ECE 344: Operating Systems
Lecture 8

Threads Implementation

1.1.1

Jon Eyolfson
September 26, 2022

Multithreading Models

Where do we implement threads?

- We can either do user or kernel threads

User threads are completely in user-space

- Kernel doesn't treat your threaded process any differently

Kernel threads are implemented in kernel-space

- Kernel manages everything for you, and can treat threads specially

Thread Support Requires a Thread Table

Similar to the process table we saw previously

- It could be in user-space or kernel-space depending

For user threads, there also needs to be a run-time system to determine scheduling

In both models each process can contain multiple threads

We Could Avoid System Calls, or Let a Thread Block Everything

For pure user-level threads (again, no kernel support):

- Very fast to create and destroy, no system call, no context switches
- One thread blocks, it blocks the entire process (kernel can’t distinguish)

For kernel-level threads:

- Slower, creation involves system calls
- If one thread blocks, the kernel can schedule another one

All Threading Libraries You Use Run in User-mode

The thread library maps user threads to kernel threads

Many-to-one: threads completely implemented in user-space
the kernel only sees one process

One-to-one: one user thread maps directly to one kernel thread
the kernel handles everything

Many-to-many: many user-level threads map to many kernel level threads

Many-to-one is Pure User-space Implementation

It’s fast (as outlined before) and portable

- It doesn’t depend on the system, it’s just a library

Drawbacks are that one thread blocking causes all threads to block

- Also we cannot execute threads in parallel
  - The kernel will only schedule a process to run

One-to-one Just Uses the Kernel Thread Implementation

There's just a thin wrapper around the system calls to make it easier to use

Exploits the full parallelism of your machine

- The kernel can schedule multiple threads simultaneously

We do however need to use a slower system call interface,

- and we lose some control

Typically this is the actual implementation used, we'll assume this for Linux

Many-to-many is a Hybrid Approach

The idea is that there are more user-level threads than kernel-level threads

- Cap the number of kernel-level threads to the number we could run in parallel

We can get the most out of multiple CPUs and reduce the number of system calls

However, this leads to a complicated thread library

- Depending on your mapping luck, you may block other threads

Threads Complicate the Kernel

How should `fork` work with a process with multiple threads?

- Copy all threads to the new process, in whatever state they're in?
  - How would this get out of hand?

Linux only copies the thread that called `fork` into a new process

- If it hits `pthread_exit` it'll always exit with status 0
  - (at least as far as I can tell)

There's `pthread_atfork` (not covered in this course) to control what happens

Signals are Sent to a Process

Which thread should receive a signal? all of them?

Linux will just pick one random thread to handle the signal

- Makes concurrency hard, any thread could be interrupted

Instead of Many-to-many, You Can Use a Thread Pool

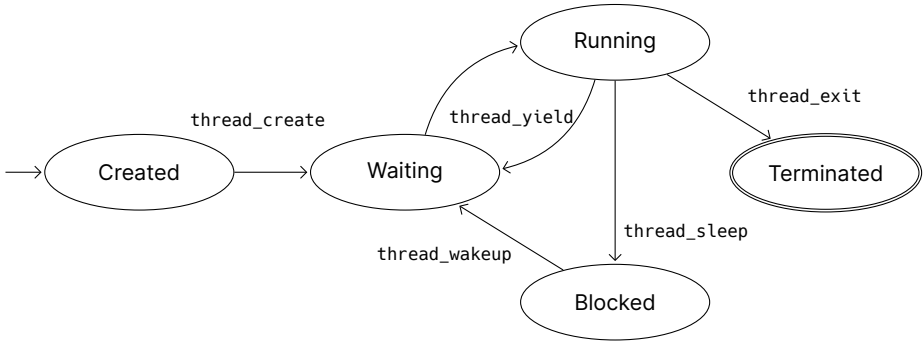
The goal of many-to-many thread mapping is to avoid creation costs

A thread pool creates a certain number of threads and a queue of tasks
(maybe as many threads as CPUs in the system)

As requests come in, wake them up and give them work to do

Reuse them, when there's no work, put them to sleep

You'll Implement Many-to-one



Your Scheduler Can Just Be Round Robin

Create a queue (list), run the thread at the front, when it yeilds at it to the back

You'll have to do the context switch (remember, you'll have to save the registers)

These are cooperative threads, so they have to be nice (next is preemptive threads)

Our Next Complication

Let's create a program that spawns 8 threads

- Each thread increments the same variable 10,000 times

What should the final value of the variable be?

- The initial value of the variable is 0

Run `lecture-08/pthread-datarace`

- Can you fix it?

Both Processes and (Kernel) Threads Enable Parallelization

- Each process can have multiple (kernel) threads
- Most implementations use one-to-one user-to-kernel thread mapping
- The operating system has to manage what happens during a fork, or signals
- We now have synchronization issues