CS 521: Systems Programming

Introduction to Threads

Lecture 11

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
- With `fork`, we created clones of a process
- With pthreads, a single process manages multiple threads

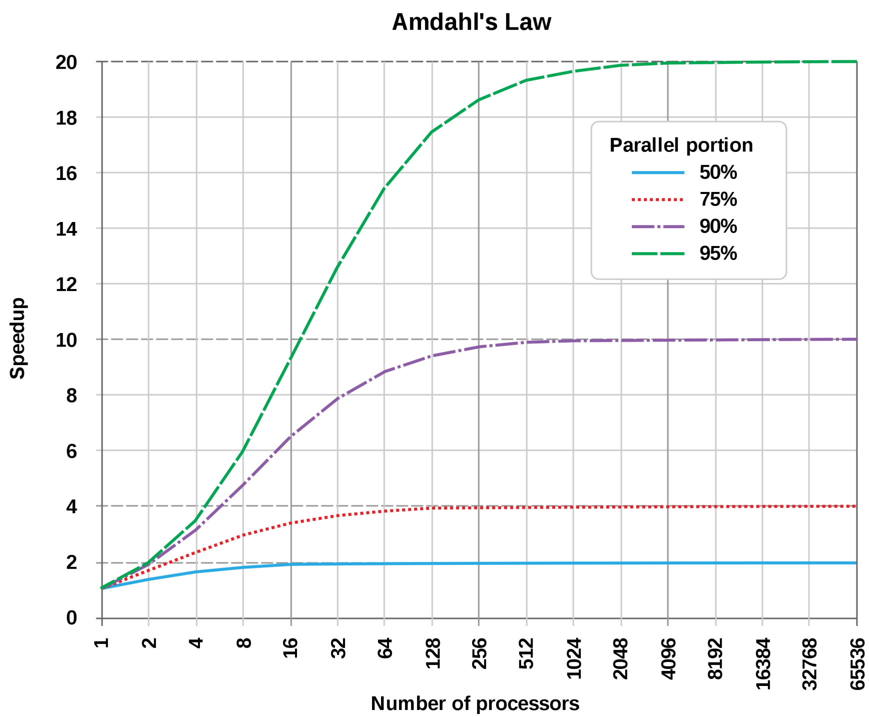
Why Learn Threads?

- Threads are one of the most important concepts for modern programmers to understand
- 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
- Multi-threading is **hard**, but many languages/frameworks are trying to address this

Amdahl's Law [1/2]

- In the best case scenario, doubling the number of cores/CPUs/processing units will halve execution time
  - In practice, this is difficult
- There is overhead associated with parallelism
- Amdahl's law puts a bound on potential speedup:
$$S_N = \frac{1}{(1-P)+\frac{P}{N}}$$
  - S – speedup
  - P – parallelizable portion of the program
  - N – number of cores/CPUs/processing units

Amdahl's Law [2/2]



A Departure from Cloning

- With `fork`, we simply cloned our parent process
  - No data sharing between them
- With threads, each individual thread can be unique and do something different
  - …or you can make many threads that all do the same thing
  - Threads have a variety of ways to coordinate and communicate

pthreads

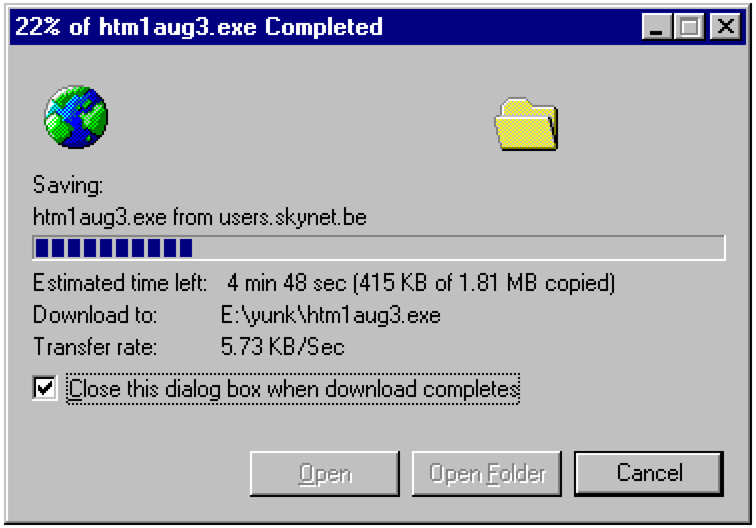
- ‘pthreads’ is short for POSIX Threads
  - POSIX - Portable Operating System Interface
  - (the spec most Unix-like operating systems follow)
- If your program conforms to POSIX, then it’ll compile and run on any compliant OS
- For instance, your pthread-enabled program will work on macOS, Linux, FreeBSD, Solaris, and more
  - Kinda… sorta. Most of the time.

What are Threads?

- Lightweight processes
- Created by processes to do some subset of the work
- In general, threads use **shared memory** to communicate
- All the threads have access to global/heap variables, whereas with separate processes we had to explicitly send information (maybe via pipes or sockets)

Use Cases [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



Use Cases [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
  - Java threads: basics are very similar to pthreads

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
```

  - You might see `-lpthread` out in the wild, but modern compilers expect `-pthread` instead

Functions We’ll Cover Today

- `pthread_create`
- `pthread_join`
- `pthread_detach`
- `pthread_exit`

Creating a Thread

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

- `thread` – populated by `pthread_create`, contains thread information
- `attr` – scheduling attributes
  - stack size, scheduling policies, etc.
  - to use the defaults, pass in `NULL`
- `start_routine` – function to run (function pointer)
- `arg` – argument to the function (passed as a `void *`)

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

The 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
    - Just like how they sound: they're a pointer to a specific function
      - We did this with `qsort` comparators

Passing Arguments

- The last argument to `pthread_create` is “arg”
- This can be anything we want to pass to the thread
  - One common pattern: pass in a thread “ID number” (called a *rank* in parallel computing)
    - Have threads coordinate based on these ranks (for example, *rank 0* may gather up the result(s) of the computations from other ranks)
- Want to pass in more than one argument? Use a struct

Joining Threads

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

- The pthread_join function waits for a pthread to finish execution (when it calls 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
  - Often used to coordinate shutting down the threads, waiting for their results, and synchronizing logic
  - Similar to `wait()` / `waitpid()`

Detaching Threads

```
int pthread_detach(pthread_t thread);
```

- Normally, threads will continue to live on until they are joined
  - (even if they have exited)
- Joining cleans up the resources allocated to the thread (such as its stack)
- Sometimes we can't join threads. By detaching them, we tell the OS to clean them up once they exit

pthread_exit

- Any guesses what `pthread_exit` does?
- You're right, it launches your default web browser!
  - Note: if you're skimming these slides to cram for an quiz later, that was only a joke
- `pthread_exit` also has a weird/handy property: if you call it from the main thread, it won't exit until the rest of the threads are finished
  - Nice if you want to wait for your workers to finish
  - What happens if you just call `exit()` or return from main?

Multi-Process or Multi-Thread?

- In general, using multiple processes is simpler and easier to implement
  - Split up the problem, have the processes work on it independently
- However, if the algorithm you are implementing requires lots of communication or shared state, threads are a better choice
  - They are also faster to create and require fewer system resources

Thread Creation Overhead

- On Solaris, creating a process is 30x slower than creating a thread, and context switches are 5x slower
- Each operating system has different overhead associated with processes/threads
  - Windows: huge process overhead. Use threads where possible
  - Linux: relatively low process creation overhead

Scalability

- Remember: single-threaded (or single-process) applications cannot run on more than one CPU
- This impacts scalability: the ability of your algorithm to run faster when given more resources
- Threads are a great way to take advantage of more cores to carry out background tasks and make applications more responsive