ECE 3574: Applied Software Design

Threads

Today we are going to start looking at threads, multiple executing programs within the same process that share the code segment and heap, but have separate stacks.

- Threads and OS scheduling
- C++11 Threads
- C++11 `std::async`
- Examples

Threads

A *thread* is a single executing sequence, consisting of memory (code segment, stack, heap) and state (program counter, registers).
A process has one or more threads, that share the code segment and heap, but have *separate* stacks, and state.

A typical memory layout of a process with two threads

```
   Memory
+----------+ (low address)
|   code   |
+----------+
|   heap   |
|    |     |
|    v     |
|    ^     |
|    |     |
| stack 2  |<-------------- thread 2's stack
|    ^     |
|    |     |
| stack 1  |<-------------- thread 1's stack
+----------+ (high address)
```

Process versus Threads

Threads get scheduled by the OS
On most OS's there is no real distinction between threads and processes in the kernel scheduler (e.g. both are *tasks*).

- From the schedulers perspective a process is just a single thread.
- From the virtual memory manger's perspective the address space layout *is* different.

This means creation and context switches are faster for threads.
Heap memory is also shared by default, so no shm_create/size/map/unmap/close required!

C++11 Threads

All programming languages define details relative to an *abstract machine*.
Prior to C++11 threading was platform specific (e.g. posix-threads versus win-threads), the abstract machine was single-threaded.
In C++11 this abstraction was extended to include a memory model that allows the specification to say what a compiler can and cannot do relative to memory.

- Atomics were added
- And a standard thread library was added

We will use it to explore multithreading, then talk about other threading implementations (pthreads and QThread) later.

Creating a thread

```
#include <thread>
class thread;
```

Creates thread objects, each represents a single (unique) thread of execution.

- cannot copy construct a thread
- cannot copy assign a thread
- you can move construct and move assign a thread

Each thread has a unique `id` within the process:
`std::this_thread::get_id()`

A thread object has an associated function where execution begins

- This is like “main” for the thread. In fact main is the entry point for thread 0.
- The function can take arguments which are passed to the thread constructor.
- Constructing a thread object starts the thread.

After starting a thread you must wait for it to complete before the main thread exits.

To wait in a thread to finish (by exiting the thread entry function) you call the join method.
See `hello_thread1.cpp` and `hello_thread2.cpp`.

Threads share the heap.

Each thread shares the heap, giving them implicit shared memory.
We will see how to coordinate this in the next few meetings.
For now, lets see how to use threads as asynchronous functions, using higher-level concepts than the heap.

Promises and Futures

Conceptually a thread has an input channel and and output channel.
`#include <future>`

- The input channel is represented by a *promise*:
  `std::promise<T>`
- The output channel is represented by a *future*:
  `std::future<T>`

Both are objects that have a shared state (empty, ready, invalid), synchronized automatically by the C++ library.

Example: returning a value from a thread.

We can pass arguments to the thread when we create it. How do we return a value from the thread?
We use a promise/future together.
See example: `promise_future.cpp` (example from Bartosz Milewski)

What about exceptions thrown in a thread?

If not caught in the thread, the program terminates.
But, we can return exceptions in the future, which gets re-thrown automatically by the get call.
See example: `thread_exceptions.cpp`.

Promises/Futures are nice, but there is a lot of boilerplate in the code.

The function `std::async` allows us to just return values from the thread function like from any function, *and* does the thread creation/join for us!
See example: `async_future.cpp`

Thread local storage

Recall the three standard storage specifiers in C++.

- automatic: exists in block/function scope
- static (global): exists during the entire program
- dynamic: exists between new/delete calls

C++11 adds one more: thread_local meaning it exists during thread execution.
This is like a global for each thread, so has limited uses.

How many threads can I run?

You can get a hint to how many threads can run in parallel using `std::thread::hardware_concurrency()`.
See `how_many.cpp`.
Note it can *sometimes* be advantageous to run more threads than the hardware supports, in particular if the task is I/O bound.
However, as we will see there is a point of diminishing returns.

Example: asynchronous search version 1

Lets implement a concurrent binary search.
See `concurrent_binary_search.cpp`.
Why isn't this faster?

Example: asynchronous search version 2

Lets use a slower (linear) search.
See `concurrent_linear_search.cpp`.
Still not faster?

Example: Concurrent Merge-sort

See `serial_mergesort.cpp` and `concurrent_mergesort.cpp`.
Note: too many threads is bad for performance.
More is not always better.
Concurrent performance often requires adapting based on problem size.

Next Actions and Reminders

- Read about Locking and *Semaphores*