

Лекция 11. Продвинутое управление потоками

ИУ8

November 18, 2016

План лекции:

- пул потоков
- прерывание работы потоков

Объект потока

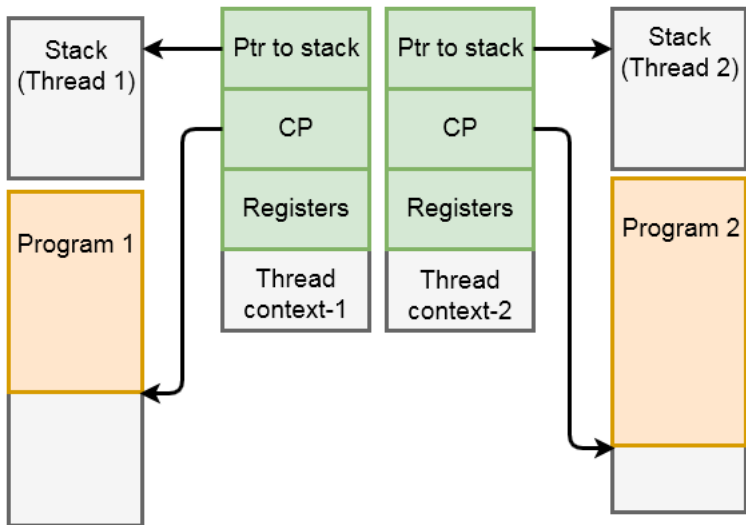
Чтобы ОС поддерживала многозадачность, каждый выполняемый поток должен обладать своим контекстом исполнения.

Этот контекст используется для хранения данных о текущем состоянии потока

Таким образом контекст состоит как минимум из

- значения регистров процессора
- указатель на стек потока
- счетчик команд

Memory



Таким образом, каждый новый поток занимает некоторый объем оперативной памяти.

Идем дальше.

Переключение контекста

В системе количество потоков может превышать число ядер, тогда будет применяться механизм переключения задач.

В произвольный момент времени ОС передает поток на исполнение процессору. Этот поток исполняется в течение некоторого временного интервала. После завершения этого интервала контекст ОС переключается на другой поток. При этом происходит следующее:

- 1 обновляется контекст потока;
- 2 из набора имеющихся потоков выбирается один, который будет исполняться на процессоре;
- 3 значения из выбранного контекста потока загружаются во внешнее окружение.

Context Switching

Total cost of
context switching

Multitasking



vs. Multitasking with context switching



Получаем, что новые потоки не только занимают объем оперативной памяти, но и создают дополнительные временные издержки, так как операционной системе приходится планировать исполнение потоков и выполнять переключения контекста.

Пул потоков

Во многих системах бессмысленно заводить отдельный поток для каждой задачи, которая потенциально может выполняться одновременно с другими, но тем не менее хотелось бы пользоваться благами параллелизма там, где это возможно.

Именно это позволяет сделать пул потоков: задачи, которые могут выполняться параллельно, отправляются в пул, а тот ставит их в очередь ожидающих работ. Затем один из рабочих потоков забирает задачу из очереди, исполняет ее и принимается за следующую задачу.

Простейший пул потоков

```
1 class thread_pool {
2     std::atomic_bool done = false;
3     thread_safe_queue<std::function<void()>> work_queue;
4     std::vector<std::thread> threads;
5
6     void worker_thread(){
7         while(!done.load()) {
8             std::function<void()> task;
9             if(work_queue.try_pop(task)) {
10                 task();
11             } else {
12                 std::this_thread::yield();
13             }
14         }
15     }
16 public :
17     ~thread_pool() {
18         done.store(true);
19     }
```

Простейший пул потоков: продолжение

```
1  thread_pool(){
2      int const thread_count = std::thread::hardware_concurrency();
3      try {
4          for(auto i = 0; i < thread_count; ++i) {
5              threads.push_back(std::thread(
6                  &thread_pool::worker_thread,
7                  this));
8          }
9      } catch(...) {
10         done.store(true);
11     }
12 }
13
14 template<typename Func>
15 void submit(Func f) {
16     work_queue.push(std::function<void()>(f));
17 }
18 }; // end of thread_pool
```

Пул потоков с возможностью получения результата задачи

см `thread_pool.h`

Как модифицировать пул

- динамически изменять количество рабочих потоков
- для каждого потока создать свою очередь задач
- предоставить доступ к списку задач
- добавить в пул потоков дополнительных рабочих

Динамическое число рабочих

Суть: в зависимости от количества задач изменять количество рабочих потоков

- + не занимаем ОП под потоки, которые ожидают задач
- + снижаем конкуренцию при доступе к очереди задач
- + уменьшаем количество потоков в ОС
 - усложняется логика работы пула
 - тратим дополнительное время на создание потока

Отдельные очереди задач

Суть: чтобы общая очередь задач не стала узким местом, создаем по очереди задач для каждого потока

- + сводим конкуренцию за доступом к очереди задач к минимуму
 - усложняется логика работы пула
 - возможна ситуация обработки наиболее «длительных» задач только одним потоком, тогда как остальные потоки простаивают

Доступ к ресурсам пула

Суть: если мы имеем дополнительный вычислительный ресурс, то можем выполнять задачи из очереди задач пула потоков

- + динамически увеличиваем вычислительные способности пула
- + уменьшаем вероятность блокировки задач
- усложнение логики работы пула

Прерывание потоков

Бывают ситуации, когда требуется сообщить работающему потоку о необходимости остановки.

Например, программа в отдельном потоке скачивает файл, но пользователь решил отметить это действие.

Важно понимать, что выполняемая задача должна сама по себе поддерживать прерывания

Для приостановки/продолжения работы потока в ОС, которые поддерживают такой функционал, используйте функции ОС и `native_handle()`. Например, в ОС Windows используйте `SuspendThread` и `ResumeThread`.


```

1 struct interrupt_flag {
2     void set();
3     bool is_set() const;
4 };
5 thread_local interrupt_flag this_thread_interrupt_flag;
6 class interruptible_thread {
7     std::thread internal_thread;
8     interrupt_flag* flag;
9 public: template<typename FunctionType>
10    interruptible_thread(FunctionType f) {
11        std::promise<interrupt_flag*> p;
12        internal_thread=std::thread([f,&p] {
13            p.set_value(&this_thread_interrupt_flag);
14            try {
15                f();
16            } catch(interruption_exception const &) {}
17        });
18        flag=p.get_future().get();
19    }
20    void interrupt() {
21        if(flag) {
22            flag->set();
23        }
24    }
25 };

```

Обнаружение прерывания

Реализуем функцию, проверяющую необходимость остановки потока

```
1 void interruption_point() {  
2     if(this_thread_interrupt_flag.is_set())  
3         throw interruption_exception;  
4 }
```

```
1 int main() {  
2     interruptible_thread th([]){  
3         byte data[2048];  
4         int received = receive_data(data);  
5         while(received) {  
6             interruption_point();  
7             received = receive_data(data);  
8         }  
9     });  
10    //...  
11    th.interrupt();  
12 }
```

Отлично, прерывать активные потоки мы умеем.

Но хотелось бы еще и прерывать потоки, которые находятся в режиме ожидания

Рассмотрим, как обрабатывать прерывания потоков, ожидающих сигнала условной переменной

```

1 template<typename Lockable>
2 struct custom_lock {
3     interrupt_flag* self;
4     Lockable& lk;
5
6     custom_lock(interrupt_flag* self_,
7                 std::condition_variable_any& cond,
8                 Lockable& lk_):
9         self(self_),lk(lk_)
10    {
11        self->set_clear_mutex.lock();
12        self->thread_cond_any=&cond;
13    }
14    void unlock() {
15        lk.unlock();
16        self->set_clear_mutex.unlock();
17    }
18    void lock() {
19        std::lock(self->set_clear_mutex,lk);
20    }
21    ~custom_lock() {
22        self->thread_cond_any = nullptr;
23        self->set_clear_mutex.unlock();
24    }
25 };

```

```

1 class interrupt_flag {
2     std::atomic<bool> flag;
3     std::condition_variable_any* thread_cond_any = nullptr;
4     std::mutex set_clear_mutex;
5
6 public:
7     void set() {
8         flag.store(true);
9         std::lock_guard<std::mutex> lk(set_clear_mutex);
10        if(thread_cond_any) {
11            thread_cond_any->notify_all();
12        }
13    }
14
15    template<typename Lockable>
16    void wait(std::condition_variable_any& cv, Lockable& lk) {
17        custom_lock cl(this, cv, lk);
18        interruption_point();
19        cv.wait(cl);
20        interruption_point();
21    }
22 };

```

Осталось реализовать функцию, которая позволяет потоку ожидать условную переменную с возможностью прерывания

```
1 template<typename Lockable>
2 void interruptible_wait(
3     std::condition_variable_any& cv,
4     Lockable& lk)
5 {
6     this_thread_interrupt_flag.wait(cv, lk);
7 }
```

Для реализации прерывания потоков, которые ожидают другие блокирующие сущности, будет полезно использовать методы `wait_for`

```
1 template<typename T>
2 void interruptible_wait(std::future<T>& fut)
3 {
4     while(!this_thread_interrupt_flag.is_set()) {
5         if(fut.wait_for(std::chrono::milliseconds(1)
6             == std::future_status::ready)
7         break;
8     }
9     interruption_point();
10 }
```

Обработка прерываний

Реализованное прерывание - это не что иное как исключение. Это исключение следует обрабатывать.

Можно перехватить и продолжить работу дальше. Можно перехватить и завершить работу функции, а следовательно, и потока.

Если же исключение выйдет за пределы функции потока, переданной в конструктор потока, то будет вызвана функция `std::terminate`, что, в свою очередь, приведет к завершению программы.

На следующих лекциях:

- Структуры данных, свободные от блокировок
- Сетевое взаимодействие
- ???
- ???
- Зачет

Самостоятельное изучение

- Реализуйте модификации пулов потоков, описанные **в лекции (жми здесь)**
- <https://habrahabr.ru/post/306332/>

Список литературы

- Уильямс - Параллельное программирование на C++ в действии
- <http://www.jlc.tcu.edu.tw/OS/932/threadmanager.pdf>
- https://www.cs.uic.edu/~jbell/CourseNotes/OperatingSystems/4_Threads.html
- <https://habrahabr.ru/post/306332/>