

# CS 111: Operating System Principles

## Lecture 13

# Threads

1.0.1

Jon Eyolfson

May 4, 2021



This work is licensed under a [Creative Commons Attribution-ShareAlike 4.0 International License](https://creativecommons.org/licenses/by-sa/4.0/)

## Threads are Like Processes with Shared Memory

The same principle as a process, except by default they share memory  
They have their own registers, program counter, and stack

They have the same address space, so changes appear in each thread

You need to explicitly state if any memory is specific to a thread (TLS)

## One Process Can have Multiple Threads

By default a process just executes code in its own address space

Threads allow multiple executions in the same address space

They're lighter weight and less expensive to create than processes  
They share code, data, file descriptors, etc.

## Assuming One CPU, Threads Can Express Concurrency

A process can appear like it's executing in multiple locations at once  
However, the OS is just context switching within a process

It may be easier to program concurrently  
e.g., handle a web request in a new thread

```
while (true) {  
    struct request *req = get_request();  
    create_thread(process_request, req);  
}
```

## Threads are Lighter Weight than Processes

### **Process**

Independent code / data / heap

Independent execution

Has its own stack and registers

Expensive creation and context switching

Completely removed from OS on exit

### **Thread**

Shared code / data / heap

Must live within an executing process

Has its own stack and registers

Cheap creation and context switching

Stack removed from process on exit

When a process dies, all threads within it die as well!

## We'll be Using POSIX Threads

For Windows, there's a Win32 thread, but we're going to use \*UNIX threads

```
#include <pthread.h> – in your source file
```

```
-pthread – compile and link the pthread library
```

All the pthread functions have documentation in the `man` pages

## You Create Threads with `pthread_create`

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

<code>thread</code>	creates a handle to a thread at pointer location
<code>attr</code>	thread attributes (NULL for defaults, more details later)
<code>start_routine</code>	function to start execution
<code>arg</code>	value to pass to <code>start_routine</code>

returns 0 on success, error number otherwise (contents of `*thread` are undefined)

## Creating Threads is a Bit Different than Processes

```
#include <pthread.h>
#include <stdio.h>

void* run(void*) {
    printf("In run\n");
    return NULL;
}

int main() {
    pthread_t thread;
    pthread_create(&thread, NULL, &run, NULL);
    printf("In main\n");
}
```

What are some differences? Are we missing anything?



## The wait Equivalent for Threads – Join

```
int pthread_join(pthread_t thread,  
                void** retval)
```

`thread` wait for this thread to terminate (thread must be joinable)

`retval` stores exit status of thread (set by `pthread_exit`) to the location pointed by `*retval`. If cancelled returns `PTHREAD_CANCELED`. `NULL` is ignored.

returns 0 on success, error number otherwise

**Only call this one time per thread!**

Multiple calls on the same thread leads to undefined behavior

## Previous Example that Waits Properly

```
#include <pthread.h>
#include <stdio.h>

void* run(void*) {
    printf("In run\n");
    return NULL;
}

int main() {
    pthread_t thread;
    pthread_create(&thread, NULL, &run, NULL);
    printf("In main\n");
    pthread_join(thread, NULL);
}
```

Now we joined, the thread's resources are cleaned up

## Ending a Thread Early (Think of `exit`)

```
void pthread_exit(void *retval);
```

`retval` return value passed to function that calls `pthread_join`

Note: `start_routine` returning is equivalent of calling `pthread_exit`  
Think of the difference between returning from `main` and `exit`

`pthread_exit` is called implicitly when the `start_routine` of a thread returns

## Detached Threads

*Joinable* threads (the default) wait for someone to call `pthread_join` then they release their resources

*Detached* threads release their resources when they terminate

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

`thread` marks the thread as detached

returns 0 on success, error number otherwise

Calling `pthread_detach` on an already detached is undefined behavior

## Detached Threads Aren't Joined

```
#include <pthread.h>
#include <stdio.h>

void* run(void*) {
    printf("In run\n");
    return NULL;
}

int main() {
    pthread_t thread;
    pthread_create(&thread, NULL, &run, NULL);
    pthread_detach(thread);
    printf("In main\n");
}
```

This code just prints “In main”, why?

## pthread\_exit in main Waits for All Detached Threads to Finish

```
#include <pthread.h>
#include <stdio.h>

void* run(void*) {
    printf("In run\n");
    return NULL;
}

int main() {
    pthread_t thread;
    pthread_create(&thread, NULL, &run, NULL);
    pthread_detach(thread);
    printf("In main\n");
    pthread_exit(NULL);
}
```

This code now works as expected

## You Can Use Attributes To Get/Set Thread Variables

```
size_t stacksize;
pthread_attr_t attributes;
pthread_attr_init(&attributes);
pthread_attr_getstacksize(&attributes, &stacksize);
printf("Stack size = %i\n", stacksize);
pthread_attr_destroy(&attributes);
```

Running this should show a stack size of 8 MiB (on most Linux systems)

You can also set a thread state to joinable

```
pthread_attr_setdetachstate(&attributes,
                           PTHREAD_CREATE_JOINABLE);
```

## 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 the blocks 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 while reducing 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

## 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 `examples/lecture-13/pthread-datarace`

Can you fix it?

## Both Processes and (Kernel) Threads Enable Parallelization

We explored threads, and related them to something we already know (processes)

- Threads are lighter weight, and share memory by default
- 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