Project 1: Java Thread Pool in C++11

Marco Di Rienzo
marco.dirienzo@stud.unifi.it

Abstract

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

1. Introduction

1.1. Thread pools

Threads are used to execute concurrent tasks. It is possible to self manage those threads, but since creating and destroying threads requires a significant CPU usage, it may become problematic when you need to perform lots of small, simple tasks. The overhead of creating your own threads along with the high number of context switches can take up a significant portion of the CPU cycles and severely affect the final response time.
This is where thread pools become useful, they provide both a way to reduced per-task invocation overhead and 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.
The main advantages are that threads can be created beforehand, allowing to put a strict upper limit on the total number of threads and hence resources (in particular memory) that are allocated concurrently. Also the 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.

1.2. Design specifications

The developed library is entirely written in standard C++11 and allows the creation of a simple thread pool. In particular the pool should be of fixed size, that is the number of threads in the pool will not vary for the whole pool lifetime. Threads of the pool may consume tasks as they become available.
The API of the library is inspired by the Java class `ThreadPoolExecutor`.

2. Documentation

2.1. 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`.

2.1.1 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.

2.2. 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.

2.2.1 Constructor & Destructor

`FixedThreadPool``(int n_threads)`
Creates a `FixedThreadPool` with `n_threads` initial threads, the number of threads will remain constant for the whole pool lifetime.

`~FixedThreadPool`
Calls `shutdown` if pool is not already shut down.

2.2.2 Methods

`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.

`size_t` **`getPoolSize`**`() const`
Returns the current number of threads in the pool.

`size_t` **`getTaskCount`**`() const`
Returns the approximate total number of tasks that have ever been scheduled for execution.

`bool` **`isShutdown`**`() const`
Returns `true` if this pool has been shut down.

`void` **`shutdown`**`()`
Initiates an orderly shutdown in which previously submitted tasks are executed, but no new tasks will be accepted.

`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.

3. Implementation

A explanation on how non-trivial methods are implemented follows.

3.1. FixedThreadPool

The `FixedThreadPool` class implements some of the method of the Java class `ThreadPoolExecutor`.
To do this, it maintains a `std::vector` of threads of fixed size and a `std::queue` of tasks shared among those threads. A mutex lock is also necessary to synchronize reads and writes to the queue and to the pool state.
The uses of such resources by each of the implemented methods are explained next.

3.1.1 Instantiation

At pool instantiation, the specified number of threads will be created, each one of them running the same loop described by the pseudocode in Algorithm 1.
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.

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
```

3.1.2 Shutdown

The `shutdown` method corresponds to the `shutdown` method of the Java `ExecutorService` interface.
This method sets the pool to *terminated* state to allow threads waiting on empty queue to exit.
The pseudocode can be seen in Algorithm 2.

Algorithm 2: Shutdown

```
acquire pool shared queue $lock$;
set pool to $terminated$;
// this unlocks threads waiting on empty queue
release $lock$;
// wake all threads up
notify_all();
join all threads of the pool;
```

3.1.3 Execute

The `execute` method corresponds to the `execute` method of the Java `Executor` interface.
It acquires the *lock* to add the task to the shared *queue*, then notifies a thread that a job has been added, as shown in Algorithm 3.

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();
```

3.1.4 Submit

The `submit` method corresponds to the `submit` method of the Java `ExecutorService` interface.
As such, 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.
The pseudocode is described in Algorithm 4.

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 $future$;
```