**Lecture 4 - Higher level multithreading concepts**

**Thread creation issues**

Creating many threads has two issues:

1. creating threads, and swithcing between threads, is expensive;
2. there is an OS-dependent upper limit on the number of threads.

This can be illustrated by running the program [vector\_sum\_multithread.cpp](https://www.cs.ubbcluj.ro/~rlupsa/edu/pdp/progs/vector_sum_multithread.cpp). It attempts to compute a vector sum, creating one thread for each element (or for every specified number of consecutive elements). Sample runs:

To avoid the OS-imposed limit on the number of threads, we can suspend creating new threads when some pre-configured maximum is reached, and to resume when some of them terminate. The resulting program is neither efficient nor maintanable: [vector\_sum\_limited\_thread.cpp](https://www.cs.ubbcluj.ro/~rlupsa/edu/pdp/progs/vector_sum_limited_thread.cpp).

**Thread pools**

The idea behind a thread pool is the following: instead of creating a new thread when we have some work to do and finish it when the work is done, we do the following:

* we pre-create a number of threads and have them wait (on a condition variable, so the SO doesn't give them CPU time yet;
* when some work comes in, we give it to a free thread;
* when the work is done, the thread returns to the waiting state;
* if work comes in, but all the threads are busy, there are two possibilities: either we temporarily increase the number of threads, or we just block waiting for a thread to finish its work. Note that increasing the number of threads runs the risk of reaching the OS limit, while blocking waiting for a thread to finish its work runs the risk of a deadlock.

An example, with a fixed size thread pool, is given at [vector\_sum\_thread\_pool.cpp](https://www.cs.ubbcluj.ro/~rlupsa/edu/pdp/progs/vector_sum_thread_pool.cpp).

**Producer-consumer communication**

**Futures and promises**

This is an easier mechanism to work with, compared to the condition variables. Essentially, we have an object that exposes two interfaces:

* On the *promise* interface, a thread can, a single time, set a value. This actions also marks the completion of an activity (a *task*), and the value is the result of that computation.
* On the *future* interface, a thread can wait for the value to become available, and retrieve that value. Thus, a future is a result of some future computation.
* It is also possible to set a *continuation* on a future (see next lecture).

Examples:

[futures-demo1.cpp](https://www.cs.ubbcluj.ro/~rlupsa/edu/pdp/progs/futures-demo1.cpp)

using futures to get the result from asynchronous tasks

[futures-demo1-with-impl.cpp](https://www.cs.ubbcluj.ro/~rlupsa/edu/pdp/progs/futures-demo1-with-impl.cpp)

as above, but with a possible implementation for futures and the async() call

**Producer-consumer queue**

See the examples:

[producer-consumer.cpp](https://www.cs.ubbcluj.ro/~rlupsa/edu/pdp/progs/producer-consumer.cpp)

threads communicating through producer-consumer queues

[ProducerConsumer.java](https://www.cs.ubbcluj.ro/~rlupsa/edu/pdp/progs/ProducerConsumer.java)

same, but in Java

[producer-consumer2.cpp](https://www.cs.ubbcluj.ro/~rlupsa/edu/pdp/progs/producer-consumer2.cpp)

same, but the queues have limited length and enqueueing blocks if the queue is full