UNIVERSITÀ DEGLI STUDI FIRENZE

C++11 Thread Pool

Marco Di Rienzo

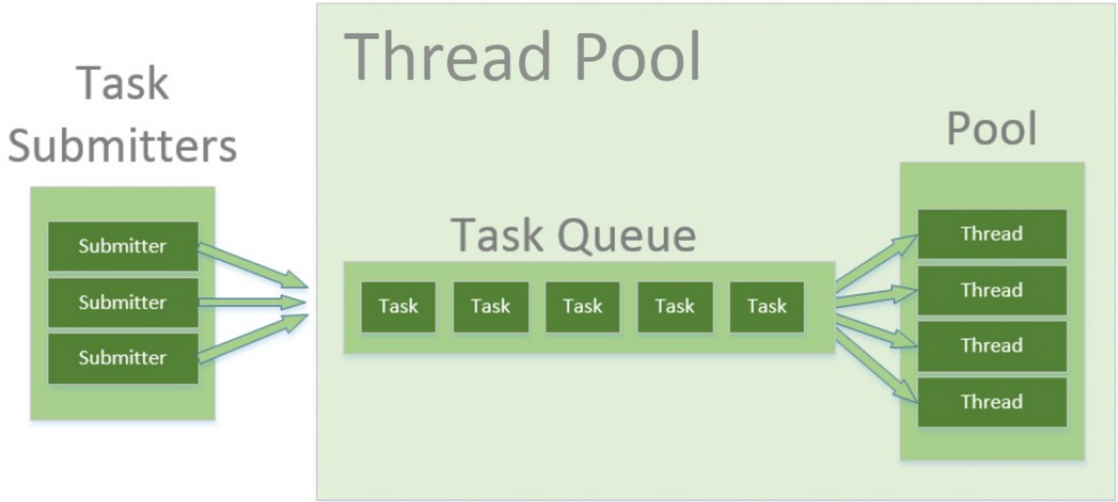
Mid-term assignment

Introduction

Thread pools

Thread pools provide a way
• to reduced per-task invocation overhead
• to manage resources, including threads, consumed when executing a collection of tasks

When you use a thread pool, you submit concurrent tasks for execution to an instance of a thread pool. This instance controls several reused threads for executing these tasks.



Advantages

• threads can be created beforehand, allowing to put a strict **upper limit** on the total number of threads and hence resources that are allocated concurrently

• threads of the pool are not destroyed until the pool itself is terminated and can be reused for multiple tasks, avoiding the overhead of creating a thread for each one of your tasks

• thread pools can also maintain some basic statistics such as the number of completed tasks

The trade-off is that once the pool is saturated, the execution of a new task will be **delayed** until a previous task is completed.

Abstract

The goal of this project is to implement a simple thread pool library in C++11.

The pool should have this features:

- fixed size
- threads of the pool may consume tasks as they become available
- be able to get a ***Future*** representing the task

The API of the library is inspired by the Java class **ThreadPoolExecutor.**

Classes

Runnable

The `Runnable` interface should be implemented by any class whose instances are intended to be executed by a `FixedThreadPool`. The class must define a method of no arguments called `run`.

Methods

`void run()`

Submitting an object implementing interface `Runnable` to a `FixedThreadPool` will cause the object's `run` method to be called in a separately executing thread.

FixedThreadPool

This class allows the creation of a pool that reuses a fixed number of threads operating off a **shared unbounded queue**.

At any point, at most a fixed number of threads will be active processing tasks.

If additional tasks are submitted when all threads are active, they will wait in the queue until a thread is available.

The threads in the pool will exist until it is explicitly shutdown.

UNIVERSITÀ DEGLI STUDI FIRENZE

Methods (refer to the paper for a complete list)

```
void execute(Runnable *command)
```

Executes the given command at some time in the future. The command will be executed by a thread of the pool.

```
template<class T>
std::future<T> submit(Runnable *task, T result)
```

Submits a `Runnable` task for execution and returns a *Future* representing that task. The *Future*'s `get` method will return the given result upon successful completion.
The task will be executed by a thread of the pool.

Implementation

Thread loop

Algorithm 1: Thread loop

```
while true do
    acquire pool shared queue lock;
    while pool not terminated and queue is empty do
        // block thread until wakeup
        wait(lock);
    end
    if pool terminated and queue is empty then
        return;
    end
    get first task in queue;
    release lock;
    execute task;
end
```

Every thread will wait for tasks to enter the shared queue, at which point one of them **synchronously** removes a task that can then be executed in parallel with others.

Execute

Algorithm 3: Execute

Data: $task$
acquire pool shared queue $lock$;
if *pool terminated* **then**
  // do not allow enqueing on terminated pool
  throw exception;
end
enqueue the $task$;
release $lock$;
// wake a thread up
notify();

The `execute` method acquires the *lock* to add the task to the shared *queue*, then notifies a thread that a job has been added.

Submit

Algorithm 4: Submit

Data: $task$, $result$
create an asynchronous operation that invokes the $task$ and provides a $future$ storing the $result$;
create a `RunnableFuture` wrapping the operation;
pass the `RunnableFuture` to `execute`;
return the $future$;

The `submit` method it is similar to the `execute` method but it returns a *Future* to the caller which is able to retrieve the result at a later time. To do this, a `std::future` representing the task is created and returned.
An *adapter* class `RunnableFuture` has been implemented to be able to pass the future operation to the `execute` method.