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 4. Проблемы параллельного программирования thread

Национальный Открытый Университет «ИНТУИТ»:
Параллельные вычисления и многопоточное программирование

- http://www.intuit.ru/studies/courses/10554/1092/lecture/27086

Основы современных операционных систем

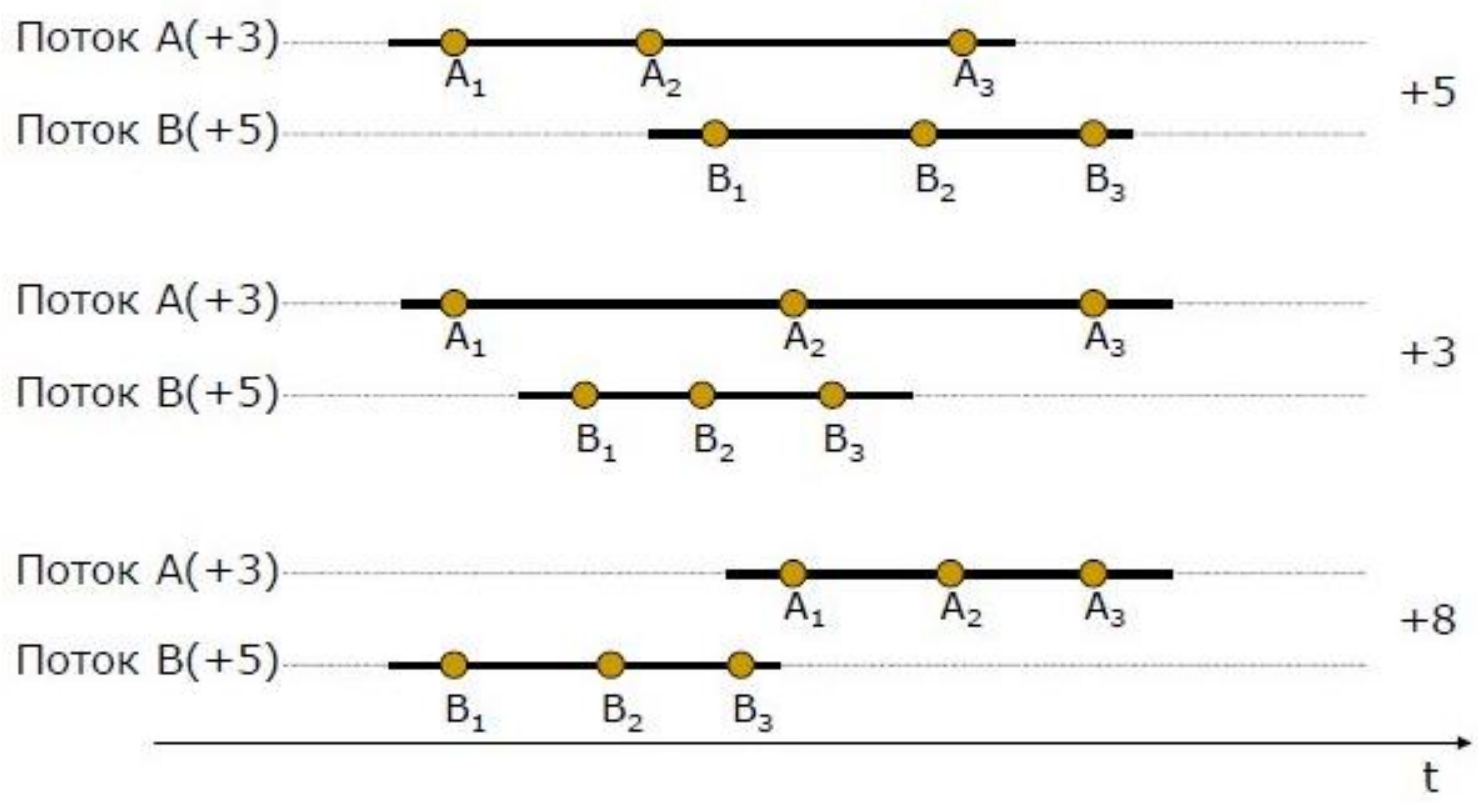
- http://www.intuit.ru/studies/courses/641/497/lecture/11284

Примеры:

- http://en.cppreference.com/w/cpp/thread/timed_mutex/lock
- http://ru.cppreference.com/w/cpp/thread/mutex/try_lock
- http://habrahabr.ru/post/147796/
- http://github.com/undertherain/pi

Необходимость синхронизации

или еще раз «гонка потоков»



Варианты диаграммы выполнения потоков на многопроцессорной системе

1.Считать запись из базы данных в локальный буфер потока.
2.Изменить значение записи.
3.Записать модифицированную запись из локального буфера потока в базу данных.

Важно! Шаги 2 и 3 должны выполняться последовательно.

Необходимость синхронизации

- Результат вычислений не однозначен, зависит от условий выполнения потоков, и разные запуски программы приведут к разным результатам! Это происходит при совместном использовании любых ресурсов, например, при использовании "обычных" общих (глобальных и статических) переменных.
- *Критическая секция* (КС, *critical section*) – часть программы, результат выполнения которой может непредсказуемо меняться, если в ходе ее выполнения состояние ресурсов, используемых в этой части программы, изменяется другими потоками.
- *Критическая секция* всегда определяется по отношению к определенным критическим ресурсам (например, критическим данным), при несогласованном доступе к которым могут возникнуть нежелательные эффекты.
- Одна *программа* может содержать несколько критических секций, относящихся к одним и тем же данным.
- Одна *критическая секция* может работать с различными критическими данными. Если несколько потоков одновременно работают с разными записями, никаких проблем нет – они возникают только при обращении нескольких потоков к одной и той же записи.
- Для разрешения проблемы согласованного доступа к ресурсу необходимо использовать синхронизацию. Например, обеспечить, чтобы в каждый момент времени в критической секции, связанной с определенными ресурсами, находился только один *поток*. Все остальные потоки должны блокироваться на входе в критическую секцию. Когда один *поток* покидает критическую секцию, один из ожидающих потоков может в нее войти. Подобное требование обычно называется взаимоисключением (*mutual exclusion*).

Критическая секция

Сколько хлеба купим?

Решение:

Сделаем процесс добывания хлеба *атомарной операцией*: перед началом этого процесса

- закрыть дверь изнутри на засов;
- уйти добывать хлеб через окно;
- вернуться в комнату через окно и отодвинуть засов.

Тогда пока один студент добывает хлеб, все остальные находятся в состоянии ожидания под дверью.

http://www.intuit.ru/studies/courses/2192/31/lecture/976?page=5

Время	Студент 1	Студент 2	Студент 3
17-05	Приходит в комнату		
17-07	Обнаруживает,что хлеба нет		
17-09	Уходит в магазин		
17-11		Приходит в комнату	
17-13		Обнаруживает, что хлеба нет	
17-15		Уходит в магазин	
17-17			Приходит в комнату
17-19			Обнаруживает,что хлеба нет
17-21			Уходит в магазин
17-23	Приходит в магазин		
17-25	Покупает 2 батона на всех		
17-27	Уходит из магазина		
17-29		Приходит в магазин	
17-31		Покупает 2 батона на всех	
17-33		Уходит из магазина	
17-35			Приходит в магазин
17-37			Покупает 2 батона на всех
17-39			Уходит из магазина
17-41	Возвращается в комнату		
17-43			
17-45			
17-47		Возвращается в комнату	
17-49			
17-51			
17-53			Возвращается в комнату

Задача взаимного исключения

Постановка задачи взаимного исключения: необходимо согласовать работу p > 1 параллельных потоков при использовании некоторого критического ресурса таким образом, чтобы:

- одновременно внутри критической секции должно находиться не более одного потока;
- относительные скорости развития потоков неизвестны и произвольны;
- критические секции не должны иметь приоритета в отношении друг друга;
- остановка какого-либо потока вне его критической секции не должна влиять на дальнейшую работу потоков по использованию критического ресурса;
- любой поток может переходить в любое состояние, отличное от активного, вне пределов своей критической секции;
- решение о вхождении потоков в их критические секции при одинаковом времени поступления запросов на такое вхождение и равноприоритетности потоков не откладывается на неопределенный срок, а является конечным во времени;
- освобождение критического ресурса и выход из критической секции должны быть произведены потоком, использующим критический ресурс, за конечное время.

Решением задачи взаимного исключения является *программа*, которая удовлетворяет всем поставленным условиям при любой реализовавшейся последовательности выполнения потоков (на любой диаграмме выполнения потоков).

Алгоритм Лампорта взаимного исключения

Каждый поток поддерживает очередь запросов на вход в критическую секцию. Приоритет – временная метка (часы с прямой зависимостью) (+ номер потока ?).

Когда поток хочет войти в критическую секцию, он:

1. добавляет свой запрос в свою очередь;
2. посылает всем потокам запрос;
3. ждет от них ответа;
4. получив все ответы, ждет, когда он станет первым в своей очереди, и входит в критическую секцию;
5. выйдя из критической секции, посылает всем сообщение release.

Действия вне критической секции:

• при получении запроса от другого потока, запрос добавляется в очередь и запрашивающему потоку посылается ответ;

• при получении release от другого потока, его запрос удаляется из очереди.

Суммарно на каждую критическую секцию приходится 3(N – 1) сообщение.

Еще см. алгоритм булочной.

Управление мьютексом

Пример демонстрирует использование lock, try_lock и unlock.

```
#include <iostream>
#include <mutex>
int main() {
    std::mutex test;
    if (test.try_lock()==true)
        std::cout << "блокировка установлена" << std::endl;
    else
        std::cout << "блокировка не установлена" << std::endl;
    test.unlock();    //теперь разблокируем мьютекс
    test.lock();          //заблокируем его снова
    if (test.try_lock())  //true можно опустить
        std::cout << "  блокировка установлена" << std::endl;
    else
        std::cout << "  блокировка не установлена" << std::endl;
    test.lock(); //и последнее (заблокируем)
}
```

В случае успешной установки блокировки возвращается true, в противном случае – false.

```
блокировка установлена
  блокировка не установлена
execution expired
```

Управление замком

```
#include <thread>
#include <mutex>
#include <iostream>
int g_i = 0;
std::mutex g_i_mutex;  // protects g_i
void safe_increment(){
    std::lock_guard<std::mutex> lock(g_i_mutex);
    ++g_i;
    std::cout << std::this_thread::get_id() << ": " << g_i << '\n'; // g_i_mutex освобождается автомати
}
int main(){
    std::cout << __func__ << ": " << g_i << '\n';
    std::thread t1(safe_increment);
    std::thread t2(safe_increment);
    t1.join();
    t2.join();
    std::cout << __func__ << ": " << g_i << '\n';
}
```

```
main: 0
139886758622976: 1
139886750230272: 2
main: 2
```

Безопасность потоков

Блокировка безопасна (с точки зрения взаимоблокировки).

std::mutex

```
1  #include <vector>
2  #include <mutex>
3  #include <thread>
4  std::vector<int> x;
5  std::mutex mutex;
6  void thread_func1()
7  {
8     mutex.lock();
9     x.push_back(0);
10    mutex.unlock();
11 }
12 void thread_func2()
13 {
14    mutex.lock();
15    x.pop_back();
16    mutex.unlock();
17 }
18 int main()
19 {
20    std::thread th1(thread_func1);
21    std::thread th2(thread_func2);
22    th1.join();
23    th2.join();
24 }
```

std::lock_guard

```
1  #include <vector>
2  #include <mutex>
3  #include <thread>
4  std::vector<int> x;
5  std::mutex mutex;
6  void thread_func1()
7  {
8     std::lock_guard<std::mutex> lock(mutex);
9     x.push_back(0);
10 }
11 void thread_func2()
12 {
13    std::lock_guard<std::mutex> lock(mutex);
14    x.pop_back();
15 }
16 int main()
17 {
18    std::thread th1(thread_func1);
19    std::thread th2(thread_func2);
20    th1.join();
21    th2.join();
22 }
23
```

Deadlock и безопасность

std::mutex

Один поток блокирует mutex, другой поток при входе в функцию lock mutex-а, который уже заблокирован, входит в режим ожидания и просыпается тогда, когда mutex освободится (т.е. заблокировавший его поток вызовет unlock).
deadlock – заблокированный поток так и остается ждать.

std::lock_guard

Техника RAII (Resource Acquisition Is Initialization) – решает проблему deadlock-а.
std::lock_guard – простой класс, конструктор которого вызывает метод lock для заданного объекта, а деструктор вызывает unlock.

В идеальном мире любую задачу можно было бы разделить на N подзадач, которые могли бы быть выполнены параллельно, тем самым мы бы получили 100% прирост производительности на 2-х процессорах, 400% на 4-х и так далее, но в реальном мире далеко не все задачи можно так разделить.

Классы управления блокировки

std::unique_lock – класс контролирующий блокировки mutex-а, предоставляет больше возможностей, чем std::lock_guard, например, предоставляет:

- возможность ручной блокировки и разблокировки контролируемого mutex-а с помощью методов lock и unlock;
- std::unique_lock можно перемещать с помощью вызова std::move;
- наиболее важно – объект класса std::unique_lock может не владеть правами на mutex, который он контролирует. При создании объекта можно отложить блокирование mutex-а передачей аргумента std::defer_lock конструктору std::unique_lock и указать, что объект на владеет правами на mutex и вызывать unlock в деструкторе не надо. Права на mutex можно получить позже, вызвав метод lock для объекта. Функцией owns_lock можно проверить, владеет ли текущий объект правами на mutex.

std::call_once

Класс std::call_once создан для того, чтобы защищать общие данные во время инициализации, это техника, позволяющая вызвать некий участок кода один раз, независимо от количества потоков, которые пытаются выполнить этот участок кода.

std::condition_variable – объект синхронизации, предназначенный для блокирования одного потока, пока он не будет оповещен о наступлении некоего события из другого.

std::promise – базовый механизм, позволяющий передавать значение между потоками.

Кто пойдет дальше

Mutual exclusion (Mutex)

Общие алгоритмы блокирования

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

Условные переменные
Условная переменная – примитив синхронизации, который позволяет множеству потоков взаимодействовать между собой. Она позволяет нескольким потокам ожидать (возможно, с таймером) оповещения от другого потока о том, что они могут продолжить выполнение. Условная переменная всегда связана с мьютексом.

Фьючерсы
Стандартная библиотека предоставляет средства для получения возвращаемых значений и обработки исключений, возникающих в процессе выполнения асинхронных задач (т. е. функций, запущенных в отдельных потоках). Эти данные передаются в “общее состояние” (*shared state)*. Доступ “общему состоянию” имеют все потоки, связанные с конкретным экземпляром класса std::future или std::shared_future, они могут получать, ожидать и изменять данные, которые там хранятся.

Считаем Пи параллельно

$$\int_0^1 \frac{4}{1+x^2}\,dx = \pi$$

http://habrahabr.ru/post/147796/

http://github.com/undertherain/pi

Подключить

```
7   #include <unistd.h>
```

Использован методов численного интегрирования – метод прямоугольников.

Параллельные потоки, исполняемые на разных ядрах или процессорах видят одно и то же адресное пространство, т.е. нет нужды явно передавать данные между потоками, если разные потоки пишут\читают одну и ту же переменную, то приходится синхронизировать.

Интервал интегрирования для каждого потока вычислен исходя из его порядкового номера.

Результат из каждого потока суммируем в переменной pi += sum*step, находящейся в общем адресном пространстве.

```
for (long long i=args->left; i<args->right; i++)
{
    x = (i + .5)*step;
    sum = sum + 4.0/(1.+ x*x);
}
```

```
pthread_mutex_lock(&mutexReduction);
pi += sum*step;
pthread_mutex_unlock(&mutexReduction);
```

Код thread

Код довольно громоздкий и некроссплатформенный.

```
for (unsigned int idThread=0; idThread<cntThreads; idThread++)
{
    arrArgsThread[idThread].left  = idThread*cntStepsPerThread;
    arrArgsThread[idThread].right = (idThread+1)*cntStepsPerThread;
    arrArgsThread[idThread].step = step;
    if (pthread_create(&threads[idThread], NULL, worker, &arrArgsThread[idThread]) != 0)
    {
        return EXIT_FAILURE;
    }
}
for (unsigned int idThread=0; idThread<cntThreads; idThread++)
{
    if (pthread_join(threads[idThread], NULL) != 0)
    {
        return EXIT_FAILURE;
    }
}
```

Почему не pthreads?

Интерфейс для организации нитей (**Pthreads**) поддерживается широко (практически на всех UNIX-системах), однако по многим причинам не подходит для практического параллельного программирования.

- нет поддержки Fortran;
- низкоуровневое средство;
- нет поддержки параллелизма по данным;
- механизм нитей изначально разрабатывался не для целей организации параллелизма.

Код OpenMP

```
#pragma omp parallel for private (x), reduction (+:sum)
for (int i=0; i<numSteps; i++)
{
    x = (i + .5)*step;
    sum = sum + 4.0/(1.+ x*x);
}
pi = sum*step;
```

OpenMP можно рассматривать как высокоуровневую надстройку над Pthreads (или аналогичными библиотеками нитей).