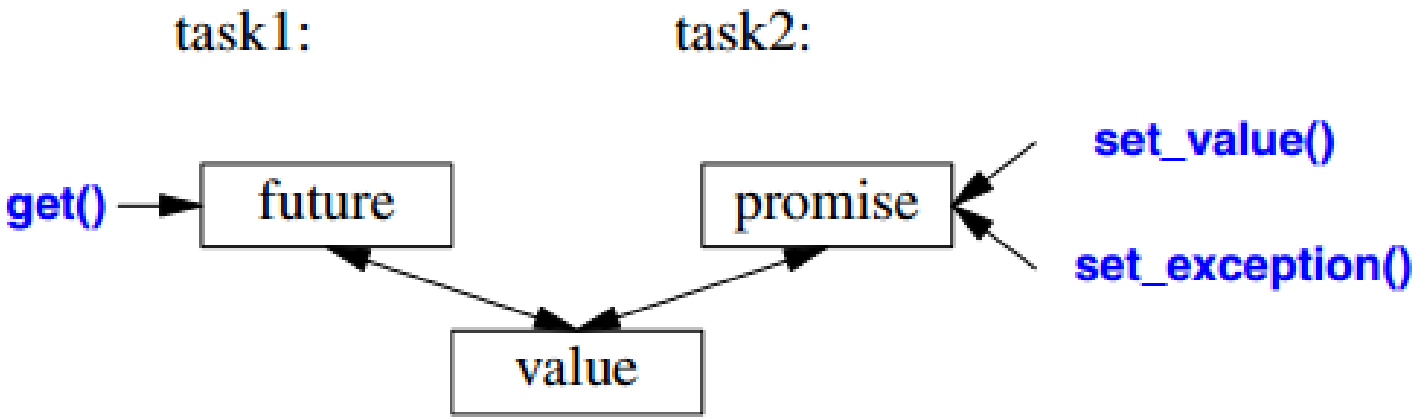
```
int main()
{
    ringbuffer<int> buffer(100);

    int nconsumers = 3;
    std::vector<std::thread> consumers;
    for (int i = 0; i < nconsumers; ++i)
        consumers.push_back(std::thread(consumer<int>, i, std::ref(buffer)));

    int nproducers = 2;
    std::vector<std::thread> producers;
    for (int i = 0; i < nproducers; ++i)
        producers.push_back(std::thread(producer<int>, i, std::ref(buffer)));

    for (std::thread& t : consumers)
        t.join();
    for (std::thread& t : producers)
        t.join();
    return 0;
}
```

Кольцевой буфер – активное ожидание (Busy wait)

```
void put(T value)
{
    std::unique_lock<std::mutex> ulock(mutex, std::defer_lock);
    // Wait for free positions in the buffer
    while (true) {
        ulock.lock();
        if (count != capacity) {
            buffer[tail] = value;
            tail = (tail + 1) % capacity;
            ++count;
            break;
        }
        ulock.unlock();
        std::this_thread::sleep_for(std::chrono::milliseconds(10));
    }
}
```

Будущие результаты (future) и асинхронные задачи (async)

- Как передать данные и исключения из одного потока в другой?



Stroustrup B. A Tour of C++

Будущие результаты (future) и асинхронные задачи (async)

```
#include <future>

double square_root(double x)
{
    print_time("Async started");
    if (x < 0)
        throw std::out_of_range("x < 0");
    return sqrt(x);
}

void do_other_stuff()
{
    std::this_thread::sleep_for(std::chrono::seconds(4));
}

void async_test(std::launch policy)
{
    print_time("Test started");
    std::future<double> f = std::async(policy, square_root, 10);
    do_other_stuff();
    std::cout << "Result: " << f.get() << std::endl;
}
```

Будущие результаты (future) и асинхронные задачи (async)

```
int main()
{
    async_test(std::launch::async | std::launch::deferred);
    async_test(std::launch::deferred);  // Launches in current thread (lazy evaluation)
    async_test(std::launch::async);     // Launches async in a separate thread
    return 0;
}
```

```
$ ./async
Test started: thread 140371877111680: Fri Feb 21 13:33:44 2016
Async started: thread 140371877111680: Fri Feb 21 13:33:48 2016
Result: 3.16228
Test started: thread 140371877111680: Fri Feb 21 13:33:48 2016
Async started: thread 140371877111680: Fri Feb 21 13:33:52 2016
Result: 3.16228
Test started: thread 140371877111680: Fri Feb 21 13:33:52 2016
Async started: thread 140371877103360: Fri Feb 21 13:33:52 2016
Result: 3.16228
```

std::packaged_task

```
double mysqrt(int x)
{
    print_time("mysqrt task");
    return sqrt(x);
}

int main()
{
    std::packaged_task<double(int)> ptask(mysqrt); // Связывает callable с будущим результатом
    std::future<double> f = ptask.get_future();
    print_time("Launch thread");
    std::thread t(std::move(ptask), 100);
    std::this_thread::sleep_for(std::chrono::seconds(4));
    print_time("Get future value");
    std::cout << "Value = " << f.get() << "\n";
    t.join();
    return 0;
}
```

```
Launch thread: thread 140650856376192: Sat Feb 22 09:11:51 2016
mysqrt task: thread 140650856367872: Sat Feb 22 09:11:51 2016
Get future value: thread 140650856376192: Sat Feb 22 09:11:55 2016
Value = 10
```

std::packaged_task – очередь задач (пул задач, task pool)

```
std::mutex m;
std::deque<std::packaged_task<void()>> tasks; // Очередь задач
bool stopflag = false;

void worker_thread()
{
    while (!stopflag) {
        std::packaged_task<void()> task;
        {
            std::lock_guard<std::mutex> lock(m);
            if (tasks.empty())
                continue;
            task = std::move(tasks.front()); // Извлекаем задачу из очереди
            tasks.pop_front();
        }
        task(); // Выполняем задачу в текущем потоке
    }
    std::cout << "Shutdown" << std::endl;
}
```

std::packaged_task – очередь задач (пул задач, task pool)

```
void task1() { std::cout << "Task1\n"; }   // Задача типа 1
void task2() { std::cout << "Task2\n"; }   // Задача типа 2
void shutdown_task() { stopflag = true; }  // Задача типа 3

template <typename Callable> std::future<void> post_task(Callable callable) {
    std::packaged_task<void()> task(callable);
    std::future<void> f = task.get_future();
    std::lock_guard<std::mutex> lock(m); // Помещаем задачу в конец двусторонней очереди
    tasks.push_back(std::move(task));
    return f;  // Возвращаем будущий результат задачи
}

int main() {
    std::thread worker(worker_thread);
    for (int i = 0; i < 10; i++)
        post_task(((i % 2 == 0) ? task1 : task2));  // Помещаем в очередь задачи
    post_task(shutdown_task);
    worker.join();
    return 0;
}
```

Очередь задач с возвращаемыми значениями

```
std::mutex m;
std::deque<std::packaged_task<double()>> tasks;
bool stopflag = false;

void worker_thread()
{
    while (!stopflag) {
        std::packaged_task<double()> task;
        {
            std::lock_guard<std::mutex> lock(m);
            if (tasks.empty())
                continue;
            task = std::move(tasks.front());
            tasks.pop_front();
        }
        task();
    }
    std::cout << "Shutdown" << std::endl;
}
```

Очередь задач с возвращаемыми значениями

```
double task1(int x) { std::cout << "Task1\n"; return sqrt(x); }
double task2(int x) { std::cout << "Task2\n"; return static_cast<double>(x) * x; }
double shutdown_task(int dummy) { stopflag = true; return 0; }

template <typename Callable> std::future<double> post_task(Callable callable) {
    std::packaged_task<double()> task(callable);
    std::future<double> f = task.get_future();
    std::lock_guard<std::mutex> lock(m);
    tasks.push_back(std::move(task));
    return f;
}

int main() {
    std::thread worker(worker_thread);
    std::vector<std::future<double>> results;
    for (int i = 0; i < 10; ++i)
        results.push_back(post_task(std::bind((i % 2 ? task1 : task2), i)));
    post_task(std::bind(shutdown_task, 0));
    worker.join();
    for (int i = 0; i < 10; ++i)
        std::cout << "Result " << results[i].get() << "\n";
    return 0;
}
```

std::promise – обещанный результат (ожидание результата)

```
double square_root(double x) {
    if (x < 0) throw std::out_of_range("x < 0");
    return sqrt(x);
}

void background_thread(std::promise<double> promise, double x) {
    std::this_thread::sleep_for(std::chrono::seconds(3));
    try {
        promise.set_value(square_root(x)); // Если не вызвать set_value => std::future_error
    } catch (std::exception&) {
        promise.set_exception(std::current_exception());
    }
}

int main() {
    std::promise<double> promise;  // Обещанный результат
    std::future<double> f = promise.get_future();
    print_time("Start thread");
    std::thread t(background_thread, std::move(promise), 10);
    print_time("Before get");
    std::cout << "Result " << f.get() << std::endl; // Поток блокируется (ждет вызова set_value)
    print_time("After get"); t.join();
}
```

std::promise – обещанный результат (ожидание результата)

```
Start thread: thread 140214906034048: Sat Feb 22 11:15:10 2016
Before get: thread 140214906034048: Sat Feb 22 11:15:10 2016
Result 3.16228
After get: thread 140214906034048: Sat Feb 22 11:15:13 2016
```

```
    std::this_thread::sleep_for(std::chrono::seconds(3));
    try {
        promise.set_value(square_root(x)); // Если не вызвать set_value => std::future_error
    } catch (std::exception&) {
        promise.set_exception(std::current_exception());
    }
}

int main() {
    std::promise<double> promise;  // Обещанный результат
    std::future<double> f = promise.get_future();
    print_time("Start thread");
    std::thread t(background_thread, std::move(promise), 10);
    print_time("Before get");
    std::cout << "Result " << f.get() << std::endl; // Поток блокируется (ждет вызова set_value)
    print_time("After get"); t.join();
}
```

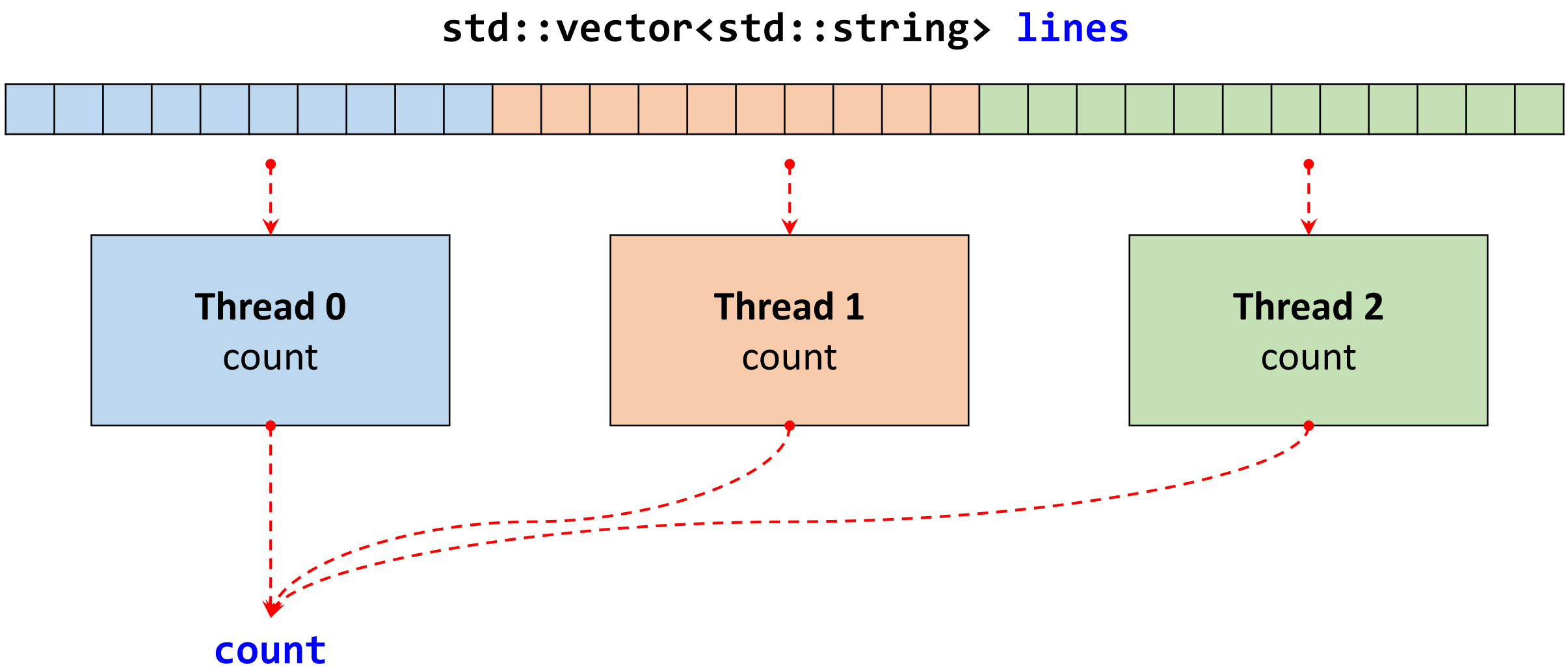
Поиск строки в файле (grep)

```
int main(int argc, char **argv)
{
    // Load file into std::vector
    start = std::chrono::high_resolution_clock::now();
    std::ifstream fs(argc > 1 ? argv[1] : "test.xml");
    std::string needle = argc > 2 ? argv[2] : "concurrent";
    std::vector<std::string> lines;
    for (std::string line; std::getline(fs, line); )
        lines.push_back(line);

    // Search
    std::vector<int> results;
    for (size_t i = 0; i < lines.size(); ++i) {
        std::transform(lines[i].begin(), lines[i].end(), lines[i].begin(), ::tolower);
        if (lines[i].find(needle) != std::string::npos)
            results.push_back(i);
    }
    std::cout << "Found " << results.size() << " matches for '" << needle << "'\n";
    return 0;
}
```

Поиск строки в файле (grep)

- **Поиск в эстонской Википедии (Vikipeedia)**
- XML-дамп ~ 478 MiB
- **Read**: 1622 ms
- **Search**: 2355 ms
- Lenovo ThinkPad X230
  - CPU Dual Core Intel Core i5-3320M (HyperThreading enabled), RAM 16 GiB, SSD OCZ Vertex3 256 GiB + SSD Kingston 128 GiB
  - GNU/Linux: Fedora 20: 3.13.3-201.fc20.x86_64, I/O scheduler=NOOP

Параллельный поиск строки в файле (Parallel grep)



Параллельный поиск строки в файле (Parallel grep)

```
// Распределяем строки по потокам
int nthreads = std::thread::hardware_concurrency();
int lines_per_thread = lines.size() / nthreads;

std::vector<std::future<std::vector<int>>> futures;
for (int i = 1; i < nthreads; ++i) {
    int from = i * lines_per_thread;
    int to = (i != nthreads - 1) ? from + lines_per_thread - 1 : lines.size() - 1;
    futures.push_back(std::async(std::launch::async, search, needle,
                      std::ref(lines), from, to));
}
std::vector<int> result = search(needle, std::ref(lines), 0, lines_per_thread - 1);

int count = result.size();
for (int i = 0; i < nthreads - 1; ++i)
    count += futures.at(i).get().size();
```

Параллельный поиск строки в файле (Parallel grep)

```
std::vector<int> search(const std::string& needle,
                        const std::vector<std::string>& lines, int from, int to)
{
    std::vector<int> results;
    for (int i = from; i < to; ++i) {
        std::string str = lines[i];
        std::transform(str.begin(), str.end(), str.begin(), ::tolower);
        if (str.find(needle) != std::string::npos)
            results.push_back(i);
    }
    return results;
}
```

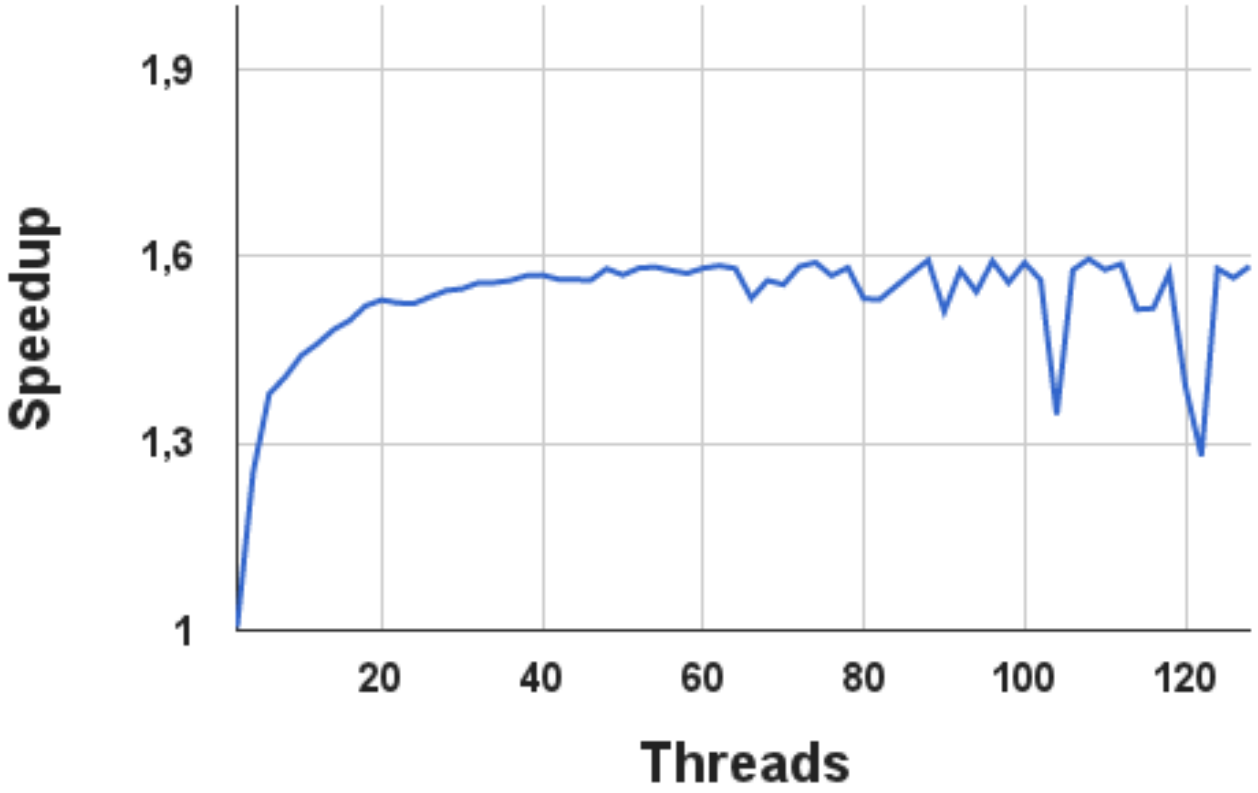
Параллельный поиск строки в файле (Parallel grep)

- **Коэффициент *S* ускорения (Speedup)**

$$S = \frac{T_{serial}}{T_{parallel}}$$

- **Узел вычислительного кластера**
- 2 x Quad Core Intel Xeon E5420 (8 cores per computer node)
- RAM 8 GiB
- HDD SATAII 500GB (Seagate Barracuda)

Плохая масштабируемость!



Параллельное чтение и поиск (производитель – потребитель)

```
int main(int argc, char **argv) {
    threadsafe_queue<std::string> queue;

    // Launch threads N - 1 for serach, master thread - read
    std::string needle = argc > 2 ? argv[2] : "concurrent";
    int nthreads = std::thread::hardware_concurrency();
    std::vector<std::future<std::vector<int>>> futures;
    for (int i = 1; i < nthreads; ++i)
        futures.push_back(std::async(std::launch::async, search, needle, std::ref(queue)));

    std::ifstream fs(argc > 1 ? argv[1] : "test.xml");  // Read file into queue
    for (std::string line; std::getline(fs, line); )
        queue.put(line);
    for (int i = 1; i < nthreads; ++i)  // Put terminate commands into the queue
        queue.put("_STOP_");

    int count = 0;
    for (int i = 0; i < nthreads - 1; ++i)
        count += futures.at(i).get().size();
    std::cout << "Found " << count << " matches for '" << needle << "'" << std::endl;
    return 0;
}
```

Параллельное чтение и поиск (производитель – потребитель)

```
template <typename T> class threadsafe_queue {
    // см. слайды выше
};

std::vector<int> search(const std::string& needle,
                        threadsafe_queue<std::string>& queue)
{
    std::vector<int> results;
    std::string str;
    while ((str = queue.take()) != "_STOP_") {
        std::transform(str.begin(), str.end(), str.begin(), ::tolower);
        if (str.find(needle) != std::string::npos)
            results.push_back(0);
    }
    return results;
}
```

Параллельное чтение и поиск (производитель – потребитель)

- Модифицировать threadsafe_queue – put/take помещают и извлекают из очереди блок строк (std::vector<std::string>)

Домашнее чтение

- **Disruptor: High performance alternative to bounded queues for exchanging data between concurrent threads**
- http://lmax-exchange.github.com/disruptor/files/Disruptor-1.0.pdf

- **Herb Sutter. Доклады на C++ and Beyond 2012**
- http://channel9.msdn.com/Shows/Going+Deep/C-and-Beyond-2012-Herb-Sutter-Concurrency-and-Parallelism

- **atomic<> Weapons: The C++ Memory Model and Modern Hardware**
- http://channel9.msdn.com/Shows/Going+Deep/Cpp-and-Beyond-2012-Herb-Sutter-atomic-Weapons-1-of-2
- http://channel9.msdn.com/Shows/Going+Deep/Cpp-and-Beyond-2012-Herb-Sutter-atomic-Weapons-2-of-2

Домашнее задание 1 – Задача 1

- Предложить решение задачи об обедающих философах (Dining philosophers problem), исключающее
  - ❑ взаимную блокировку (deadlock)
  - ❑ голодание потоков (starvation, livelock)
  - ❑ неравномерное распределение ресурсов (lack of fairness)
- Дополнительно требуется
  - ❑ минимизировать время ожидания философом еды
  - ❑ обеспечить масштабируемость решения при увеличении количества философов

Домашнее задание 1 – Задача 1

- **Реализуйте на C++11 многопоточный поисковый робот, основанный на обходе Web-графа в ширину и сохраняющий на диск все посещенные страницы**
- При запуске роботу передаются URL начальной страницы и глубина обхода
- Робот не должен посещать одну и ту же страницу более одного раза
- Попытайтесь добиться максимальной скорости работы робота, обоснуйте используемый для этого подход
- Для загрузки страниц на C++ рекомендуется использовать библиотеку cURL http://curl.haxx.se/libcurl/c/

Домашнее задание 1

- Решение + отчет + README присылать на почту (правила на сайте: Subject, …)
- **Deadline**: **22 марта 2016 до 23:59 NSK**