Лекция 11. Продвинутое управление потоками

ИУ8

November 17, 2016

Содержание

План лекции:

- пул потоков
- прерывание работы потоков

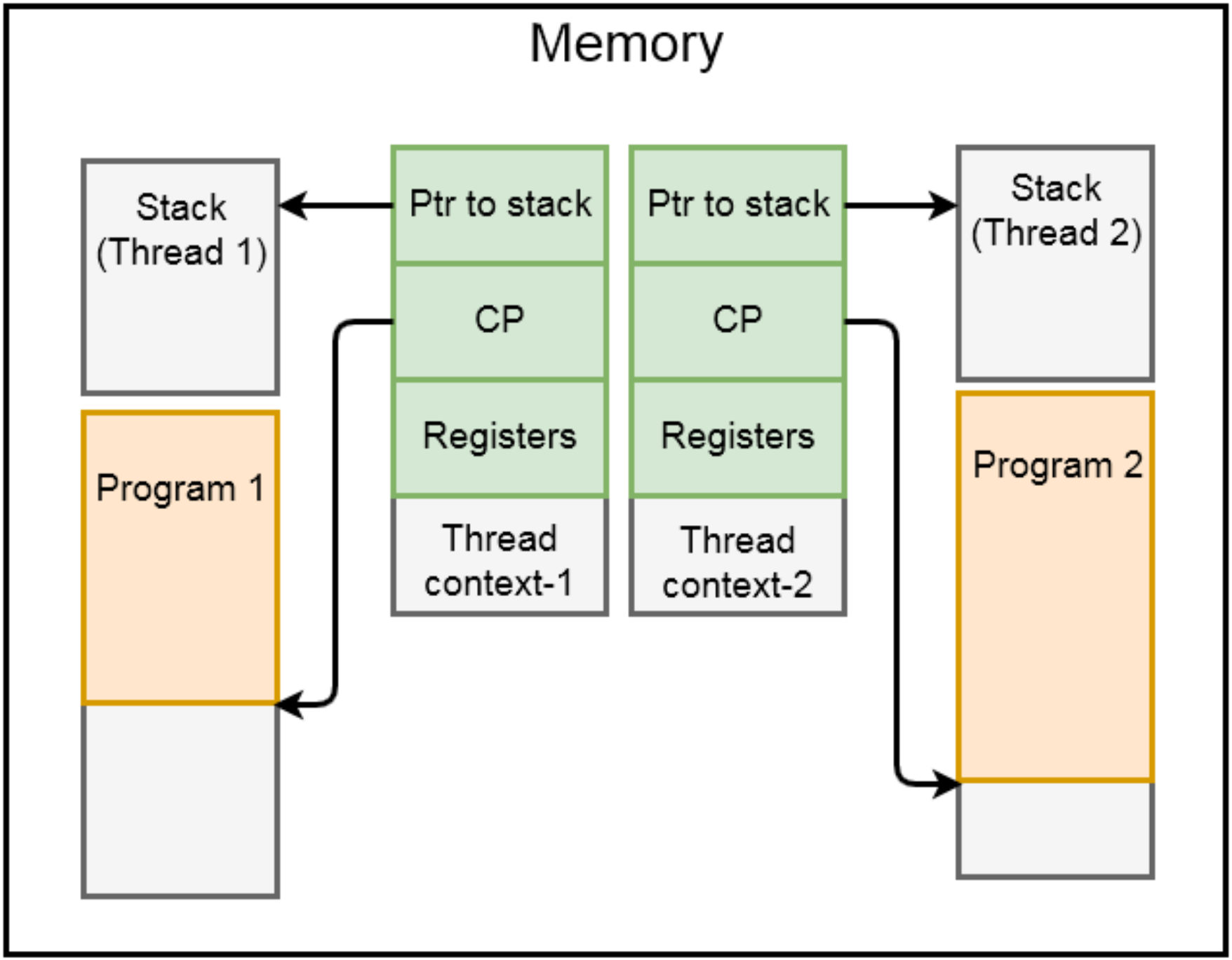
Объект потока

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
Stack (Thread 1)
Program 1
Ptr to stack
CP
Registers
Thread context-1
Ptr to stack
CP
Registers
Thread context-2
Stack (Thread 2)
Program 2

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

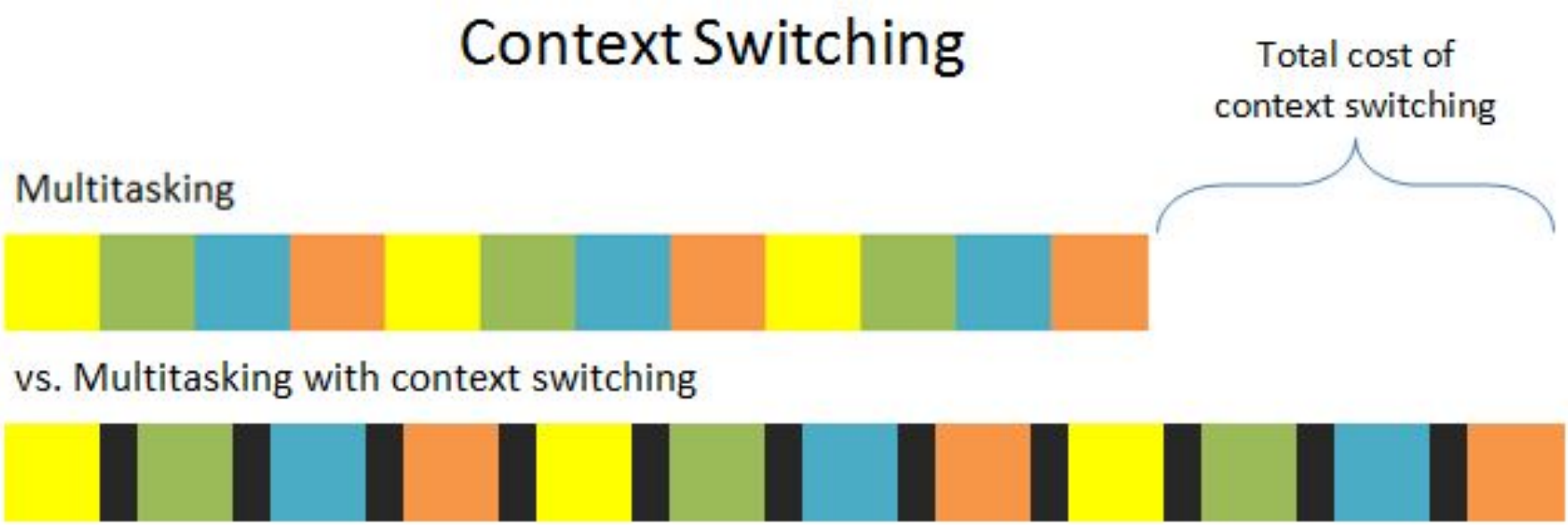
Идем дальше.

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.



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

```
class thread_pool {
  std::atomic_bool done = false;
  thread_safe_queue<std::function<void()>> work_queue;
  std::vector<std::thread> threads;

  void worker_thread(){
    while(!done.load()) {
      std::function<void()> task;
      if(work_queue.try_pop(task)) {
        task();
      } else {
        std::this_thread::yield();
      }
    }
  }
public :
  ~thread_pool() {
    done.store(true);
  }
```

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           thit));
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

Диначическое 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 имеет 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.

```cpp
struct interrupt_flag {
    void set();
    bool is_set() const;
};
thread_local interrupt_flag this_thread_interrupt_flag;
class interruptible_thread {
    std::thread internal_thread;
    interrupt_flag* flag;
public: template<typename FunctionType>
    interruptible_thread(FunctionType f) {
        std::promise<interrupt_flag*> p;
        internal_thread=std::thread([f,&p] {
           p.set_value(&this_thread_interrupt_flag);
           try {
              f();
           } catch(interruption_exception const &) {}
        });
        flag=p.get_future().get();
    }
    void interrupt() {
        if(flag) {
            flag->set();
        }
    }
};
```

Обнаружение прерывания

Реализуем функцию, проверяющую необходимость остановки потока

```
void interruption_point() {
  if(this_thread_interrupt_flag.is_set())
    throw interruption_exception;
}
```

```
int main() {
  interruptible_thread th([](){
    byte data[2048];
    int received = receive_data(data);
    while(received) {
      interruption_point();
      received = receive_data(data);
    }
  });
  //...
  th.interrupt();
}
```

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

```
template<typename Lockable>
struct custom_lock {
     interrupt_flag* self;
     Lockable& lk;

     custom_lock(interrupt_flag* self_,
        std::condition_variable_any& cond,
        Lockable& lk_):
     self(self_),lk(lk_)
     {
         self->set_clear_mutex.lock();
         self->thread_cond_any=&cond;
     }
     void unlock() {
         lk.unlock();
         self->set_clear_mutex.unlock();
     }
     void lock() {
         std::lock(self->set_clear_mutex,lk);
     }
     ~custom_lock() {
          self->thread_cond_any = nullptr;
          self->set_clear_mutex.unlock();
     }
};
```

```
class interrupt_flag {
    std::atomic<bool> flag;
    std::condition_variable_any* thread_cond_any = nullptr;
    std::mutex set_clear_mutex;

public:
    void set() {
        flag.store(true);
        std::lock_guard<std::mutex> lk(set_clear_mutex);
    if(thread_cond_any) {
            thread_cond_any->notify_all();
        }
    }

    template<typename Lockable>
    void wait(std::condition_variable_any& cv, Lockable& lk) {
        custom_lock cl(this, cv, lk);
        interruption_point();
        cv.wait(cl);
        interruption_point();
    }
};
```

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

```
template<typename Lockable>
void interruptible_wait(
  std::condition_variable_any& cv,
    Lockable& lk)
{
    this_thread_interrupt_flag.wait(cv,lk);
}
```

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.

Анонс

На следующих лекциях:

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

The end

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/