



# Thread Pooling in C++

## Summary

Thread-pooling is used to reduce the number of threads created in a system (typically a server system of some sorts). Threads enter the pool, and are available when a new thread is required, rather than creating a new thread each time, which involves system overhead.

This document briefly describes the architecture of the `ThreadPool` class that permits very easy implementation of a reasonably simple thread pool. The `ThreadPool` class source code can be downloaded at [http://www.lateral.co.za/5mins/threadpool\\_windows.zip](http://www.lateral.co.za/5mins/threadpool_windows.zip) or [http://www.lateral.co.za/5mins/threadpool\\_linux.zip](http://www.lateral.co.za/5mins/threadpool_linux.zip).

Version	1.0, 15 February 2006
Language	C++, STL
Platform	Platform Independent
Tested On	Windows XP, Microsoft Visual C++ (.Net, not 2003), Ubuntu Linux, g++
Author	Craig Mason-Jones (craig@lateral.co.za)
Licence	<a href="http://creativecommons.org/licenses/by-nc-sa/2.0/za/">http://creativecommons.org/licenses/by-nc-sa/2.0/za/</a>

## Problem

Provide an extremely easy-to-use, but nonetheless flexible, thread pool implementation.

## Usage

Using the thread pool is as simple as creating a function that each thread will call. This function can take only one parameter (which can be a pointer to an struct or an object, so there is no limit to the data that can be passed), and cannot return any values, since there is no thread waiting to receive the return value.

Instantiate a `ThreadPool` with this constructor:

```
1  ThreadPool<parameter_type> pool ( thread_function, int minimum_threads, int maximum_threads, int
    maxExecutionCount );
```

To launch a thread from the pool, use the method:

```
2  pool.Launch( parameter );
```

The `parameter_type` passed here is the parameter type that the thread function will take. The thread function is passed as the first parameter to the pool. The `minimum_threads` is the minimum number of threads that the pool will keep available, while the `maximum_threads` is the maximum number that it will create if it receives many requests. `maxExecutionCount` is a count of the maximum number of function calls each thread can make before it will be killed, and a new thread created to replace it. This is probably unnecessary in a well-coded C++ system, but is there in case you think your threads might be leaking memory, and shutting them down would reclaim this.

Certainly in Java I once encountered a JDBC driver that leaked memory, and simply shutting it down after every few thousand queries reclaimed the memory perfectly. It was inelegant, but far faster than trying to get the vendor to fix the bug. `maximum_threads` can be set to

`ThreadPool<parameter_type>::UnlimitedThreads` if you don't want a cap on the number of threads. This is not recommended. `maxExecutionCount` can be set to

`ThreadPool<parameter_type>::UnlimitedLifetime` if you don't want your threads replaced ever.

## Example

This trivial example illustrates use of the ThreadPool:

```

3
4 #include "ThreadPool/ThreadPool.h"
5
6 void threadFunction(const char* param) {
7     Sleep(1000);
8     puts( param );
9 }
10
11 int _tmain(int argc, _TCHAR* argv[])
12 {
13     ThreadPool<const char*> pool (threadFunction,
14         1, // Minimum number of threads to maintain
15         2, //ThreadPool<const char*>::UnlimitedThreads,
16         1 //ThreadPool<const char*>::UnlimitedLifetime
17     );
18     pool.Launch("First Message");
19     pool.Launch("Second Message");
20     pool.Launch("Third Message");
21     pool.Launch("Fourth Message");
22     pool.Launch("Fifth Message");
23     pool.Launch("Final Message");
24     system("PAUSE");
25     return 0;
26 }

```

## Solution

The crux of this pool solution is to separate the pool's responsibilities from the threads. The pool is responsible for queuing the parameters that are passed to each call, and for awaking or creating threads as needed. Each thread, through the `ThreadWrapper` class, is responsible for fetching a parameter from the pool, processing the parameter (through the function that the pool manages), and rejoining the pool when it is complete.

When a `Launch` request is made of the `ThreadPool`, it determines whether there is an available thread to handle the function call, creating a new thread if the maximum number of threads has not been reached. The available thread (found or created), is awoken. The pool does not pass it the parameter or take any other action – the thread will fetch the parameter when it awakes. The pool merely queues the parameter to be handled by the next available thread.

When a thread awakes, or has finished processing a request, it joins the pool, via the `ThreadPool<>::Rejoin` method. If there is a queued parameter, the pool will pass this to the `ThreadWrapper` and return `ThreadPool::Execute` (an enum defined in `ThreadPool<>`). If there are no parameters to process, the pool will either ask the thread to sleep, by returning `ThreadPool::Sleep`, or, if the thread has executed its maximum execution count or is more than the minimum number of threads to maintain, `ThreadPool::Die`, which will cause the Thread to terminate itself.

## Cross-Platform

The `ThreadPool` class is cross-platform, insofar as it abstracts the essential OS-related issues to a class, either `Windows32Specific` or `PosixSpecific`. There are only three 'concepts' from the OS that are used: threads (rather obviously), Mutex's to prevent mutual execution of a specific block of code, and conditions (or Events in Windows terminology), to signal the occurrence of a particular condition.

## Linux

The Linux version required some tweaking, which is probably more a function of my Visual C++ being out of date, and not requiring `typename`'s, etc. I compiled it with:

```
27 g++ Mainu.cpp ThreadPool/OSSpecific_posix.cpp -lpthread -oThreadP
```

## Conclusion

I have used this thread pooling model very successfully in both a small standalone web-server I created, and in a Java incarnation, for a large SMS mobile communications platform. In both circumstances it has served me very well, allowing me to start the development using a single-threaded approach for ease of debugging, and switching to a thread pool with almost no code changes.

I am extremely interested if other programmers find this model useful, or whether they encounter any problems with it.