Уральский федеральный университет

Параллельные вычисления Многопоточное программирование, часть 1

Созыкин Андрей Владимирович

к.т.н.

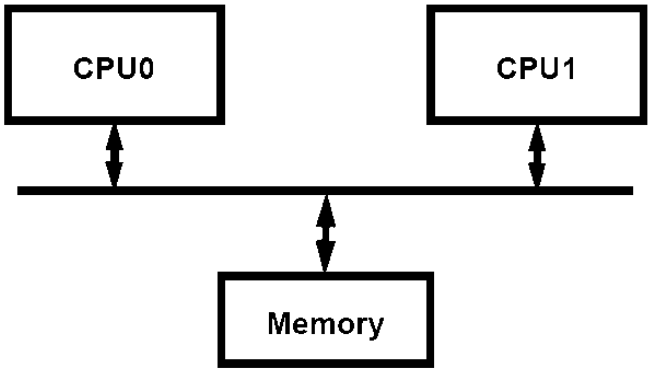
Заведующий кафедрой высокопроизводительных компьютерных технологий

Институт математики и компьютерных наук

Многопоточное программирование

Вычислительные системы с общей памятью:

- Hyper-Threading
- Многоядерные процессоры
- Многопроцессорные компьютеры



Многозадачная операционная система:

- Одновременно выполняется много программ (приложений и системных)

Процесс:

- абстракция операционной системе, позволяющая программе работать так, будто она выполняется на компьютере одна

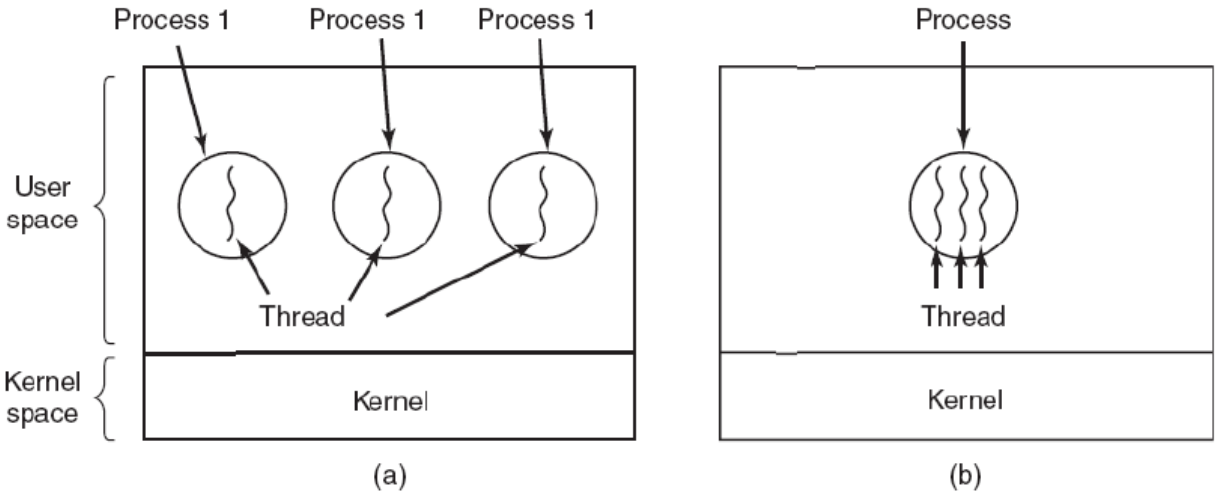
Процессы и потоки

Процесс – механизм выделения ресурсов

- Память, файлы и т.п.
- Как минимум один поток

Поток:

- Набор выполняемых команд
- Ресурсы у потоков общие



Таненбаум. Операционные системы

Зачем нужны несколько потоков?

Одновременная работа независимых задач (многозадачность)

Увеличение производительности программы

Увеличение пропускной способности

Отзывчивость графического интерфейса

Инструменты

Операционные системы:

- Pthreads (Linux, OS X), Windows threads

Библиотеки:

- Boost.Threads, Intel Threading Building Blocks

Языки программирования:

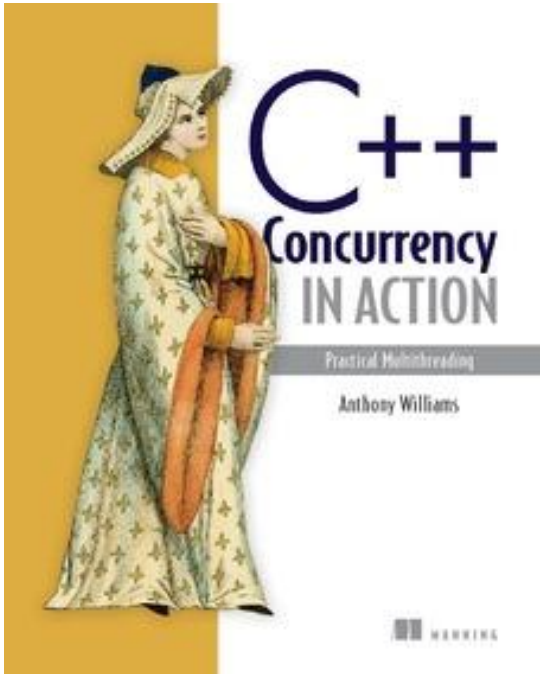
- Java, C#, C++11, Python

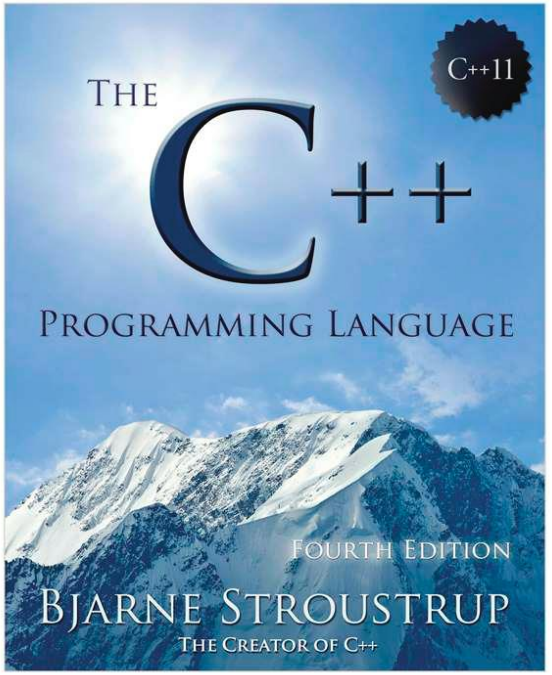
Многопоточное программирование на C++

Anthony Williams. C++ Concurrency in Action: Practical Multithreading. Manning, 2012

- http://www.cplusplusconcurrencyinaction.com/

Bjarne Stroustrup. The C++ Programming Language. 2013





Потоки в C++

Появились в C++11

До этого в C++ (и C) можно было использовать только платформо-зависимые потоки (Pthreads, Windows Threads)

Класс thread

Заголовочный файл <thread>

Потоки в C++

```
#include <iostream>
#include <thread>
using namespace std;

void thread_func(){
    // Printing Hello world and ID of the thread
    cout << "Hello world from thread " << this_thread::get_id() << "!" <<
endl;
}

int main() {
    // Creating new thread which will run the thread_func function
    thread t(thread_func);
    // Waiting for thread to finish
    t.join();
    // Printing ID of main thread
    cout << "Main thread id is " << this_thread::get_id() << endl;
}
```

Компиляция

gcc 4.7 и выше (icc, clang)

- `g++ -std=c++11 -pthread –Wall –g program.cpp -o program`
- В примерах кода есть makefile

Microsoft Visual Studio 2012 и выше

Потоки в C++. Callable

```
#include <iostream>
#include <thread>
using namespace std;

class thread_functor{
    public:
        void operator()() const {
            cout << "Hello world from functor!"  << endl;
        }
};

int main() {
    // Please notice the special syntax to deal with "most vexing parse"
    thread t2{thread_functor()};
    t2.join();
    cout << "Main thread id is " << this_thread::get_id() << endl;
}
```

Потоки в C++. Лямбда-выражение

```
#include <iostream>
#include <thread>
using namespace std;


int main() {
    // Creating new thread which will run the supplied lambda expression
    thread t1([]{
        cout << "Hello world from lambda!" << endl;
    });
    t1.join();
    cout << "Main thread id is " << this_thread::get_id() << endl;
}
```

Отделение потока

```
#include <iostream>
#include <thread>
using namespace std;

void thread_func(){
    // Do some background work here
}

int main() {
    // Creating new thread which will run the thread_func function
    thread t(thread_func);
    // Detaching the thread
    t.detach();
    // Printing ID of main thread
    cout << "Main thread id is " << this_thread::get_id() << endl;
}
```

Ожидание и отделение потока

thread::join() – ожидание завершения потока

thread::detach() – «отделение» потока и выполнение в фоновом режиме

Если не вызвать join или detach:

- Деструктор потока вызывает std::terminate()
- Программа остановится после завершения потока

Нет возможности принудительно остановить поток

Передача аргументов потоку

```
#include <iostream>
#include <thread>
#include <string>
using namespace std;

// Thread function with 2 arguments
void thread_func(string const &question, string &answer){
    cout << "The question: " << question << endl;
    // Return value through reference
    answer = "42";
}
int main(){
    string answer;
    string question("Life, the Universe and Everything");
    // Passing arguments to the thread function in the thread constructor
    thread t(thread_func, question, ref(answer));
    t.join();
    cout << "The answer: " << answer << endl;
}
```

Передача аргументов потоку

Аргументы **копируются** в память потока

- Даже если в объявлении функции указана передача по ссылке

Чтобы передать ссылку нужно использовать **std::ref**

Из потока нельзя «вернуть» значение (см. task)

Несколько потоков

```
#include <iostream>
#include <thread>
#include <vector>

void hello(unsigned id) {
  std::cout << id << ": Hello, Concurrent World!\n";
}
int main() {
  unsigned num_proc = std::thread::hardware_concurrency();
  std::cout << "Processors: " << num_proc << "\n";
  std::vector<std::thread> threads;
  for(unsigned i=0; i<num_proc; ++i) {
    threads.push_back(std::thread(hello, i));
  }
  std::for_each(threads.begin(), threads.end(),
  std::mem_fn(&std::thread::join));
}
```

Результаты запусков

0: Hello, Concurrent World!
1: Hello, Concurrent World!
2: Hello, Concurrent World!
3: Hello, Concurrent World!

2: Hello, Concurrent World!
3: Hello, Concurrent World!
0: Hello, Concurrent World!
1: Hello, Concurrent World!

10: Hello, Concurrent World!
: Hello, Concurrent World!
2: Hello, Concurrent World!
3: Hello, Concurrent World!

0: Hello, Concurrent World!
2: Hello, Concurrent World!
31: Hello, Concurrent World!
: Hello, Concurrent World!

Недетерминированность

Одно из неприятных свойств многопоточных и параллельных программ

Возможно большое количество вариантов выполнения программы

- Трасса – один из вариантов выполнения потоков в программе

Недетерминированность

Одно из неприятных свойств многопоточных и параллельных программ

Возможно большое количество вариантов выполнения программы

- Трасса – один из вариантов выполнения потоков в программе

Почему так происходит?

Недетерминированность

Одно из неприятных свойств многопоточных и параллельных программ

Возможно большое количество вариантов выполнения программы

- Трасса – один из вариантов выполнения потоков в программе

Почему так происходит?

Может ли программист задавать, какая именно трасса будет выполняться?

Недетерминированность

Одно из неприятных свойств многопоточных и параллельных программ

Возможно большое количество вариантов выполнения программы

- Трасса – один из вариантов выполнения потоков в программе

Многопоточные программы сложно отлаживать

Невозможно тестировать традиционными средствами

Условия гонок (race conditions)

```
int const THREAD_NUM = 20;

void compute(int const x, int &result){
    result += x * x;
}

int main(){
    int sum = 0;
    vector<thread> threads;
    for (int i = 0; i <= THREAD_NUM; i++)
        threads.push_back(thread(compute, i, ref(sum)));

    for (thread& t : threads)
        t.join();

    cout << sum << endl;
}
```

Проверим работу программы

```
#!/bin/bash
n=${2:-1000}
for (( i=0; i<n; i++ )) ; do $1 ; done | sort | uniq -c
```

Проверим работу программы

```
#!/bin/bash
n=${2:-1000}
for (( i=0; i<n; i++ )) ; do $1 ; done | sort | uniq -c
```

```
      1 2789
      2 2869
    997 2870
```

```
      1 2674
      1 2821
      1 2834
      1 2845
      1 2869
    995 2870
```

```
      1 2546
      1 2789
      2 2869
    996 2870
```

```
      1 2546
      1 2645
      1 2726
      2 2821
      1 2834
      1 2845
    993 2870
```

Условия гонок (race conditions)

Некорректное поведение программы при одновременной работе нескольких потоков с общими данными

Условия возникновения гонок

- Несколько потоков модифицируют общие данные

Несколько потоков читают данные

- Гонок не возникает

Условия гонок (race conditions)

```
int const THREAD_NUM = 20;

void compute(int const x, int &result){
    result += x * x;
}

int main(){
    int sum = 0;
    vector<thread> threads;
    for (int i = 0; i <= THREAD_NUM; i++)
        threads.push_back(thread(compute, i, ref(sum)));

    for (thread& t : threads)
        t.join();

    cout << sum << endl;
}
```

Почему возникают проблемы

```
void compute(int const x, int &result){
    result += x * x;
}
```

Как это выполняется:

- x загружается в регистр
- выполняется умножение x * x
- result загружается в регистр
- выполняется сложение
- результат записывается в result

Почему возникают проблемы

```
void compute(int const x, int &result){
    result += x * x;
}
```

Как это выполняется:

- x загружается в регистр
- выполняется умножение x * x
- result загружается в регистр

Что будет, если здесь поток прервётся?

- выполняется сложение
- результат записывается в result

Почему возникают проблемы

Поток 1 (x = 1, result = 0)

- x загружается в регистр
- выполняется умножение x * x (1)
- result загружается в регистр (0)
- выполняется сложение (1)
- результат записывается в result (1)

Поток 2 (x = 2, result = 0)

- x загружается в регистр
- выполняется умножение x * x (4)
- result загружается в регистр (0)
- выполняется сложение (4)
- результат записывается в result (4)

Синхронизация

Безопасный доступ к данным:

- Устранение гонок

Координация действий между потоками:

- Условная синхронизация (следующая лекция)

Взаимное исключение (Mutual Exclusion)

Критическая секция:

- Часть кода, где выполняются потенциально опасные действия с общими данными

Необходимо, чтобы критическая секция выполнялась только одним потоком:

- В каждый момент времени в критической секции только один поток
- Остальные потоки ждут завершения выполнения критической секции

В C++ реализуется с помощью объектов класса mutex (MUTual EXclusion)

Используем mutex для избавления от гонок

```
#include <iostream>
#include <vector>
#include <thread>
#include <mutex>
using namespace std;
int const THREAD_NUM = 20;

int main(){
    int sum = 0;
    mutex sum_mutex;
    vector<thread> threads;
    for (int i = 0; i <= THREAD_NUM; i++)
        threads.push_back(thread(compute, i, ref(sum), ref(sum_mutex)));

    for (thread& t : threads)
        t.join();

    cout << sum << endl;
}
```

Используем mutex для избавления от гонок

```
void compute(int const x, int &result, mutex &result_mutex){
    int temp = x * x;
    // Lock mutex
    result_mutex.lock();
    // Write to shared variable
    result += temp;
    // Unlock mutex after writing
    result_mutex.unlock();
}
```

Проверим работу программы

```
#!/bin/bash
n=${2:-1000}
for (( i=0; i<n; i++ )) ; do $1 ; done | sort | uniq -c
```

```
 1000 2870      1000 2870      1000 2870      1000 2870
```

Mutex и исключения

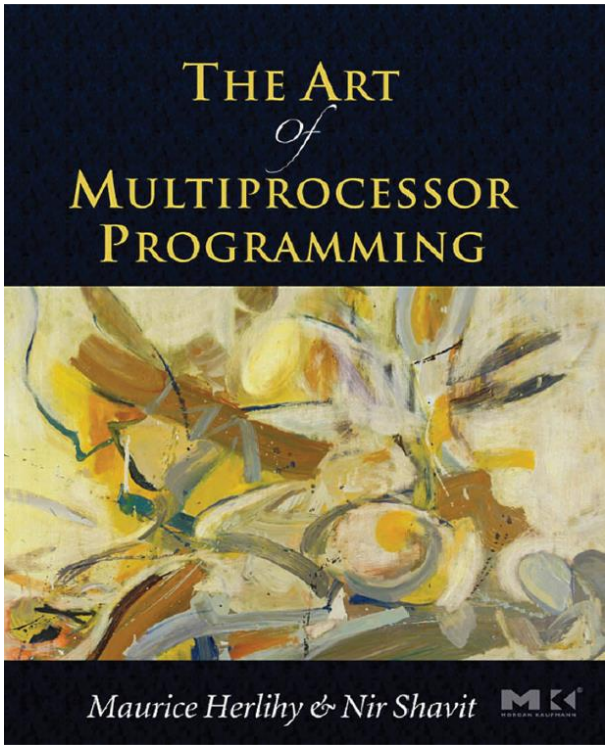
```
void compute(int const x, int &result, mutex &result_mutex){
    // Lock mutex
    result_mutex.lock();
    try {
        // Do some work here
        // Unlock mutex
        result_mutex.unlock();
    } catch (exception& e) {
        // Unlock mutex
        result_mutex.unlock();
    }
}
```

Mutex и RAII

```
void compute(int const x, int &result, mutex &result_mutex){
    int temp = x * x;
    // Creating lock guard for mutex
    lock_guard<std::mutex> lg(result_mutex);
    // Write to shared variable
    result += temp;
    // Lock guard going out of scope and unlocks mutex
}
```

Как устроен mutex?

Maurice Herlihy. Nir Shavit. The Art of Multiprocessor Programming, Revised Reprint



Атомарные операции

```
void compute(int const x, atomic<int> &result){
    result += x * x;
}

int main(){
    /// Atomic shared variable sum
    atomic<int> sum(0);
    vector<thread> threads;
    for (int i = 0; i <= THREAD_NUM; i++)
        threads.push_back(thread(compute, i, ref(sum)));

    for (thread& t : threads)
        t.join();

    cout << sum << endl;
}
```

Атомарные операции

Гарантированно выполняются без прерываний:

- Поддержка на уровне оборудования

Часть модели памяти C++11

Работают быстрее, чем mutex

Очень много ограничений (вызваны аппаратной реализацией)

task, async и future

"The best way to avoid data races is not to share the data"
- *Bjarne Stroustrup. The C++ Programming Language. 2013*

Программа состоит из независимых частей:

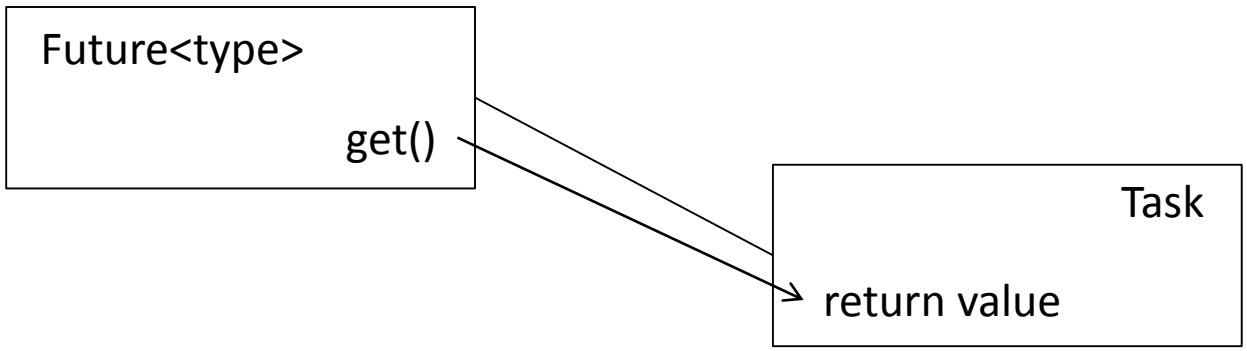
```
int compute(int const x){
    return x * x;
}
```

Такие задачи можно запускать отдельно

task, async и future

```
#include <iostream>
#include <vector>
#include <future>
using namespace std;

int compute(int const x){
    return x * x;
}

int main(){
    int sum = 0;
    vector<future<int>> futures;
    for (int i = 0; i <= THREAD_NUM; i++)
        futures.push_back(async(&compute, i));

    for (future<int>& f : futures)
        sum += f.get();

    cout << sum << endl;
}
```

task, async и future



task, async и future

task – задача, которая работает независимо и возвращает значение

async – служба для запуска task. Может запускать в нескольких потоках

future – позволяет получить результат выполнения task

- Если результат не готов, то метод get() блокируется до готовности результата

Преимущества

- Нет разделяемых данных – нет условий гонок
- Не нужно явно управлять потоками
- При переходе на машину с большим количеством ядер производительность увеличится автоматически

task, async и future

Всегда ли можно применять task, async и future?

Рекомендации по использованию

1. Task, async и future
2. mutex
3. Атомарные переменные и операции

Рекомендации по использованию

0. Оптимизируйте последовательную программу!

Следующая лекция

Типичные проблемы многопоточного программирования:

- Взаимоблокировки (deadlocks)
- Livelocks
- Условная синхронизация (проблема читателей и писателей)

Домашнее задание

Вопросы?