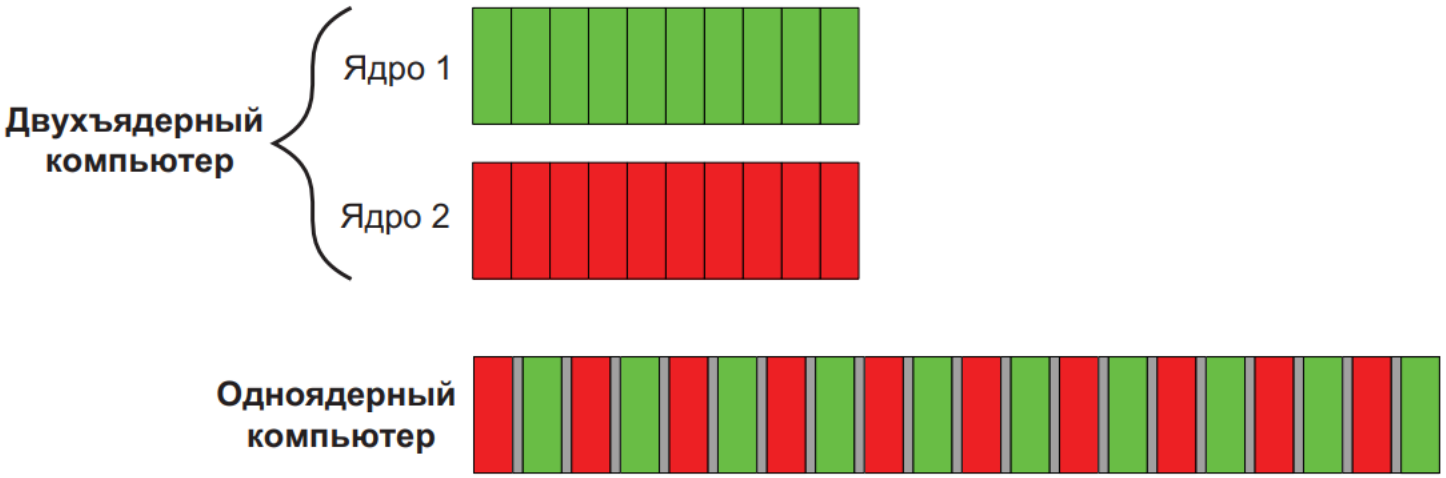
Лекция 8.
Библиотека STL и многопоточное программирование

3 семестр

Лектор: ст.пр. Бельченко Ф.М.

Понятие параллелизма

Параллелизм – это одновременное выполнение двух или более операций. Говоря о параллелизме в контексте компьютеров, мы имеем в виду, что одна и та же система выполняет несколько независимых операций параллельно, а не последовательно.

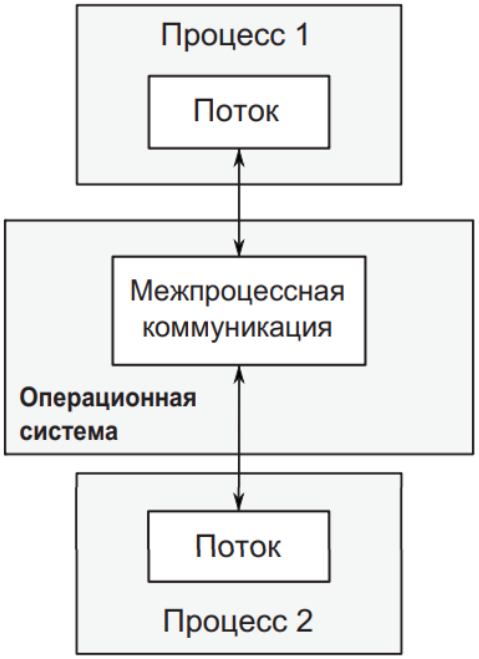


Параллелизм за счет нескольких процессов

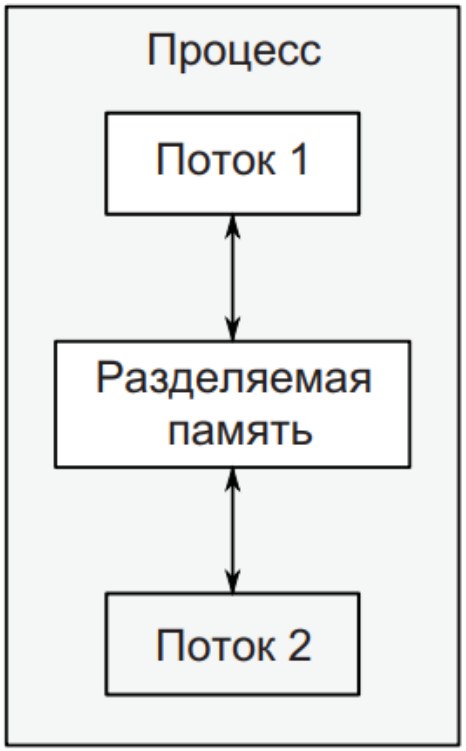
Первый способ распараллелить приложение – разбить его на несколько однопоточных одновременно исполняемых процессов. Затем эти отдельные процессы могут обмениваться сообщениями, применяя стандартные каналы межпроцессной коммуникации (сигналы, сокеты, файлы, конвейеры и т. д.).

Недостаток такой организации связи между процессами в его сложности, медленности, а иногда том и другом вместе. Дело в том, что операционная система должна обеспечить защиту процессов, так чтобы ни один не мог случайно изменить данные, принадлежащие другому.

Есть и еще один недостаток – неустранимые накладные расходы на запуск нескольких процессов: для запуска процесса требуется время, ОС должна выделить внутренние ресурсы для управления процессом и т. д.



Параллелизм за счет нескольких потоков



Альтернативный подход к организации параллелизма – запуск нескольких потоков в одном процессе. Потоки можно считать облегченными процессами – каждый поток работает независимо от всех остальных, и все потоки могут выполнять разные последовательности команд.

Однако все принадлежащие процессу потоки разделяют общее адресное пространство и имеют прямой доступ к большей части данных – глобальные переменные остаются глобальными, указатели и ссылки на объекты можно передавать из одного потока в другой.

Для процессов тоже можно организовать доступ к разделяемой памяти , но это и сделать сложнее, и управлять не так просто, потому что адреса одного и того же элемента данных в разных процессах могут оказаться разными.

Библиотека STL C++ 11

С появлением стандарта C++11, библиотека STL была расширена, включив в себя прямую поддержку многопоточности через модуль в заголовочном файле <thread>.

До стандарта C++11 приходилось использовать POSIX-потоки, а новый подход решал проблемы с портируемостью.

Многопоточность

Многопоточность - свойство платформы или приложения, состоящее в том, что процесс, порождённый в операционной системе, может состоять из нескольких потоков, выполняющихся «параллельно», то есть без предписанного порядка во времени.

Такие потоки называют также потоками выполнения (от англ. thread of execution); иногда называют «нитями» (букв. пер. англ. thread) или неформально «тредами».

Кроссплатформенные многопоточные библиотеки

Набор библиотек Boost
OpenMP
OpenThreads
POCO Thread (часть проекта POCO — http://pocoproject.org/poco/info/index.html)
Zthread
Pthreads (Ptreads-w32)
Qt Threads
Intel Threading Building Blocks
Стандартная библиотека STL (C++11)

Пример использования потока
из стандартной библиотеки библиотеки

```cpp
#include <iostream>
#include <thread>
#include <string>

void say_hello(const std::string& name) {
    std::cout << "hello " << name << std::endl;
}

int main(int argc, char* argv[]) {
    std::thread th(say_hello, "world");
    th.join();
    return 0;
}
```

```
Консоль отладки Microsoft Vi
hello world

C:\Users\phile\source\
ug\Proba.exe (процесс
ту с кодом 0.
Нажмите любую клавишу,
окно:
```

Класс std::thread

Класс std::thread нельзя копировать,

но его можно перемещать (std::move) и присваивать.

– Присваивать можно только те объекты, которые не связаны ни с каким потоком, тогда объекту будет присвоено только состояние,

– а при перемещении объекту передается состояние и право на управление потоком.

Историческая справка

Стандарт C++ 1998-ого года не имел упоминаний о существовании потоков.

C++0x определяет:

1. Новую модель памяти.
2. Библиотеку для разработки многопоточных приложений C++ threading library, включающую в себя:
   – средства синхронизации,
   – создания потоков,
   – атомарные типы и операции

Пример с получением идентификатора потока

Каждый поток имеет свой идентификатор типа std::thread::id,

- который можно получить вызовом метода get_id
- или же вызовом статического метода std::thread::this_thread::get_id из функции самого потока

```cpp
//Прмер с идентификатором потока

#include <iostream>
#include <thread>
#include <string>

void thread_func()
{
    std::cout << std::this_thread::get_id() << std::endl;
}
int main(int argc, char* argv[])
{
    std::thread th(thread_func);
    std::thread::id th_id = th.get_id();
    th.join();
    std::cout << th_id << std::endl;
    return 0;
}
```

```
Консоль отладки Microsoft Visual St...
14140
14140

C:\Users\phile\source\repos\P
ug\Proba.exe (процесс 12840)
оту с кодом 0.
Нажмите любую клавишу, чтобы
окно:
```

Семафоры

Семафор (semaphore) – примитив синхронизации работы процессов и потоков, в основе которого лежит счётчик, над которым можно производить две атомарные операции: увеличение и уменьшение значения на единицу, при этом операция уменьшения для нулевого значения счётчика является блокирующей. Служит для построения более сложных механизмов синхронизации и используется для синхронизации параллельно работающих задач, для защиты передачи данных через разделяемую память, для защиты критических секций, а также для управления доступом к аппаратному обеспечению.

Семафоры могут быть двоичными и вычислительными. Вычислительные семафоры могут принимать целочисленные неотрицательные значения и используются для работы с ресурсами, количество которых ограничено, либо участвуют в синхронизации параллельно исполняемых задач. Двоичные семафоры могут принимать только значения 0 и 1 и используются для взаимного исключения одновременного нахождения двух или более процессов в своих критических секциях.

Пример семафора

```cpp
// глобальные экземпляры двоичных семафоров
// счетчик объектов установлен в ноль
// объекты в несигнальном состоянии
std::binary_semaphore
smphSignalMainToThread(0),
smphSignalThreadToMain(0);

void ThreadProc()
{
    // ждем сигнала от main,
    // пытаясь уменьшить значение семафора
    smphSignalMainToThread.acquire();

    // этот вызов блокируется до тех пор,
    // пока счетчик семафора не увеличится в main

    std::cout << "[thread] Got the signal\n"; // ответное сообщение

    // ждем 3 секунды для имитации какой-то работы,
    // выполняемой потоком
    using namespace std::literals;
    std::this_thread::sleep_for(3s);

    std::cout << "[thread] Send the signal\n"; // сообщение

    // сигнализируем обратно в main
    smphSignalThreadToMain.release();
}
```

```cpp
int main()
{
    // создаем какой-то обрабатывающий поток
    std::thread thrWorker(ThreadProc);

    std::cout << "[main] Send the signal\n"; // сообщение

    // сигнализируем рабочему потоку о начале работы,
    // увеличивая значение счетчика семафора
    smphSignalMainToThread.release();

    // ждем, пока рабочий поток не выполнит свою работу,
    // пытаясь уменьшить значение счетчика семафора
    smphSignalThreadToMain.acquire();

    std::cout << "[main] Got the signal\n"; // ответное сообщение
    thrWorker.join();
}
```

```
Выбрать Консоль отладки Microsoft...
[main] Send the signal
[thread] Got the signal
[thread] Send the signal
[main] Got the signal

C:\Users\phile\source\repos\Proba\x64\Debug\Proba.exe (процесс 1128) завершил работу с кодом 0.
```

Мьютекс

Мьютекс (англ. mutex, от mutual exclusion — «взаимное исключение») — служит для синхронизации одновременно выполняющихся потоков.

Когда какой-либо поток, принадлежащий любому процессу, становится владельцем объекта mutex, последний переводится в неотмеченное состояние. Если задача освобождает мьютекс, его состояние становится отмеченным.

Мьютексы — это простейшие двоичные семафоры, которые могут находиться в одном из двух состояний — отмеченном или неотмеченном (открыт и закрыт соответственно).

Мьютекс отличается от семафора общего вида тем, что только владеющий им поток может его освободить, т.е. перевести в отмеченное состояние.

Пример использования мьютекса

Мьютекс обеспечивает защиту объекта от доступа к нему других потоков, отличных от того, который завладел мьютексом.

- В каждый конкретный момент только один поток может владеть объектом, защищённым мьютексом.

- Если другому потоку будет нужен доступ к переменной, защищённой мьютексом, то этот поток засыпает до тех пор, пока мьютекс не будет освобождён.

```cpp
// Пример мьютекса
#include <vector>
#include <mutex>
#include <thread>

std::vector<int> x;
std::mutex mutex;

void thread_func1() {
    mutex.lock(); x.push_back(0); mutex.unlock();
}

void thread_func2() {
    mutex.lock(); x.pop_back(); mutex.unlock();
}

int main() {
    std::thread th1(thread_func1);
    std::thread th2(thread_func2);
    th1.join();
    th2.join();
    return 0;
}
```

Рабочий пример использования мьютексов

```cpp
#include <iostream>
#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;
}
```

```
Консоль отладки Microsoft Visual St...
entered thread 12312
leaving thread 12312
entered thread 25236
leaving thread 25236
entered thread 23200
leaving thread 23200

C:\Users\phile\source\repos\
ug\Proba.exe (процесс 23212)
оту с кодом 0.
Нажмите любую клавишу, чтобы
окно:
```

Проблема безопасности исключений в C++ threading library

Тем не менее не рекомендуется использовать класс std::mutex напрямую, так как если между вызовами lock и unlock будет сгенерировано исключение - произойдет deadlock (т.е. заблокированный поток так и останется ждать).

Общий совет по обходу взаимной блокировки заключается в постоянной блокировке двух мьютексов в одном и том же порядке: если всегда блокировать мьютекс А перед блокировкой мьютекса Б, то взаимной блокировки никогда не произойдет. Иногда это условие выполнить несложно, поскольку мьютексы служат разным целям, но кое-когда всё гораздо сложнее, например, когда каждый из мьютексов защищает отдельный экземпляр одного и того же класса.

Классы мьютексов

Стандарт C++ 11 определяет различные классы мьютексов:

- mutex — нет контроля повторного захвата тем же потоком;
- recursive_mutex — повторные захваты тем же потоком допустимы, ведётся счётчик таких захватов;
- timed_mutex — нет контроля повторного захвата тем же потоком, поддерживается захват мьютекса с тайм-аутом;
- recursive_timed_mutex — повторные захваты тем же потоком допустимы, ведётся счётчик таких захватов, поддерживается захват мьютекса с тайм-аутом.