Лекция 12. Многопоточность

Сперва немного общих слов...

Процесс и потоки

- Процесс – ресурсы:
  - адресное пространство (память)
  - объекты ядра (файловые дескрипторы, объекты синхронизации, сокеты, …)
- Поток – выполнение инструкций
  - последовательность команд
  - стек
  - thread local storage (TLS)
  - используют общие ресурсы процесса

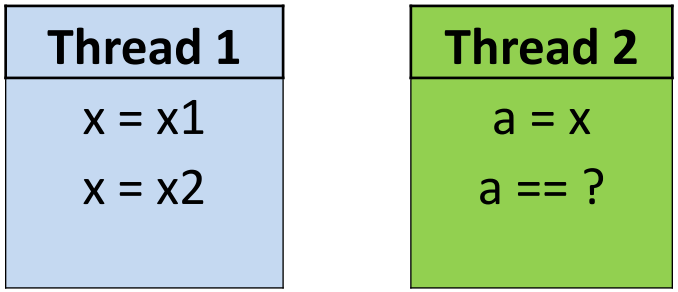
Типы многозадачности

- Cooperative (совместная). Следующая задача выполняется только после того, как предыдущая явно отдала поток управления (lightweight threads, fibers).

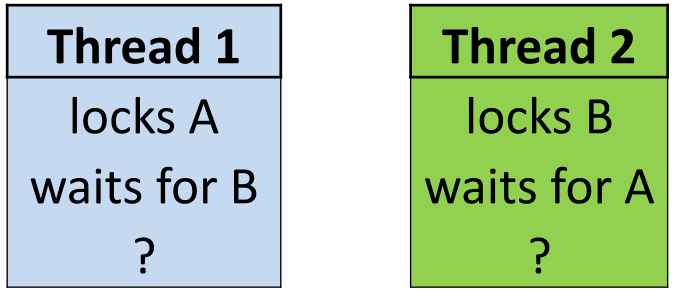
- Preemptive (вытесняющая). Операционная система сама определяет когда забрать поток управления у задачи и отдать его другой задаче.

Проблемные ситуации

- Race condition/data race (состояние гонки).



- Deadlock (взаимная блокировка).



- Livelock – блокировки как таковой нет, но крутимся в бессмысленном цикле.
- etc.

Многопоточность в C++

- Thread object
- Threads utility (id, yield, sleep, …)
- Mutual exclusion
- Lock management
- Condition variables
- Futures (async, promise, …)
- Memory model (atomic)
- Thread Local Storage (TLS)

- Косвенно:
  - OpenMP*
  - Exceptions
  - Message/event loop (косвенно)
  - Priorities

Многопоточность в C++

- Thread object
- Threads utility (id, yield, sleep, …)
- Mutual exclusion
- Lock management
- Condition variables
- Futures (async, promise, …)
- Memory model (atomic, …)
- Thread Local Storage (TLS)

- Косвенно:
  - OpenMP*
  - Exceptions
  - Message/event loop (косвенно)
  - Priorities

Создание потока

- Поток создается с помощью объекта std::thread (заголовочный файл <thread>).

```
1.  void big_calc(string how)
2.  {
3.      cout << "i\'m working " << how << "\n";
4.  }
5.
6.  int main()
7.  {
8.      thread th1(big_calc, "hard");
9.      thread th2(big_calc, "24/7");
10.
11.     // doing smth
12.
13.     th1.join();
14.     th2.join();
15.     return 0;
16. }
```

Объект потока std::thread

```
1.  // can be moved, not copied
2.  class thread
3.  {
4.      //...
5.      template< class Function, class... Args >
6.      explicit thread( Function&& func, Args&&... args );
7.
8.      // std::terminate if joinable
9.      ~thread();
10.
11.     bool                joinable            () const noexcept;
12.     std::thread::id     get_id              () const;
13.     native_handle_type  native_handle       ();
14.     static unsigned     hardware_concurrency();
15.
16.     void join  (); // waits for thread func finish
17.     void detach(); // detaches thread object from system thread
18.     // ...
19. };
```

Вспомогательные функции

- Находятся в namespace `std::this_thread`
  - `yield` - отдает поток управления
  - `get_id` – возвращает `std::thread::id`
  - `sleep_for/sleep_until` – приостанавливает поток

```
1.  void thread_func()
2.  {
3.      lock_guard<mutex> lock(m);
4.
5.      if (task_queue.empty())
6.          std::this_thread::yield(); // Hmmm...
7.      else
8.          //...
9.  }
```

Mutual exclusion

- mutex: обеспечивает базовые функции lock() и unlock() и не блокируемый метод try_lock()
- recursive_mutex: может войти «сам в себя»
- timed_mutex: 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
- shared_timed_mutex : предоставляет общий доступ нескольким потокам (читатели) или эксклюзивный одному потоку (писатель).

Lock management

- `lock_guard` – самый что ни на есть простой RAII держатеть mutex'а

- `unique_lock`
  - `try_lock[_for/_until]`
  - конструирование без или с тэгом: `defer_lock, try_to_lock, adopt_lock`
  - `owns_lock` проверка

- `shared_lock` – то же, что и `unique_lock`, но для `shared_timed_mutex`

Пример захвата mutex

- Чтобы избежать deadlock'а на нескольких mutex'ах используйте функцию `std::lock`

```
1.  void move_money(account& a1, account& a2, int amount)
2.  {
3.      unique_lock l1(a1.mut, defer_lock);
4.      unique_lock l2(a2.mut, defer_lock);
5.
6.      // avoids deadlocks
7.      std::lock(a1.mut, a2.mut);
8.
9.      if (a1.balance >= amount)
10.     {
11.         a1.balance -= amount;
12.         a2.balance += amount;
13.     }
14.     // ...
15. }
```

Exception catching

- Exception никаким образом не транслируется далее через границу потока.
- Что делать?

```
 1.  void thread_func()
 2.  {
 3.      try
 4.      {
 5.          // function body
 6.      }
 7.      catch (...)
 8.      {
 9.          // what to do with the exception?
10.      }
11.  }
```

Exception rethrow (1)

- Попытаемся его сперва поймать и сохранить!

```
1.  void thread_func()
2.  {
3.      try
4.      {
5.          // function body
6.      }
7.      catch (...)
8.      {
9.          std::exception_ptr p = current_exception();
10.
11.         lock_guard<mutex> l(exc_mut);
12.         exceptions_queue.push_back(p);
13.     }
14. }
```

Exception rethrow (2)

```
1.  void invoker()
2.  {
3.      thread th(thread_func);
4.      //...
5.      th.join();
6.      lock_guard<mutex> l(exc_mut);
7.
8.      while(!exceptions_queue.empty())
9.      {
10.         try
11.         {
12.             auto p = exceptions_queue.top();
13.             exceptions_queue.pop();
14.             rethrow_exception(p);
15.         }
16.         catch(std::exception const& e)
17.         {
18.             cout << "FAILURE: " << e.what() << end;
19.         }
20.     }
21. }
```

Вопросы?