Lecture 10 — Threads

Jeff Zarnett
jzarnett@uwaterloo.ca

Department of Electrical and Computer Engineering
University of Waterloo

August 13, 2025

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 have only one thread;
that is, your program’s code is executed one statement at a time.

Multithreaded Programs

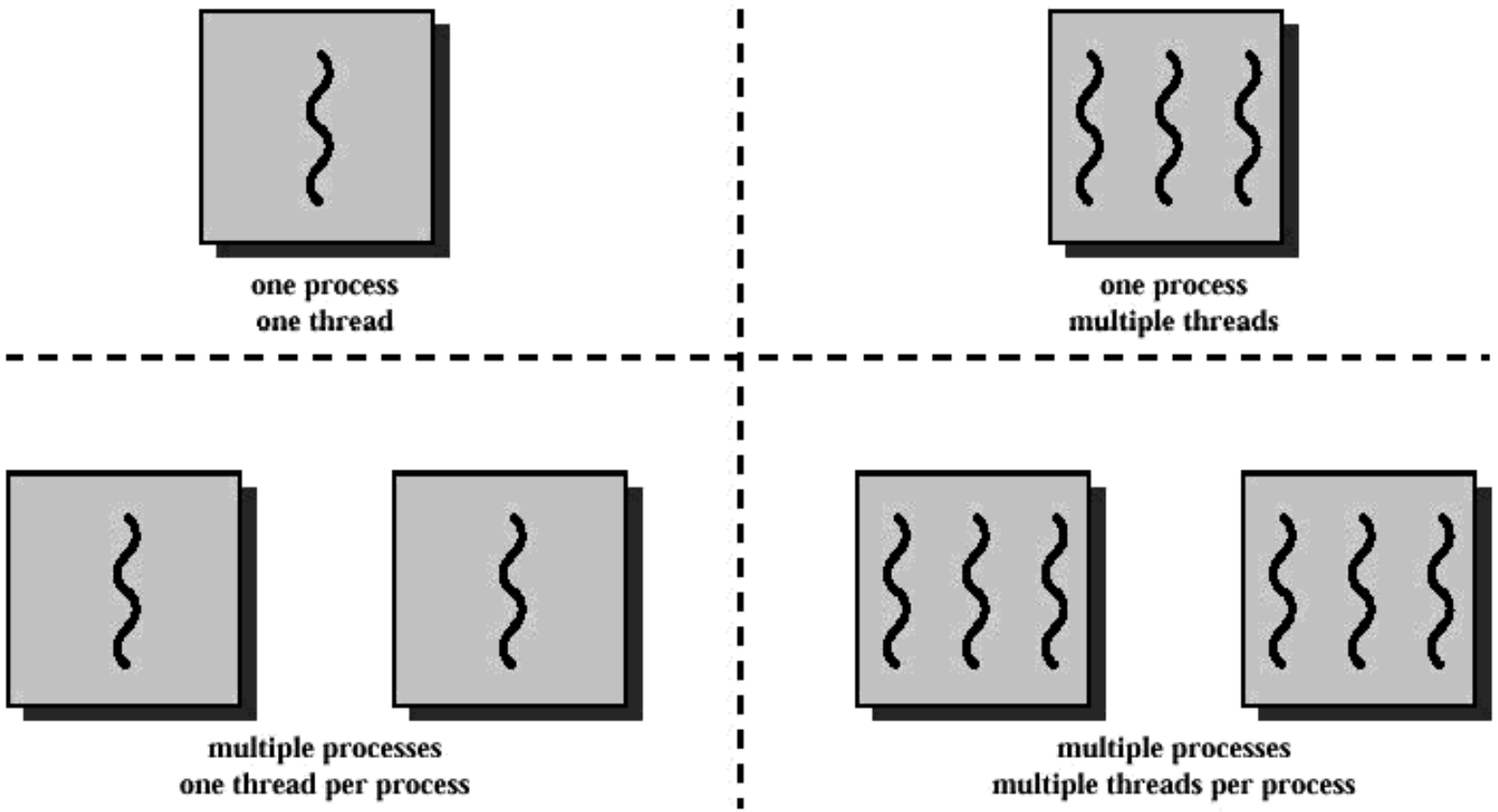
A multithreaded program uses more than one thread, (some of the time).

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

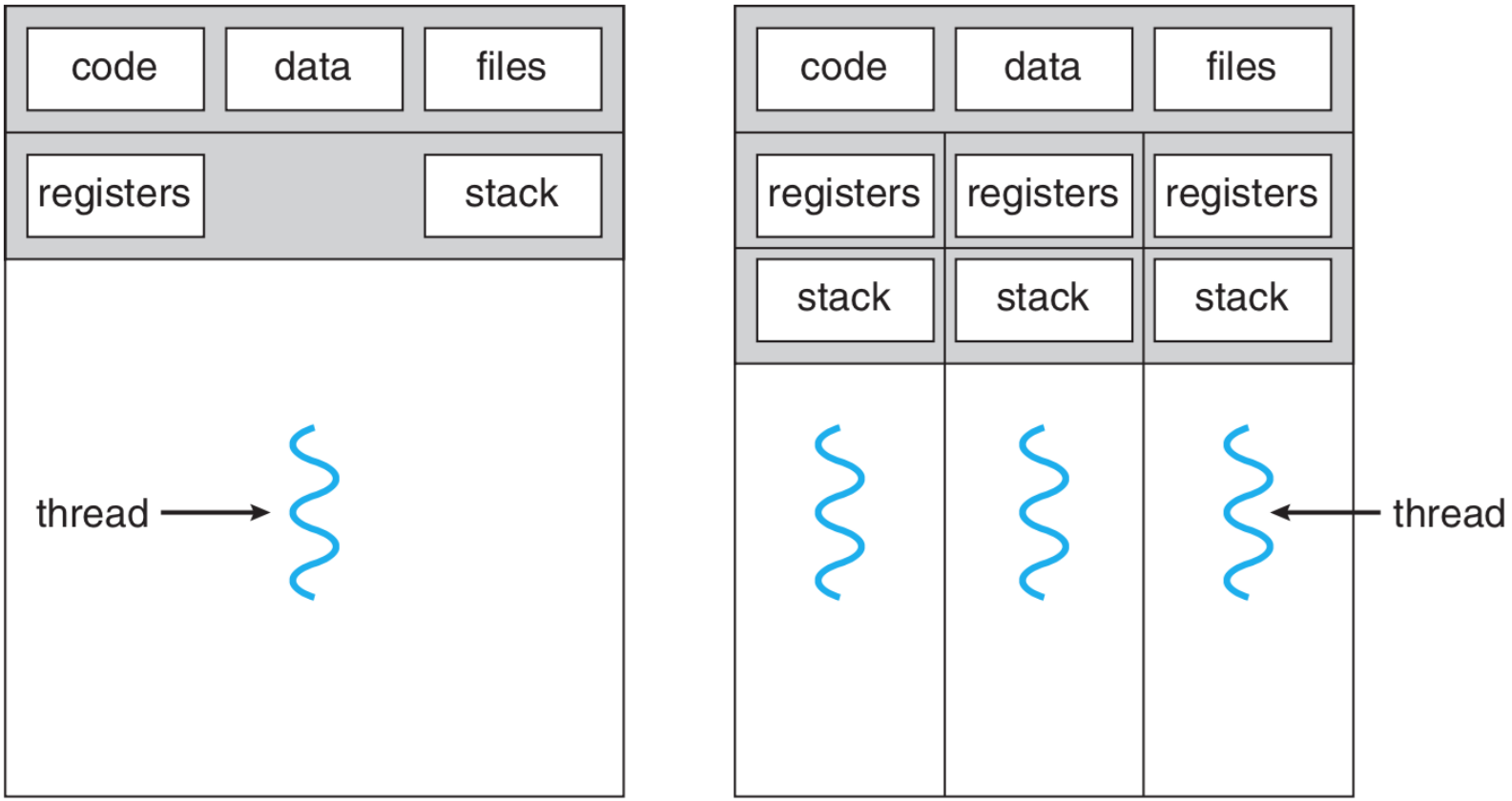


Thread Possessions

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.

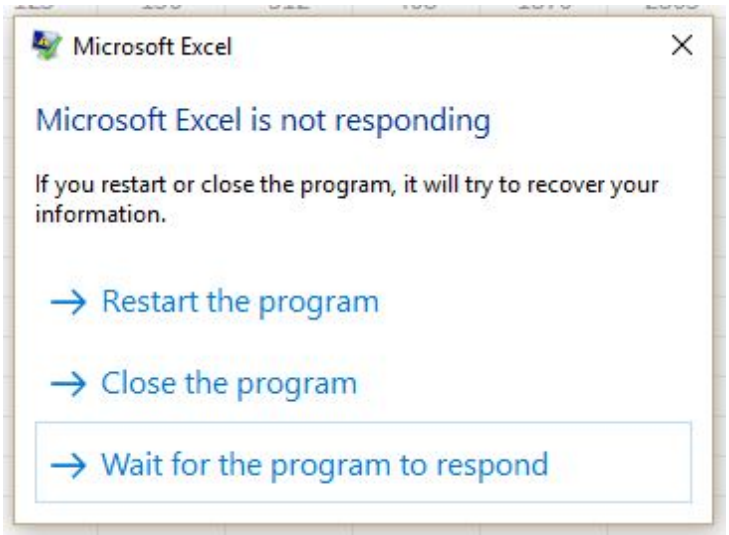
The way programs are written now, few are not multithreaded.

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!

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× 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

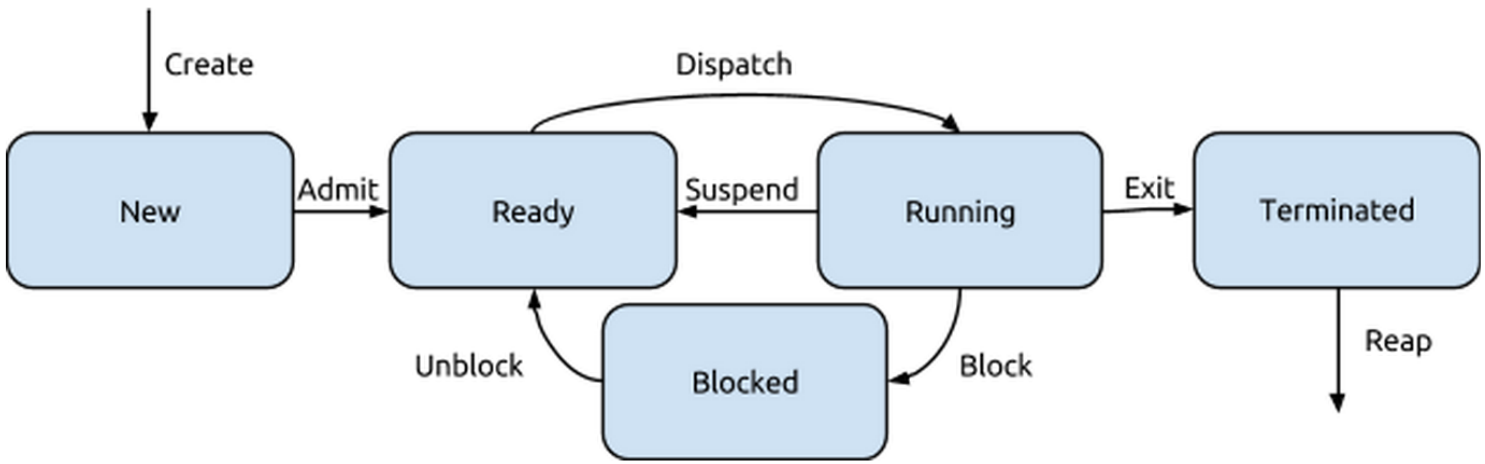
Each individual thread will have its own state.

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

If a process is blocked, we don’t really care why (even if the OS does).

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.

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

The term `pthread` refers to the POSIX standard (also known as the IEEE 1003.1c standard) that defines thread behaviour in UNIX.

- `pthread_create`
- `pthread_exit`
- `pthread_join`
- `pthread_detach`
- `pthread_yield`
- `pthread_attr_init`
- `pthread_attr_destroy`
- `pthread_cancel`
- `pthread_testcancel`

Let's Make a New Thread

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

`thread`: a pointer to a `pthread` identifier and will be assigned a value when the thread is created.

`attr`: attributes; may be NULL for defaults.

`start_routine`: the function the new thread is to run.

`arg`: The argument passed to the routine we want to start.

Start Routine

The type of `start_routine` above is a function signature.

Thus, the `pthread_create` function has to be called with the name of a function matching that signature, such as:

```
void* do_something( void* start_params )
```

After creating a new thread, the process has two threads in it.

Scheduling of the threads is up to the operating system.

There Can Be Only One

C: it is normal to have a single return value from a function, but usually we can have multiple input parameters.

But here we get only one of each?

Define a `struct` for the argument and return type!

```
void* function( void * void_arg ) {
  parameters_t *arguments = (parameters_t*) args;
  /* continue after this */
}
```

We have to cast it inside the thread anyway…

The caller of the `pthread_create` function has to know what kind of argument is expected in the function being called.

Attributes

Attributes can be used to set whether a thread is detached or joinable, scheduling policy, etc.

By default, new threads are usually joinable (that is to say, that some other thread can call `pthread_join` on them).

To prevent a thread from ever being joined, it can be created in the detached state (or use `pthread_detach`)

For virtually all scenarios that we will consider in this course the default values will be fine.

Threadception

There is no mandatory hierarchy of threads.

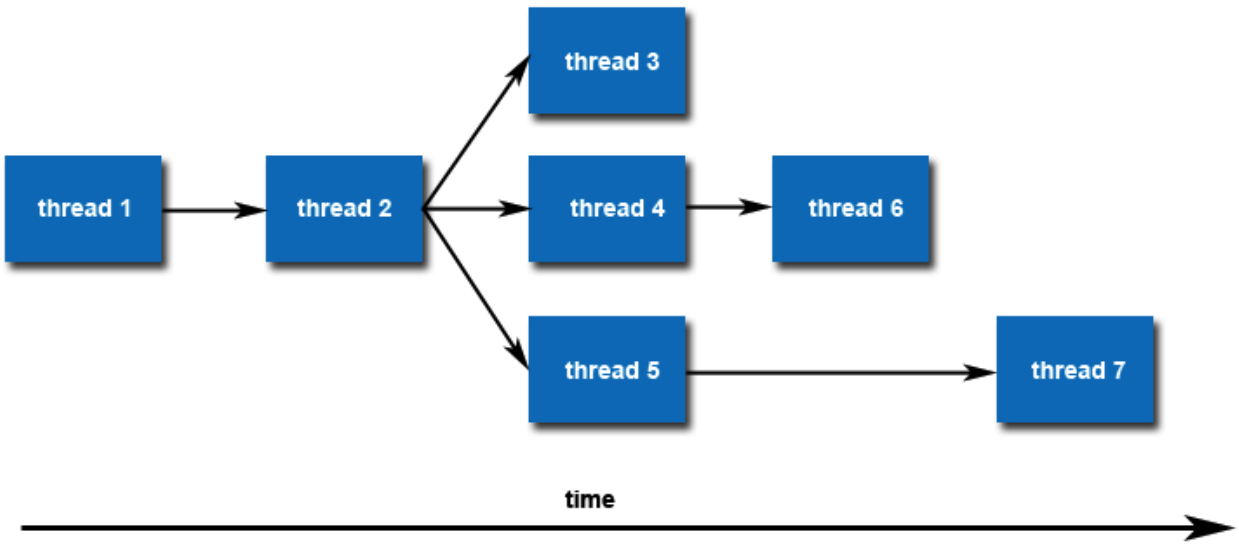


Image Credit: Blaise Barney

New threads can create other threads.

Off We Go

The thread executes its function, until of course it gets to the end.

Usually, it will terminate with `pthread_exit`.

The use of `pthread_exit` is not the only way that a thread may be terminated.

Sometimes we want the thread to persist (hang around), but if we want to get a return value from the thread, then we need it to exit.

Nobody's Listening

If a thread has no return values, it can just `return NULL;`

This will send `NULL` back to the thread that has joined it.

If the function that is called as a task returns normally rather than calling the exit routine, the thread will still be terminated.

Oh… Guess You Didn't Need This After All

Another way a thread might terminate is if the `pthread_cancel` function.
We'll come back to this topic in more detail soon.

A thread may also be terminated indirectly: if the entire process is terminated or if `main` finishes first (without calling `pthread_exit` itself).

End `main` with `pthread_exit` to automatically wait for all spawned threads.

Report, Number One!

Report, Number One!

Like the `wait` system call, the `pthread_join` is how we get a value out of the spawned thread:

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

`thread`: the thread you wish to join.

`retval`: wait... two stars?

Gotta Play the Level Again, Only Got 2 Stars

What we are looking for is a pointer to a void pointer.

That is, we are going to supply a pointer that the join function will update to be pointing to the value returned by that function.

Typically we supply the address of a pointer.

Maybe the example makes it clearer.

Collecting Return Value

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

void * run( void * argument ) {
  char* a = (char*) argument;
  printf("Provided argument is %s!\n", a);
  int * return_val = malloc( sizeof( int ));
  *return_val = 99;
  pthread_exit( return_val );
}

int main( int argc, char** argv ) {
  if (argc != 2) {
      printf("Invalid args.\n");
      return -1;
  }
  pthread_t t;
  void* vr;

  pthread_create( &t, NULL, run, argv[1] );
  pthread_join( t, &vr );
  int* r = (int*) vr;
  printf("The other thread returned %d.\n", *r);
  free( vr );
  pthread_exit( 0 );
}
```