Lecture 8 — Threads

Jeff Zarnett
jzarnett@uwaterloo.ca

Department of Electrical and Computer Engineering
University of Waterloo

April 26, 2015

What is a Thread?

Recall our earlier examination of the process.

A process has three major components:

1. An executable program;
2. The data created/needed by the program; and
3. The execution context of the program.

A process has at least one thread, and can have many.

What is a Thread?

The term “thread” is a short form of Thread of Execution.

A thread of execution is a sequence of executable commands that can be scheduled to run on the CPU.

Threads also have some state and stores some local variables.

Most programs you will write in other courses had only one thread;
that is, your program’s code is executed one statement at a time.

Multithreaded Programs

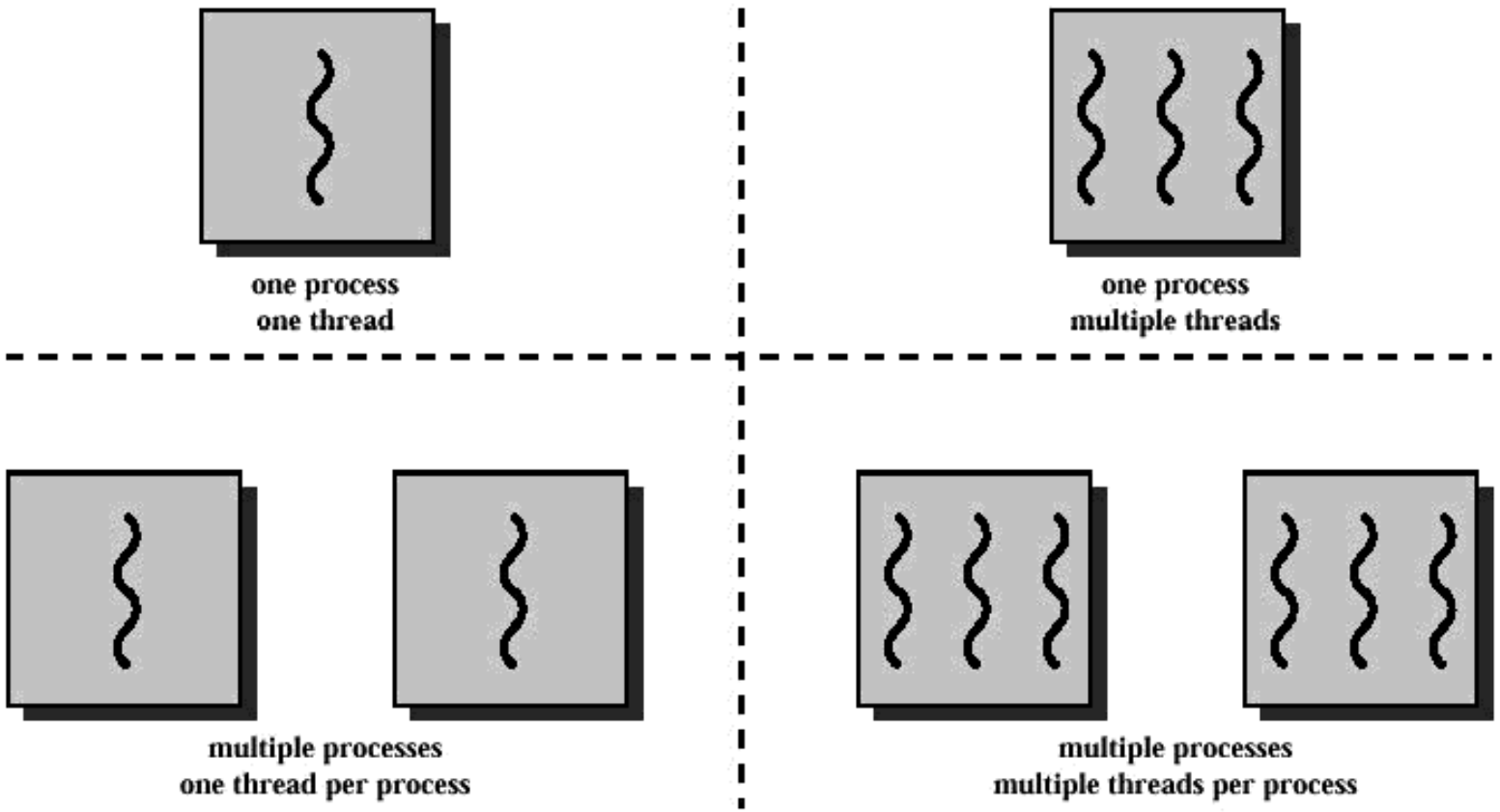
A multithreaded program uses more than one thread, (some of the time).

A program begins with an initial thread (where the `main` method is).

That main thread can create some additional threads if needed.

Threads can be created and destroyed within a program dynamically.

Threads and Processes

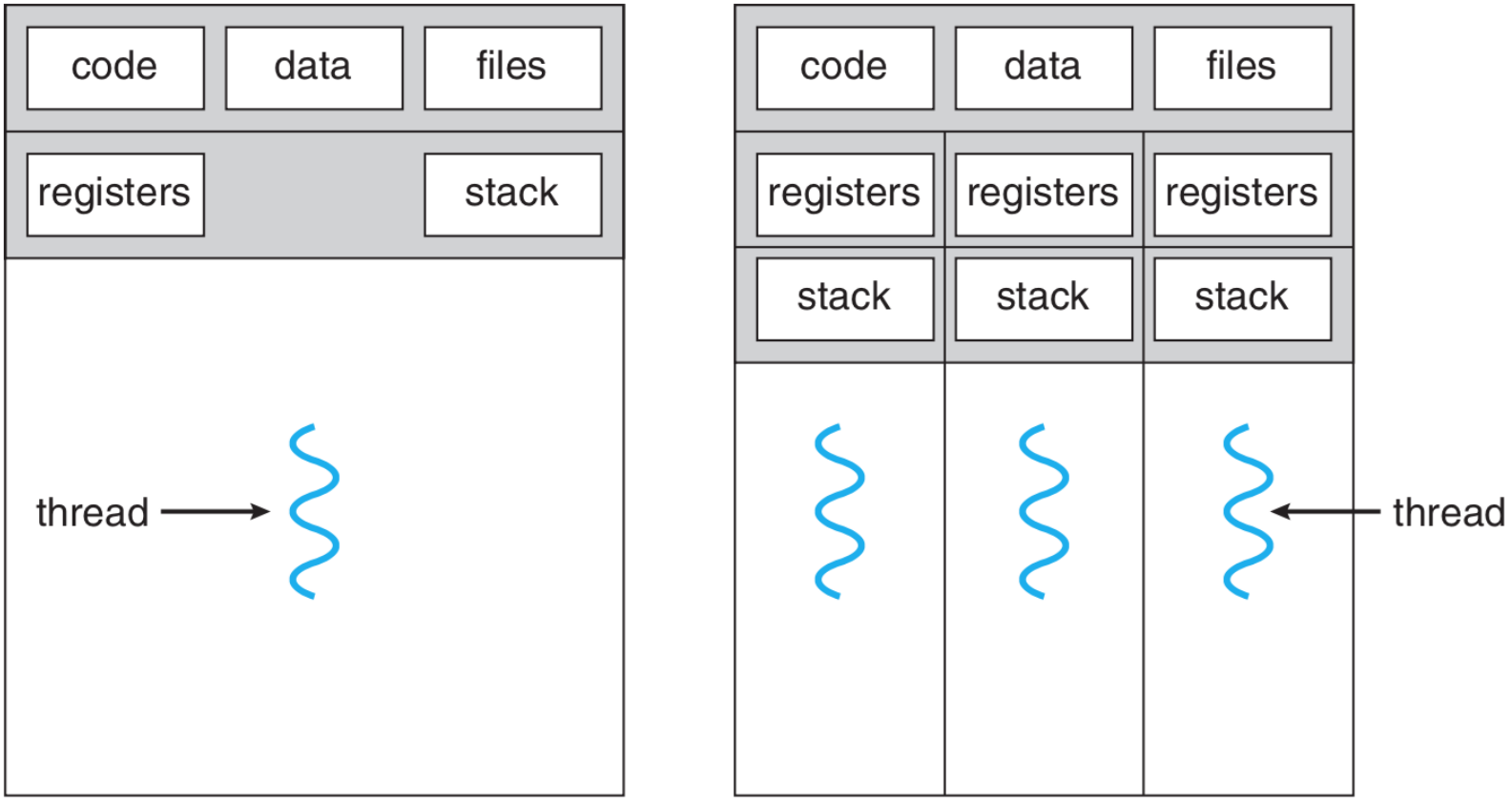


Thread Possessions

In a process that has multiple threads, each thread has its own:

1. Thread execution state.
2. Saved thread context when not running.
3. Execution stack.
4. Local variables.
5. Access to the memory and resources of the process (shared with all threads in that process).

Single vs. Multithreaded



single-threaded process

multithreaded process

Thread Notes

All the threads of a process share the state and resources of the process.

If a thread opens a file, other threads in that process can also access it.

The way programs are written now, few are not multithreaded.

UI Thread

One common way of dividing up the program into threads is to separate the user interface from a time-consuming action.

Consider a file-transfer program.

If the user interface and upload method share a thread, once a file upload has started, the user will not be able to use the UI anymore.

Not even to click the button that cancels the upload!

For some reason, users hate that.

Solving the UI Thread Problem

We have two options for how to alleviate this problem.

Option 1: `fork` a new process to do the upload; or
Option 2: Spawn new thread.

In either case, the newly created entity will handle the upload of the file.

The UI remains responsive, because the UI thread is not waiting for the upload method to complete.

Thread Motivation

Why threads instead of a new process?

Primary motivation is: performance.

1. Creation: 10$\times$ faster.
2. Terminating and cleaning up a thread is faster.
3. Switch time: 20% of process switch time.
4. Shared memory space (no need for IPC).
5. Lets the UI be responsive.

Common Usage of Threads

1. **Foreground and Background Work**
2. **Asynchronous processing**
3. **Speed of Execution**
4. **Modular Structure**

Thread Drawbacks

There is no protection between threads in the same process.

One thread can easily mess with the memory being used by another.

This once again brings us to the subject of co-ordination, which will follow the discussion of threads.

Also, if any thread encounters an error, the whole process might be terminated by the operating system.

Thread States

Each individual thread will have its own state.

Our process model has seven states.
The thread state model is the simpler five-state model.

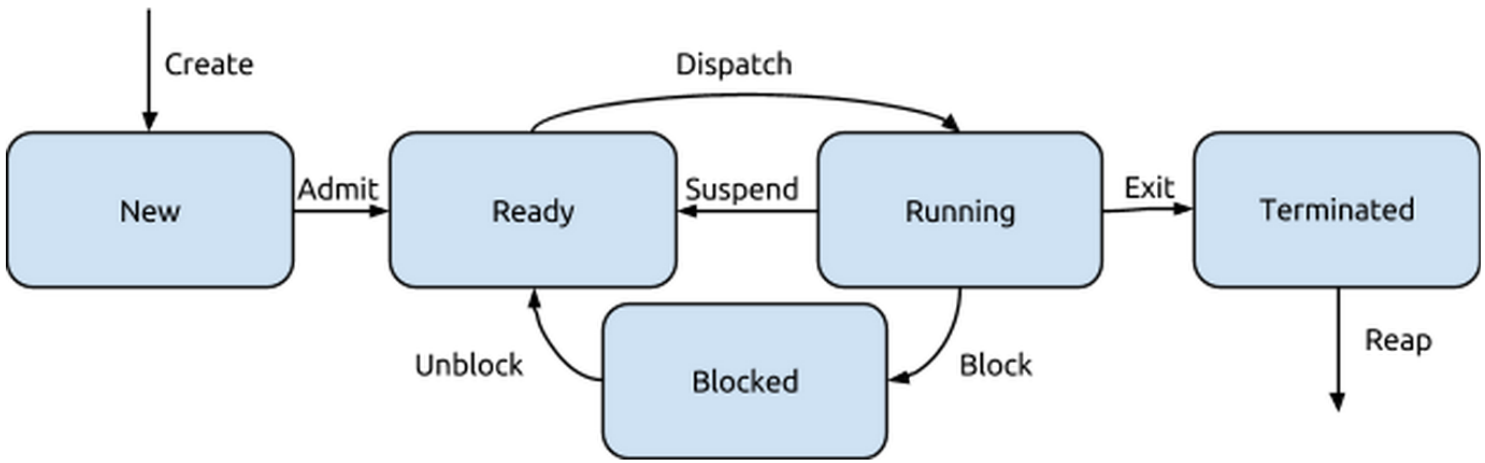
If a process is swapped out of memory, all its threads are swapped out.

When that process returns to memory, all the threads are swapped in.

Therefore we do not need to consider if a thread is in memory/swapped.

Five State Model

Five state model, once again:



Five State Model

The transitions work the same way as the state transitions for a process.

As with a process, a thread in any state can transition to terminated. W

hen a process is terminated, all its threads are terminated
Regardless of what state it is in.

Actually, let's take a minute to look at cancellation...

Thread Cancellation

Thread cancellation is exactly what it sounds like.

A running thread will be terminated before it has finished its work.

Once the user presses the cancel button on the file upload, we want to stop the upload task that was in progress.

The thread that we are going to cancel is called the target.

Thread Cancellation

1. **Asynchronous Cancellation**
2. **Deferred Cancellation**

Deferred Cancellation

For example, in Android, a background task has a function `isCancelled`.

This function returns a boolean value.

If we cancel a task then the `isCancelled` value returns true.
That on its own has no impact on the task.

The task itself is responsible for checking if `isCancelled` is true.

Yes, a thread can just ignore the cancellation.

Deferred Cancellation

If a thread can ignore cancellation, why would we ever choose deferred?

Suppose the thread we are cancelling has some resources.

If the thread is terminated in a disorderly fashion, the operating system may not reclaim all resources from that thread.

Thus a resource may appear to be in use even though it is not.
And not available to threads and processes that may want to use it.

Thread Types

There are two categories of threads in implementation:

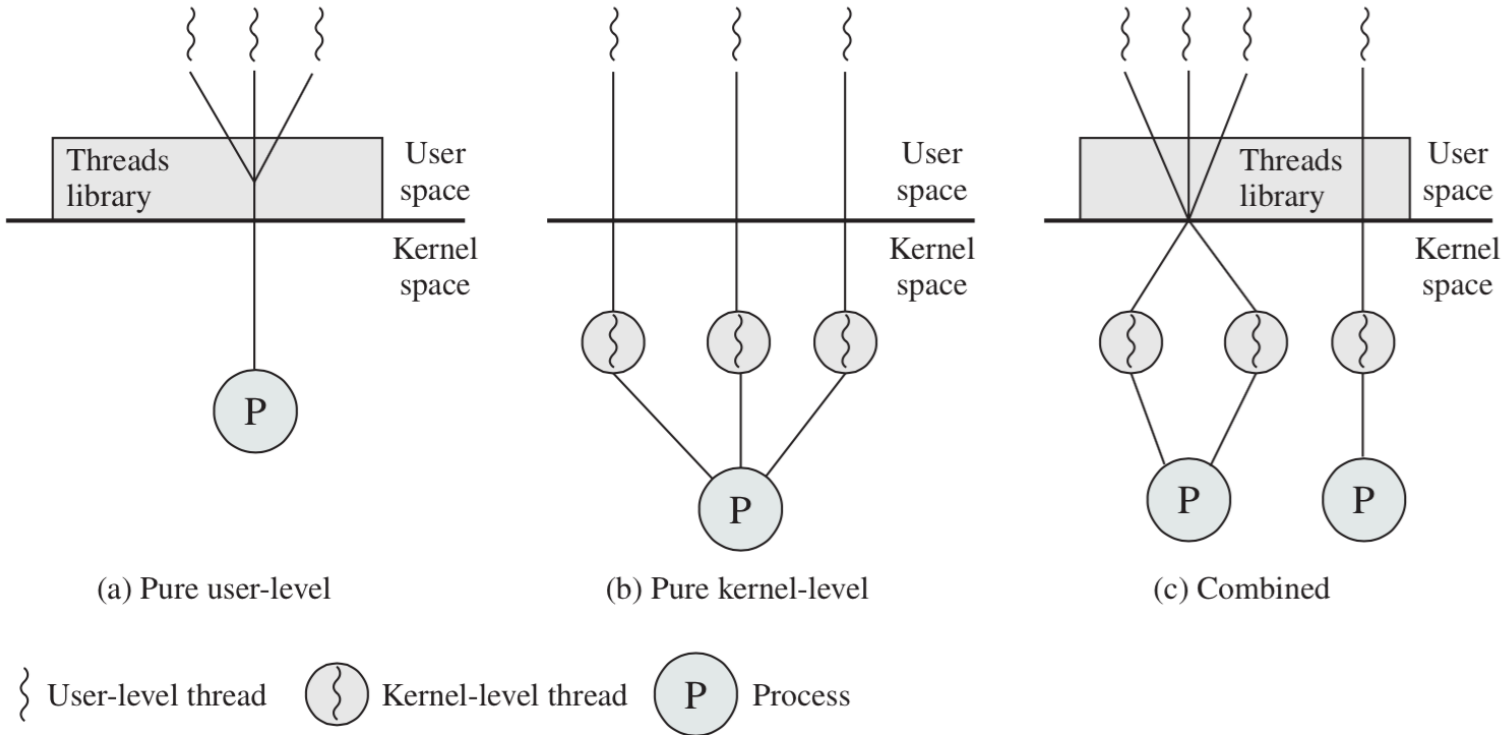
User-level threads (ULTs); and

Kernel-level threads (KLTs – also called a lightweight process).

ULTs run at the user level and KLTs run at the kernel level.

Thread Types

(1) all threads are user level;
(2) all threads are kernel level; and
(3) a combination of both approaches.



Thread Libraries

If the operating system in question does not support threads, we can still have multithreaded programming.

User level threads in some sort of threads library.

The library handles creation, management, and cleanup of threads.

The kernel is unaware of the existence of the user level thread.

It is the responsibility of the library/application to manage threads.

User Level Threads

There are three advantages to using user-level threads [**?**]:

1. Thread switches do not require kernel mode privileges.
2. Scheduling can be something the program decides.
3. Portability.

User Level Threads

The kernel thinks there is only one thread for that process.

If any thread blocks, the whole process is blocked.

Solution: jacketing.
Conversion of a blocking system call into a non-blocking one.

The jacketing routine checks if it's a blocking request.
If so, decides to switch to another thread.

Windows Threads

The kernel level threads approach is taken by Windows.

The kernel is responsible for all thread management.

If one thread in a process is blocked, the others may continue.

Windows kernel routines themselves may be multithreaded.

Relationship Model

To look at this from another angle, we can consider the relationships.

There are three: Many-To-One, One-To-One, and Many-To-Many.