Введение в параллельные вычисления

д.т.н. Мокрова Наталия Владиславовна

асп. Морунов Егор, Сырко Денис

пятница	ауд. 119	
12:50 – 14:20	Лекция	
	1 неделя	2 неделя
14:35 - 17:55	Лаб – КС44	Лаб - КС40

Выбор модели программирования

Системы с общей памятью:

- C++11 threads (pthreads);
- OpenMP.

Системы с распределенной памятью

- MPI = message passing interface.

Гетерогенные системы (графические процессоры)

- CUDA.

Лекция 3. Синхронизация в C++11 thread

Лекция основана на примерах:

- https://solarianprogrammer.com/2012/02/27/cpp-11-thread-tutorial-part-2/
- сами примеры https://github.com/sol-prog/threads (код git clone).

Вспомнить C++ (не только thread)

- http://www.cplusplus.com/reference/thread/thread/

Почитать

- http://www.quizful.net/post/multithreading-cpp11

Демонстрация примеров

- http://en.cppreference.com/w/cpp/thread
- http://coliru.stacked-crooked.com/

Вспомни C++!

Передача аргументов по значению и по ссылке

```
#define N 5
#define M 100
int a[N][M];
void max_thread(int i) {
    int max = INT_MIN;
        for (int j = 0; j < M; j++) {
            if (a[i][j] > max) {
                max = a[i][j];
            }
        }
        std::cout << "Max in " << i << " is: "<< max << std::endl;
}
int main() { …
// Вызов потоков в цикле …
        t[i] = std::thread(max_thread, i);
}
```

Параметр

Вызов

```
void max_thread(int *a) {
    int max = INT_MIN;
        for (int j = 0; j < M; j++) {
            if (a[j] > max) {
                max = a[j];
            }
        }
        std::cout << "Max is: "<< max << std::endl;
}
int main() {
    int a[N][M];
        … // Ввод значений
        std::thread t[N];
        for (int i = 0; i < N; i++) {
            t[i] = std::thread(max_thread, a[i]);
        }
        for (int i = 0; i < N; i++) {
            t[i].join();
        }
        return 0;
}
```

Передача параметров в потоке

По значению

```
10  void threadFunction(int a)
11  {
12      a++;
13  }
14
15  int main()
16  {
17      int a = 1;
18      std::thread thr(threadFunction, a);
19      thr.join();
20      std::cout << a << std::endl;
21      return 0;
22  }
```

Результат 1

По ссылке

```
10  void threadFunction(int &a)
11  {
12      a++;
13  }
14
15  int main()
16  {
17      int a = 1;
18      std::thread thr(threadFunction, std::ref(a));
19      thr.join();
20      std::cout << a << std::endl;
21      return 0;
22  }
```

Результат 2

Выполнять ли потоки?

get_id	Get thread id (public member function)
joinable	Check if joinable (public member function)
join	Join thread (public member function)
detach	Detach thread (public member function)

```
void func() {
    cout << "thread function";
}
int main() {
    thread func_thread(func);
    if (func_thread.joinable())
        func_thread.join(); // Поток заканчивается, возвращение к main
    // func_thread.detach(); // Не ждать окончания потока
    return 0;
}
```

http://www.cplusplus.com/reference/thread/thread/

Функции управления текущим потоком

Полезные функции, предоставляемые <thread>, в пространстве имен std::this_thread

- get_id – возвращает id текущего потока; (активная ссылка)

```
std::cout<<std::this_thread::get_id()<<std::endl ;
```

- yield – предлагает планировщику выполнять другие потоки, может использоваться при активном ожидании;
- sleep_for – блокирует выполнение текущего потока на заданное время;
- sleep_until – блокирует выполнение текущего потока, до достижения заданного момента времени.

(http://en.cppreference.com/w/cpp/thread/get_id)

или

http://www.cplusplus.com/reference/thread/this_thread/

Пример 1 – вектор потоков

В прошлом лекции предлагалось использовать классический С-Тип массив, как группу потоков.

```
std::thread t[num_threads];
```

Используем std::вектор (более характерно для С++11), что позволяет избегать ловушек динамического выделения памяти с помощью new и delete.

```
 7  void func(int tid) {
 8      std::cout << "Launched by thread " << tid << std::endl;
 9  }
10
11  int main() {
12      std::vector<std::thread> th;
13
14      int nr_threads = 10;
15
16      //Launch a group of threads
17      for (int i = 0; i < nr_threads; ++i) {
18          th.push_back(std::thread(func,i));
19      }
20
21      //Join the threads with the main thread
22      for(auto &t : th){
23          t.join();
24      }
25
26      return 0;
27  }
```

Пример 2 – перемножение векторов

Дано: два вектора равной длины.
Перемножить их поэлементно и добавить результат каждого умножения к скалярной переменной.

```
void dot_product(const vector<int> &v1, const vector<int> &v2, int &result, int L, int R){
    for(int i = L; i < R; ++i){
        result += v1[i] * v2[i];
    }
}
```

```
int main(){
    int nr_elements = 100000;
    int nr_threads = 2;
    int result = 0;
    vector<std::thread> threads;
    //Fill two vectors with some values
    vector<int> v1(nr_elements,1), v2(nr_elements,2);
    //Split nr_elements into nr_threads parts
    vector<int> limits = bounds(nr_threads, nr_elements);
    //Launch nr_threads threads:
    for (int i = 0; i < nr_threads; ++i) {
        threads.push_back(thread(dot_product, ref(v1), ref(v2), ref(result), limits[i], limits[i+1]));
    }
    for(auto &t : threads){
        t.join();
    }
    cout << result << endl;
    return 0;
}
```

Параметры

Искомый результат 20000

Пример 2

```
5  //Split "mem" into "parts", e.g. if mem = 10 and parts = 4 you will have: 0,2,4,6,10
6  //if possible the function will split mem into equal chuncks, if not
7  //the last chunck will be slightly larger
8
9  std::vector<int> bounds(int parts, int mem) {
10     std::vector<int>bnd;
11     int delta = mem / parts;
12     int reminder = mem % parts;
13     int N1 = 0, N2 = 0;
14     bnd.push_back(N1);
15     for (int i = 0; i < parts; ++i) {
16         N2 = N1 + delta;
17         if (i == parts - 1)
18             N2 += reminder;
19         bnd.push_back(N2);
20         N1 = N2;
21     }
22     return bnd;
23 }
```

Разбиение на части (см. прим. 4 лек. 2)

```
1  sol $g++-4.7 -Wall -std=c++11 cpp11_threads_01.cpp
2  sol $./a.out
3  138832
4  sol $./a.out
5  138598
6  sol $./a.out
7  138032
8  sol $./a.out
9  140690
10 sol $
```

Примерный результат – https://solarianprogrammer.com/2012/02/27/cpp-11-thread-tutorial-part-2/

Пример «гонок потоков»

Алгоритмы взаимного исключения не дают возможности нескольким потокам, одновременно обращающихся к общим ресурсам, получать доступ. Это предотвращает гонки данных и обеспечивает синхронизацию между потоками.

Поток 1	**Поток 2**	Счет
Сколько денег? : 1000 $		1000 $
	Сколько денег? : 1000$	1000 $
	Внести 200 $	1000 $
Внести 200 $		1000 $
Обновить счет: $1000+$200		1200$
	Обновить счет: $1000+$200	1200$

При выполнении нескольких потоков они взаимодействуют друг с другом, взаимное исключение (мьютекс) критических участков кода позволяет избежать ситуации гонки данных.

Синхронизация потоков

Средства синхронизации могут использоваться для достижения двух разных целей:

- захват (как правило, на короткое время) разделяемого объекта для защиты критического интервала (блоки взаимного исключения – Mutex);
- ожидание (долгое или даже потенциально неограниченное) наступления некоторого события, выполнения некоторого условия (переменные состояния и семафоры).

Объекты синхронизации являются переменными, к ним можно обратиться, как к данным.

Потоки в различных процессах могут связаться друг с другом через объекты синхронизации, помещенные в разделяемую память потоков, даже в случае, когда потоки в различных процессах вообще невидимы друг для друга.

Объекты синхронизации можно разместить в файлах, где они будут находиться независимо от создавшего их процесса.

Мьютексы:

Mutexes

mutex	Mutex class (class)
recursive_mutex	Recursive mutex class (class)
timed_mutex	Timed mutex class (class)
recursive_timed_mutex	Recursive timed mutex (class)

Замки:

Locks

lock_guard	Lock guard (class template)
unique_lock	Unique lock (class template)

О синхронизации

Основные ситуации, требующие синхронизации:

- Если синхронизация – единственный способ гарантировать последовательность разделяемых данных.
- Если потоки в двух или более процессах могут использовать единственный объект синхронизации совместно. При этом объект синхронизации должен инициализироваться только одним из взаимодействующих процессов, потому что повторная инициализация объекта синхронизации переводит его в открытое состояние.
- Если синхронизация может гарантировать достоверность изменяющихся данных.
- Если процесс может отобразить файл и существует поток в этом процессе, который получает уникальный доступ к записям. Как только установлена блокировка, любой другой поток в любом процессе, отображающем файл, который пытается установить блокировку, блокируется, пока запись не будет освобождена.

Пример 3 – синхронизация потоков

```
1  #include <iostream>
2  #include <thread>
3  #include <vector>
4  #include <mutex>
5
6  static std::mutex barrier;
7
8  ...
9
10 void dot_product(const std::vector<int> &v1, const std::vector<int> &v2, int &result, int L, int R){
11     int partial_sum = 0;
12     for(int i = L; i < R; ++i){
13         partial_sum += v1[i] * v2[i];
14     }
15     std::lock_guard<std::mutex> block_threads_until_finish_this_job(barrier);
16     result += partial_sum;
17 }
18 ...
```

Строка 6 – создает глобальную переменную barrier типа mutex.
Строка 15 – блокировка остальных потоков до завершения текущего (для синхронизированного доступа к переменной result).
Переменная partial_sum объявлена локально для каждого потока.

Результат 20000

Блоки взаимного исключения (mutex)

Мьютекс – синхронизирующий объект, при помощи которого множество потоков управления могут упорядочить доступ к разделяемым переменным.

Mutex – mutual exclusion (взаимное исключение).

Блоки взаимного исключения – общий метод сериализации выполнения потоков. Мьютексы синхронизируют потоки, гарантируя, что только один поток в некоторый момент времени выполняет критическую секцию кода. Мьютексы можно использовать и в однопоточном коде.

Атрибуты мьютекса могут быть связаны с каждым потоком. Чтобы изменить атрибуты мьютекса по умолчанию, можно объявить и инициализировать объект атрибутов мьютекса, а затем изменить определенные значения. Часто атрибуты мьютекса устанавливаются в одном месте, в начале приложения, чтобы можно было быстро найти и изменить их.

После того, как сформированы атрибуты мьютекса, можно его непосредственно инициализировать.

Доступны следующие действия с мьютексом: инициализация, удаление, захват или открытие, попытка захвата.

Поток захватывает мьютекс в монопольное владение до тех пор, пока сам же его не освобождает.

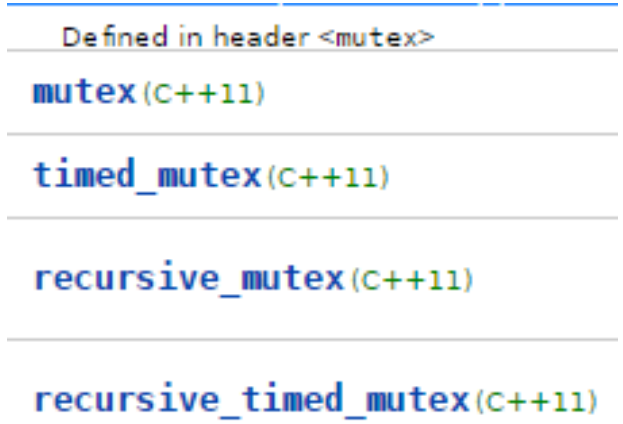
Другие потоки пытаются захватить занятый мьютекс, но им это не удается.

Mutex

Мьютекс – базовый элемент синхронизации в C++11 представлен в 4 формах:

- mutex – предоставляет базовую взаимное исключение объекта, обеспечивает базовые функции lock() и unlock() и неблокируемый метод try_lock();
- recursive_mutex – обеспечивает взаимное исключение объекта, который может быть заблокирован рекурсивно тем же потоком, может войти «сам в себя»;
- timed_mutex – обеспечивает взаимное исключение объекта, реализующего блокировку с тайм-аутом, в отличие от обычного мьютекса, имеет еще два метода – try_lock_for() и try_lock_until();
- recursive_timed_mutex – это комбинация timed_mutex и recursive_mutex.

http://en.cppreference.com/w/cpp/thread



Последовательность действий

1. Создать и инициализировать переменную для мьютекса.
2. Несколько потоков управления пытаются завладеть мьютексом.
3. Лишь один поток получает мьютекс.
4. Поток-владелец выполняет нужные операции.
5. Владелец освобождает мьютекс.
6. Другой поток управления получает мьютекс.
7. Удаление мьютекса.

Если несколько потоков управления изменяют одни и те же данные, то ответственность за то, чтобы все они перед этим обращались к мьютексу, лежит на программисте!

```
#include <iostream>
#include <thread>
#include <mutex>
using namespace std;
mutex mtx;          // mutex for critical section
void print_FUNC (int n, char c) {
  mtx.lock();
  for (int i=0; i<n; ++i) { cout << c; }
  cout << '\n';
  mtx.unlock();
}
int main () {
  thread th1 (print_FUNC,200,'|');
  thread th2 (print_FUNC,200,'_');
  th1.join();
  th2.join();
  return 0;
}
```

Еще блокировки

```cpp
#include <iostream>
#include <chrono>
#include <thread>
#include <mutex>

std::mutex g_lock;

void threadFunction(){
    g_lock.lock();
    std::cout << "entered thread " << std::this_thread::get_id() << std::endl;
    std::this_thread::sleep_for(std::chrono::seconds(rand()%10));
    std::cout << "leaving thread " << std::this_thread::get_id() << std::endl;
    g_lock.unlock();
}
int main(){
    srand((unsigned int)time(0));
    std::thread t1(threadFunction);
    std::thread t2(threadFunction);
    std::thread t3(threadFunction);
    t1.join();
    t2.join();
    t3.join();
    return 0;
}
```

Пример использования std::mutex с функциями get_id() и sleep_for().

http://habrahabr.ru/post/182610/

```
entered thread 140319302612736
leaving thread 140319302612736
entered thread 140319294220032
leaving thread 140319294220032
entered thread 140319285827328
leaving thread 140319285827328
```

Пример 4 – синхронизация потоков atomic

```
1  #include <iostream>
2  #include <thread>
3  #include <vector>
4  #include <atomic>
5
6  void dot_product(const std::vector<int> &v1, const std::vector<int> &v2, std::atomic<int> &result, int L,
7      int partial_sum = 0;
8      for(int i = L; i < R; ++i){
9          partial_sum += v1[i] * v2[i];
10     }
11     result += partial_sum;
12 }
13
14 int main(){
15     int nr_elements = 100000;
16     int nr_threads = 2;
17     std::atomic<int> result(0);
18     std::vector<std::thread> threads;
19
20         ...
21
22     return 0;
23 }
```

Для данного примера возможно более простое и более элегантное решение.

Используем тип atomic – особый вид переменной, которая позволяет безопасно выполнять одновременные операции чтения/записи, синхронизация выполняется на уровне доступа к переменной. Обратите внимание, для типа atomic представлен только определённый набор операций.

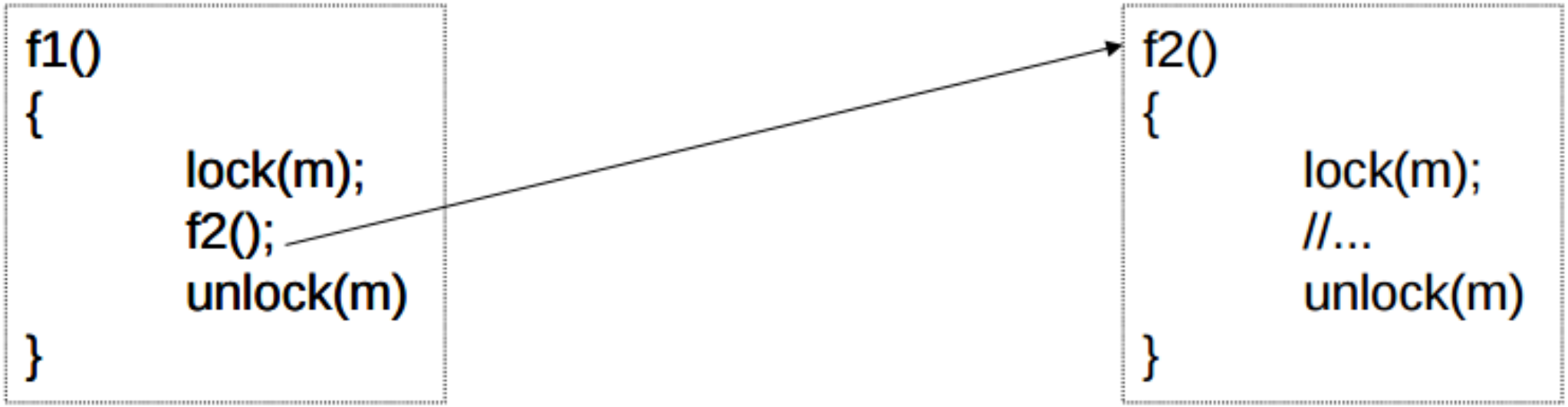
Тип atomic не доступен в текущей версии clang++.

Dead lock

thread1

thread2

pthread_mutex_t m;



Dead lock

std::lock_guard<std::mutex> lock(mutex);

или

std::lock_guard<std::recursive_mutex> lock(mutex);

```
x 32
x mul 32
x 32
x div 23
y 23
```

или

```
x 32
x mul 32
```

```
10  struct Complex {
11      std::mutex mutex;
12      int i;
13      Complex() : i(2) {}
14      void mul(int x){
15          std::lock_guard<std::mutex> lock(mutex);
16          cout << " x mul " << x << endl;
17          i *= x;
18      }
19      void div(int x){
20          std::lock_guard<std::mutex> lock(mutex);
21          i /= x;
22          cout << " x div " << x << endl;
23      }
24      void both(int x, int y){
25          std::lock_guard<std::mutex> lock(mutex);
26          mul(x);
27          cout << " x " << x << endl;
28          div(y);
29          cout << " y " << y << endl;
30      }
31  };
32  int main(){
33      Complex complex;
34      complex.both(32, 23);
35      return 0;
36  }
```

Некоторые выводы

Некоторые алгоритмы достаточно просто поддаются разбиению на независимо выполняемые фрагменты. Например, распределение работы по проверке всех чисел от 1 до 100000 на предмет того, какие из них являются простыми, может быть выполнено путем назначения каждому доступному процессору некоторого подмножества чисел с последующим объединением полученных множеств простых чисел (похожим образом реализован, например, проект GIMPS).

С другой стороны, большинство известных алгоритмов вычисления значения числа пи (π) не допускают разбиения на параллельно выполняемые части, т.к. требуют результата предыдущей итерации выполнения алгоритма. Итеративные численные методы, такие как, например, метод Ньютона или задача трёх тел, также являются сугубо последовательными алгоритмами. Некоторые примеры рекурсивных алгоритмов достаточно сложно поддаются распараллеливанию. Одним из примеров является поиск в глубину на графах.

Hello mas

```
Threads: 0.001779s
[0 ] [1 ] [2 ] [3 ]  8 7 8 6 4
[ 4 ]  7 1 7 7 7
6 10 3 10 2
 3 8 10 10 4
 2 9 8 10 10
```

```
[0 ] [1 ] [2 ] [3 ] [4 ]  8 7 8 6 4
 7 1 7 7 7
 3 8 10 10 4
 6 10 3 10 2
 2 9 8 10 10
```

- Гонка потоков

```
[ 0 ]      8     7      8      6      4
[  [ 34 ] ]      7      1      2      7      9      8      7      107      10
[ 1 ]      6     10     3      10     2

[ 2 ]      3     8      10     10     4
```

- Синхронизация потоков

```
#include <mutex>

8    static std::mutex barrier;
```

```
Max in 0 is: 8
Max in 1 is: 10
Max in 2 is: 10
Max in 4 is: 10
Max in 3 is: 7
Threads: 0.001396s
 [ 2 ]      3     8      10     10     4
 [ 3 ]      7     1      7      7      7
 [ 4 ]      2     9      8      10     10
 [ 1 ]      6     10     3      10     2
 [ 0 ]      8     7      8      6      4
```

```
std::lock_guard<std::mutex> block_threads_until_finish_this_job(barrier);
```

Hello string

```
#include <iostream>
#include <fstream>
```

Ввод-вывод с помощью потоков STL
https://code-live.ru/post/cpp-input-output/#-stl

```
// создание потока, открытие файла для записи в текстовом режиме,
// запись данных и закрытие файла.
ofstream ostr;
ostr.open(filename);
```

```
// чтение данных,
ifstream istr(filename);
```

Символы и строки в C++
http://cppstudio.com/post/437/

Результат

Поиск строки Tab }

```
42      }
43      return 0;
44  }
45
```

```
Thread: 1 Line: 22Thread: 8 Line: 39

Thread: 1 Line: 42
Thread: 2 Line: 33
```

```
./2.out 2.cpp $'\t}'
```