CS 220: Introduction to Parallel Computing

Introduction to pthreads

Lecture 25

Threads

- In computing, a **thread** is the smallest schedulable unit of execution
  - Your operating system has a **scheduler** that decides when threads/processes will run
- Threads are essentially lightweight processes
- In MPI, we duplicated our processes and ran them all at the same time (**in parallel**)
- With pthreads, a single process manages multiple threads

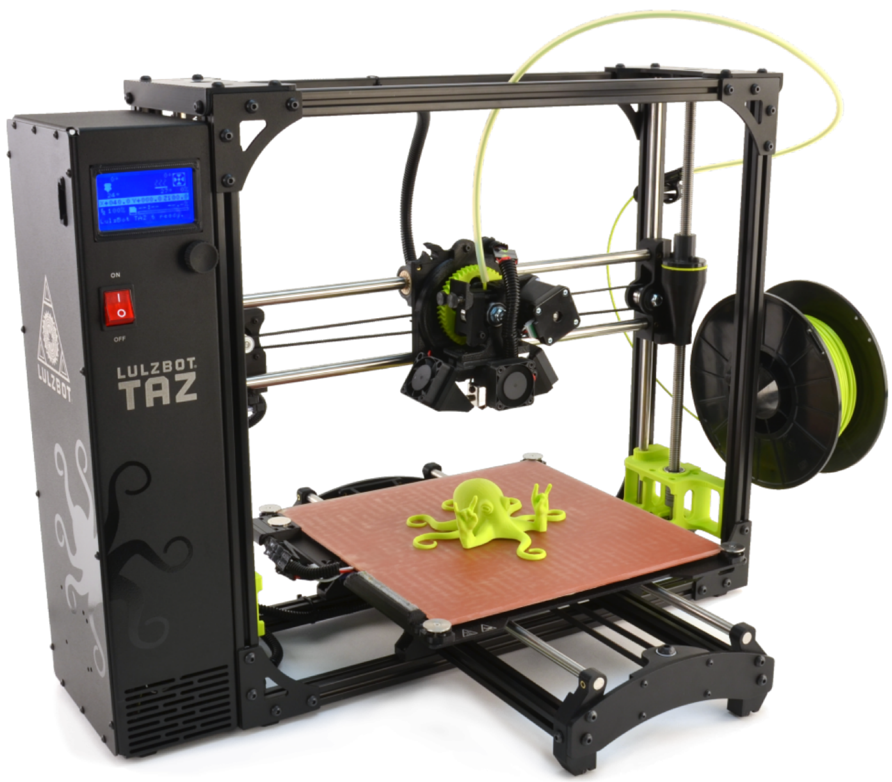
Why Learn Threads?

- Threads are the most important concept you will learn in this class
- In the past, you could get away with writing serial programs
- Today, we live in a world of asynchronous, multi-threaded code
  - Crucial for building fast, efficient applications

MPI: Photocopier

- MPI executes multiple processes in parallel
- When we specify -n 8, we'll get 8 processes
  - MPI will also **distribute** them across machines
- Each process gets a unique **rank**
  - Helps us divide workload

Threads: 3D Printer



- Each thread can be unique and do something different
  - Or you can make many threads that all do the same thing
- More flexible than MPI
- Also can be more difficult to manage

* Note: I may be **slightly** overselling threads here…

Posix Threads

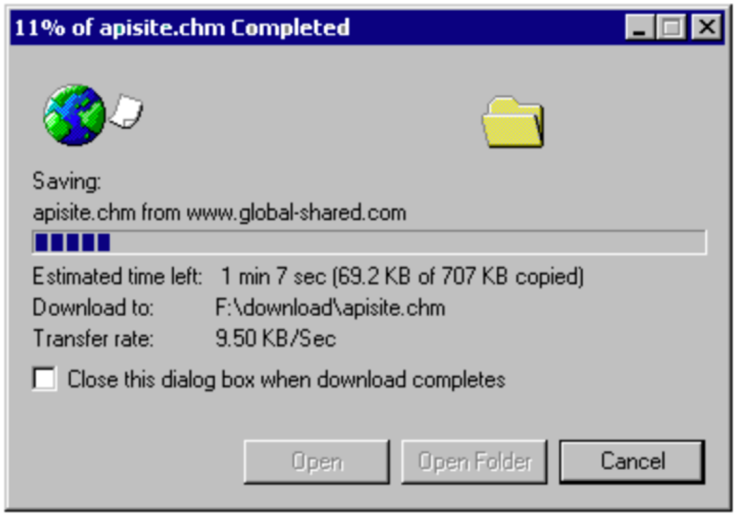
- pthreads is short for POSIX Threads
  - POSIX - Portable Operating System Interface
- POSIX is an operating system standard that helps ensure compatibility across systems
  - If your program conforms to the POSIX standard, then it'll compile and run on any compliant OS
- For instance, we can compile your C programs on macOS, Linux, FreeBSD, Solaris, and more

What are Threads?

- Lightweight processes
- Created by processes to do some subset of the work
- Rather than passing messages, threads use **shared memory**.
  - All the threads have access to internal variables, whereas with MPI we had to explicitly send our state information to another process

Common Uses for Threads (1/2)

- You may want your program to do two things at the same time
- For example, download a file in one thread and show a progress bar and dialog with another
- User interfaces are often multi-threaded
  - Helps hide the fact that CPUs can only do one thing at a time



Common Uses for Threads (2/2)

- Games often have a main **event loop** and several sub threads that handle:
  - Graphics rendering
  - Artificial Intelligence
  - Responding to player inputs
- In a video encoder, you may split the video into multiple regions and have each thread work on them individually

Stepping Back: Processes

- Recall: a process is an instance of a program
- Each process has:
  - Binary instructions, data, memory
  - File descriptors, permissions
  - Stack, heap, registers
- Threads are very similar, **but** they share almost everything with their **parent** process except for:
  - Stack
  - Registers

Sharing Data

- Since threads share the heap with their parent process, we can share pointers to memory locations
- A thread can read and write data set up by its parent process
- Sharing these resources also means that it's faster to create threads
  - No need to allocate a new heap, set up permissions, etc.

Other Types of Threads

- pthreads is just one way to manage lightweight execution contexts
- Windows has its own threading model
- Languages have other features: Go has **goroutines** that abstract away some threading details
  - C#: async/await
  - Futures
- Learning pthreads will help you understand how these models work

Getting Started with pthreads

- As usual, we have a new #include!
- `#include <pthread.h>`
- We also need to link against the pthreads library:
  - `gcc file.c -pthread`

- Luckily, we don't need a special compiler wrapper to use pthreads (like we did with MPI: mpicc)

Creating a Thread

```c
int pthread_create(
        pthread_t *thread,
        const pthread_attr_t *attr,
        void *(*start_routine)(void *),
        void *arg);
```

Let's Demo This…

Variable Access

- num_threads, defined at the top of the source file, is accessible by all the threads
  - This is a **global variable**
- Variables defined within the thread's function are private and only accessible by it
  - Remember: each thread gets its own stack
- If we malloc a struct on the heap and pass it to a thread, can it access the struct?

pthread_t

- What's pthread_t, the type we used to create our array of threads?
- This is considered an **opaque type**, defined internally by the library
  - It's often just an integer that uniquely identifies the thread, but we can't rely on this
  - For example, we shouldn't print out a pthread_t

attr

- The second parameter we pass in is the pthread attributes
- These can include the stack size, scheduling policies, and more
- For now we are fine with the defaults, so we pass in NULL

start_routine

- The most important part of pthread_create is the **start routine**
- This function is called by the pthread library as the starting point for your thread
  - Passed in as a **function pointer**
    - Pointers are back, whoo hoo!!!
    - Just like how they sound: they're a pointer to a specific function

arg

- The last argument to pthread_create is “arg”
- This can be anything we want to pass to the thread
- If we wanted to have MPI-style ranks, we can pass in a rank here
- If we were implementing P2 with pthreads, we’d want to pass in the start and end points of our mining thread

pthread_join

- `int pthread_join(pthread_t thread, void **value_ptr);`

- The pthread_join function waits for a pthread to finish execution (by calling **return**)
  - The return value of the thread is stored in **value_ptr**
- This lets our main thread wait for all its children to finish up before moving on
- Commonly used to coordinate shutting down the threads, waiting for their results, and synchronizing our logic