# ITMC403 Parallel and Distributed Computing

## Thread pools

# What is Thread Pools?

- The idea behind a thread pool is to set up a number of threads that sit idle, waiting for work that they can perform.
- As your program has tasks to execute, it encapsulates those tasks into some object (typically a *Runnable object*) and informs the thread pool that there is a new task.
- One of the idle threads in the pool *takes the task and executes it*; when it finishes the task, it goes back and waits for another task.
- Thread pools have a maximum number of threads available to run these tasks. Consequently, when you add a task to a thread pool, it might have to wait for an available thread to run it. That may not sound encouraging, but it's at the core of why you would use a thread pool.

# Reasons for using thread pools fall into three categories.

- **The first reason:** because the *overhead of creating a thread is very high*; by using a pool, we can gain some *performance* when the threads are reused. The degree to which this is true depends a lot on the program and its requirements. It is true that creating a thread can take as much as a few hundred microseconds, which is a significant amount of time for some programs.
- **The second reason:** it allows for *better program design*. If a program
- has a lot of tasks to execute, you simply create a task and send the task to the pool to be executed; this leads to much more elegant programs and lets you focus on the logic of your program
- **The third reason:** to use a thread pool is that *they carry important performance benefits* for applications that want to run many threads simultaneously. In fact, anytime you have more active threads than CPUs, a thread pool can play a crucial role in making your program *seem to run faster and more efficiently*.

# Thread Pools and Throughput

- *what does it mean that your program "seems" to run faster?*

It means that the *throughput* of your CPU-bound program running multiple calculations will be faster, and that leads to the perception that your program is running faster. *It's all a matter of throughput.*

***What is Throughput?***

***Answer:***

# Thread Pools and Throughput (Cont.)

Remember, **our first example**, we have three threads and one CPU. The three threads run at the same time, are time-sliced by the OS, and all completed execution in around 8 seconds.  produces this output:

```
Starting task Task 2 at 00:04:30:324
Starting task Task 0 at 00:04:30:334
Starting task Task 1 at 00:04:30:345
Ending task Task 1 at 00:04:38:052 after 7707 milliseconds
Ending task Task 2 at 00:04:38:380 after 8056 milliseconds
Ending task Task 0 at 00:04:38:502 after 8168 milliseconds
```

**In this case,**

assume that we have written this program as a server where each time a client connects, it is given a separate thread.

When the three clients each request the service (that is, the calculation of the Fibonacci number), each will wait 8 seconds for its answer.

# Thread Pools and Throughput (Cont.)

Remember, **our second example**, we have three threads and we run the threads sequentially, all completed execution in around 8 seconds and see this output:

```
Starting task Task 0 at 00:04:30:324
Ending task Task 0 at 00:04:33:052 after 2728 milliseconds
Starting task Task 1 at 00:04:33:062
Ending task Task 1 at 00:04:35:919 after 2857 milliseconds118
Starting task Task 2 at 00:04:35:929
Ending task Task 2 at 00:04:38:720 after 2791 milliseconds
```

**In this case,**

A server that runs the calculations sequentially will provide its first answer in 2.7 seconds, and the average waiting time for the clients will be 5.4 seconds. This is what we mean by the throughput of the program. In both cases, we've done the same amount of work, but in the second case, users of the program are generally happier with the performance.

# Thread Pools and Throughput (Cont.) Class Discussions

Discussions on **our second examples**,

we have three threads and we have written this program as a server where each time a client connects, it is given a separate new thread.

**Now consider these options:**

- **What happens** If we create a new thread for every client?
  **Answer:** the server could quickly become overloaded
  the server could quickly become overloaded: the more threads it starts, the slower it provides an answer for each request.
- **What happens** if we run the requests sequentially using only one thread? **Answer:** The server reaches a steady state.

**Consumer/Producer model**

With three requests in the queue, each subsequent request arrives as another one finishes. We can supply an endless number of answers to the clients; each client waits about eight seconds for a response.

# A Traditional I/O Server

- In this (blocking) I/O model, a network server must start a new thread for every client that attaches to the server. We solve the problem of blocking while we're waiting for data.
- Once a data connection has been negotiated, the server and client communicate through the private connection. This simplifies our server-side programming.