Lecture 2 — Review of Processes and Threads

Jeff Zarnett
jzarnett@uwaterloo.ca

Department of Electrical and Computer Engineering
University of Waterloo

March 26, 2023

Past is Prologue

The Process

A process is a program in execution.

1. The instructions and data.
2. The current state.
3. Any resources that are needed to execute.

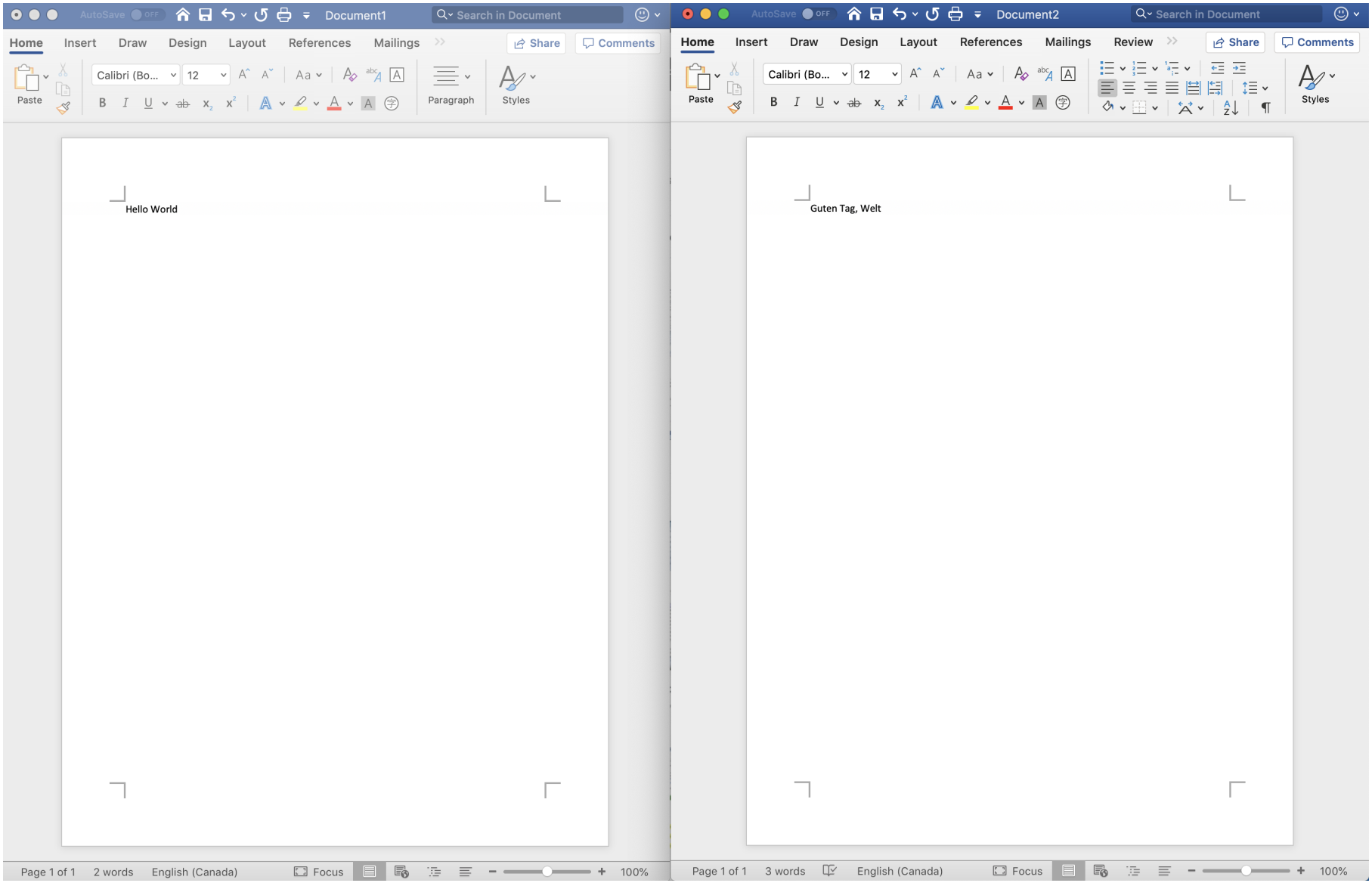
Processes vs. Programs

Note: two instances of the same program running equals two processes.

You may have two windows open for Microsoft Word, and even though they are the same program, they are separate processes.

Similarly, two users who both use Firefox at the same time on a terminal server are interacting with two different processes.

Two Documents, Two Processes



Process Control Block

Data structure for managing processes: Process Control Block (PCB).

It contains everything the OS needs to know about the program.

It is created and updated by the OS for each running process.

It can be thrown away when the program has finished executing and cleaned everything up.

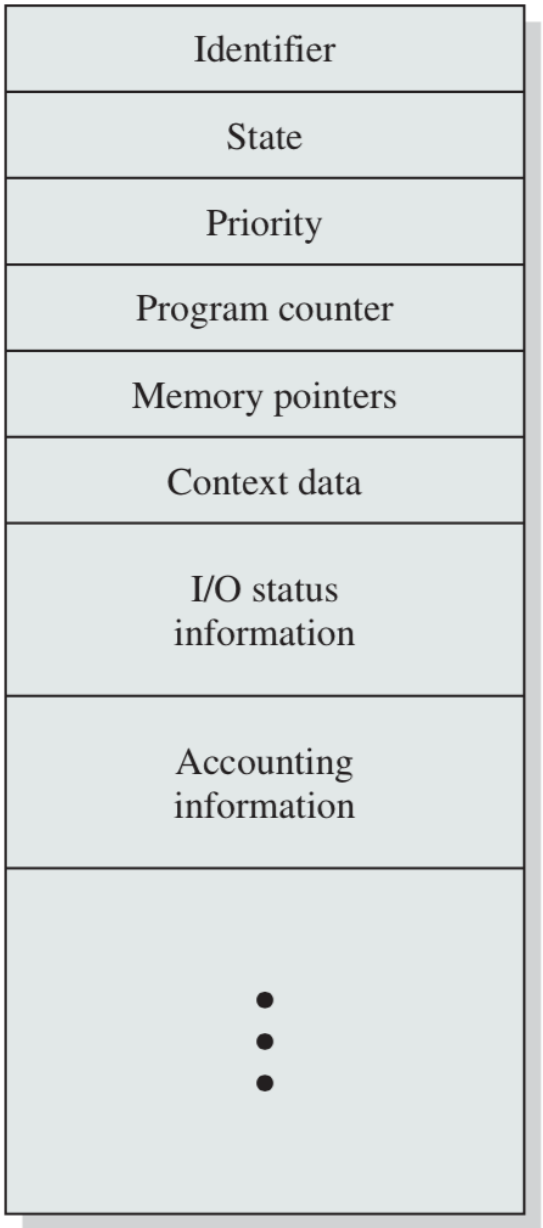
The blocks are held in memory and maintained in some container (e.g., a list) by the kernel.

Process Control Block

The process control block will (usually) have:

- **Identifier.**
- **State.**
- **Priority.**
- **Program Counter*.**
- **Register Data*.**
- **Memory Pointers.**
- **I/O Status Information.**
- **Accounting Information.**

Process Control Block (Simplified)



Process Control Block

This data is kept up to date constantly as the process executes.

The program counter and the register data are asterisked.

When the program is running, these values do not need to be updated.

They are needed when a system call (trap) or process switch occurs.

PCB: Asterisked Fields

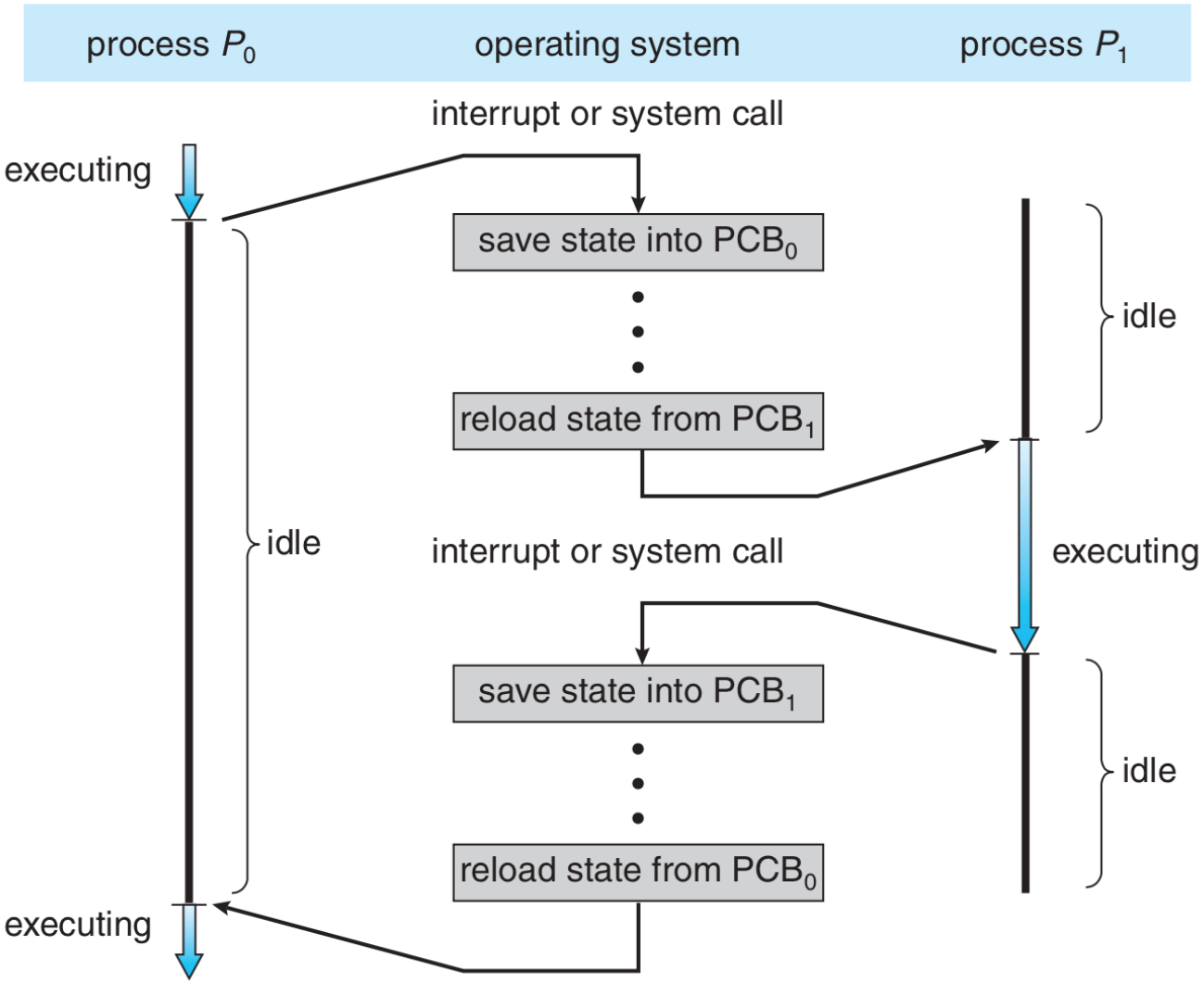
We will need a way to restore the state of the program.

Save the state of the process into the PCB.

1. Save the state into the Program Counter variable.

2. Save the Register variables.

A switch between the execution of process P_0 and process P_1:



The Circle of Life

Unlike energy, processes may be created and destroyed.

Upon creation, the OS will create a new PCB for the process.
- Also initialize the data in that block.

Set: variables to their initial values.
- the initial program state.
- the instruction pointer to the first instruction in `main`

Add the PCB to the set.

After the program is terminated and cleaned up:
- Collect some data (like a summary of accounting information).
- Remove the PCB from its list of active processes and carry on.

Process Creation

Three main events that may lead to the creation of a process:

1. System boot up.
2. User request.
3. One process spawns another.

Process Creation: At Boot Up

When the computer boots up, the OS starts and creates processes.

An embedded system might have all the processes it will ever run.

General-purpose operating systems: allow one (both) of the other ways.

Process Creation: At Boot Up

Some processes will be in the foreground; some in the background.

A user-visible process: log in screen.

Background process: server that shares media on the local network.

UNIX term for a background process is Daemon.

Example: `ssh` (Secure Shell) command to log into a Linux system.

Process Creation: Users

Users are well known for starting up processes whenever they feel like it.

Much to the chagrin of system designers everywhere.

Every time you double-click an icon or enter a command line command (like `ssh` above) that will result in the creation of a process.

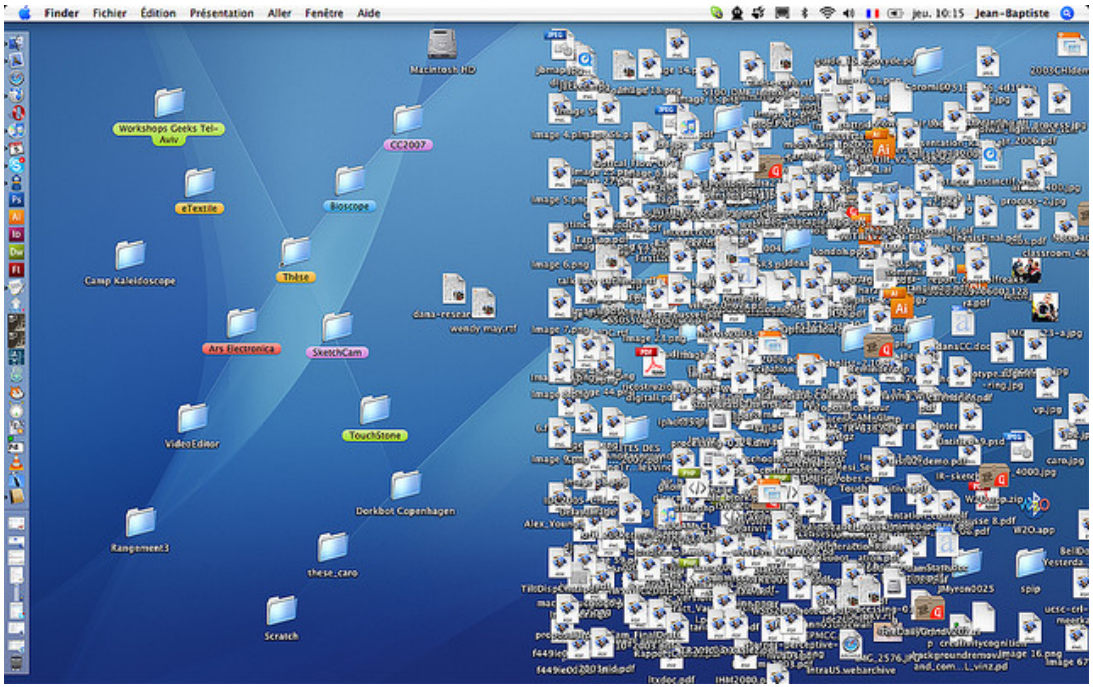


Image Credit: CS Tigers

Process Creation: Spawning

An already-executing process may spawn another.

E-mail with a link? Click it; the e-mail program starts the web browser.

A program may break its work up into different logical parts.
To promote parallelism or fault tolerance.

The spawning process is the parent and the one spawned is the child.

Process Destruction

Eventually, most processes die.

This is sad, but it can happen in one of four ways:

1. Normal exit (voluntary)
2. Error exit (voluntary)
3. Fatal Error (involuntary)
4. Killed by another process (involuntary)

Process Destruction: Normal

Most of the time, the process finishes because they are finished.
Or the user asks them to.

Compiler: when compilation is finished, it terminates normally.

You finish writing a document in a text editor, click the close button.

Both examples of normal, voluntary termination.

Process Destruction: Error

Sometimes there is voluntary exit, but with an error.

Required write access to the temporary directory & no permission.

Compiler: exit with an error if you ask it to compile a non-existent file.

The program has chosen to terminate because of the error.

Process Destruction: Fatality

The third reason for termination is a fatal error.
    Examples: stack overflow, or division by zero.

The OS will detect this error and send it to the program.

Often, this results in involuntary termination of the offending program.

A process may tell the OS it wishes to handle some kinds of errors.

If it can handle it, the process can continue.

Otherwise, unhandled exceptions result in involuntary termination.

Process Destruction: Killed

The last reason for termination: one process might be killed by another.
(Yes, processes can murder one another. Is no-one safe?!).

Typically this is a user request:
a program is stuck or consuming too much CPU...
the user opens task manager (Windows) or `ps` (UNIX)

Programs can, without user intervention, theoretically kill other processes.

Example: a parent process killing a child it believes to be stuck.

Process Destruction: Killed

There are restrictions on killing process.

A user or process must have the rights to execute the victim.

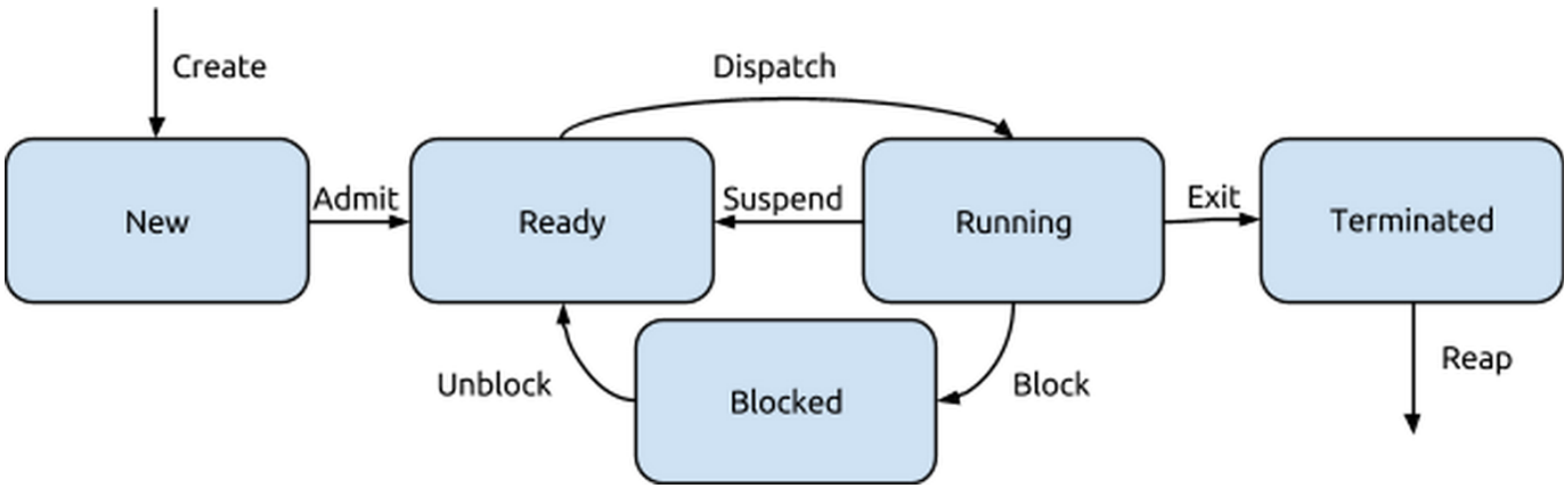
Typically a user may only kill a process he or she has created.

Exception: system administrator.

While killing processes may be fun, it is something that should be reserved for when it is needed.

Maybe when a process is killed, all processes it spawned are killed too.

Five State Model



Five State Model

There are now eight transitions:

- **Create**
- **Admit**
- **Dispatch**
- **Suspend**
- **Exit**
- **Block**
- **Unblock**
- **Reap**

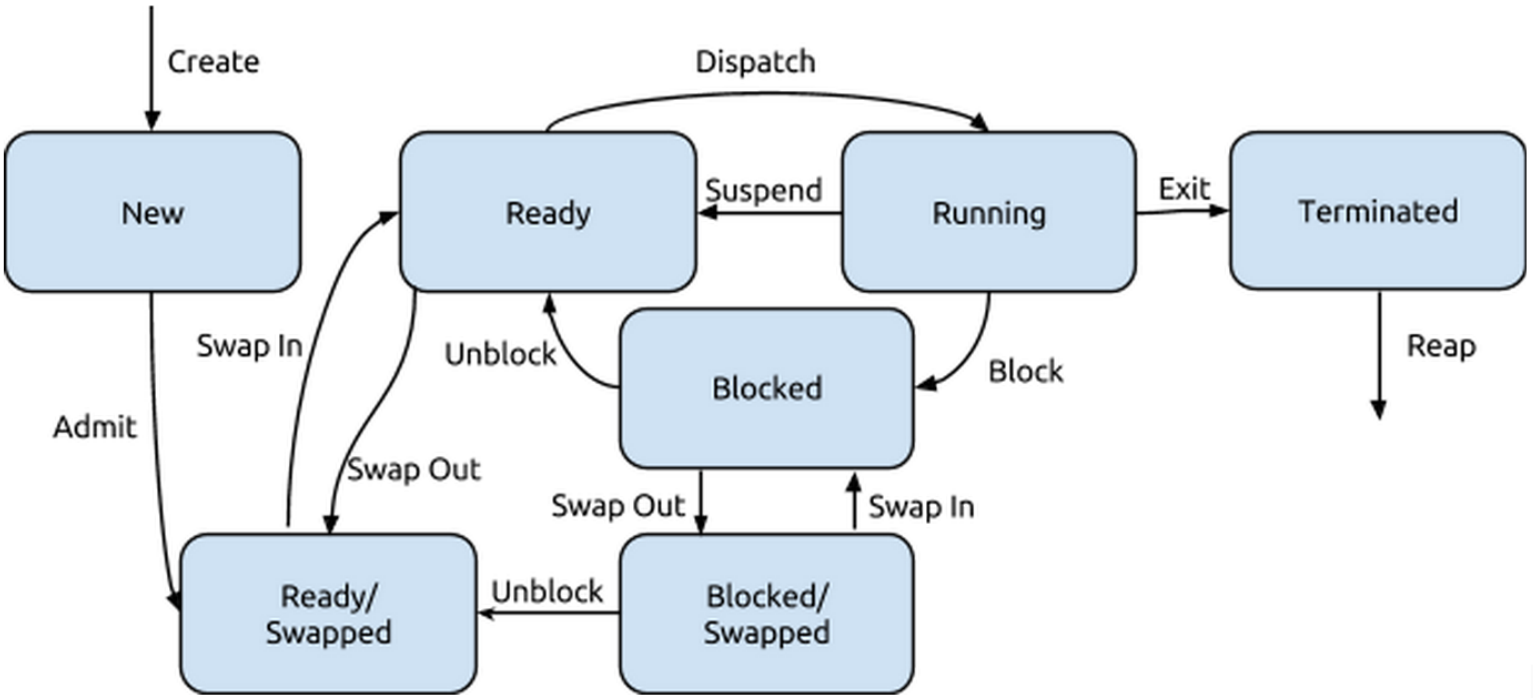
Five State Model

There are two additional exit transitions that are not shown.

A process can go directly from “Ready” or “Blocked” to “Terminated”.

This happens if a process is killed.

Seven State Model



Seven State Model

A variant of the five state model.

The “Admit” transition is modified: by default the new process does not start in main memory.

Two new transitions: “Swap In” and “Swap Out”.

A second “Unblock” transition.

As in the five state model, some additional “Exit” transitions.

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

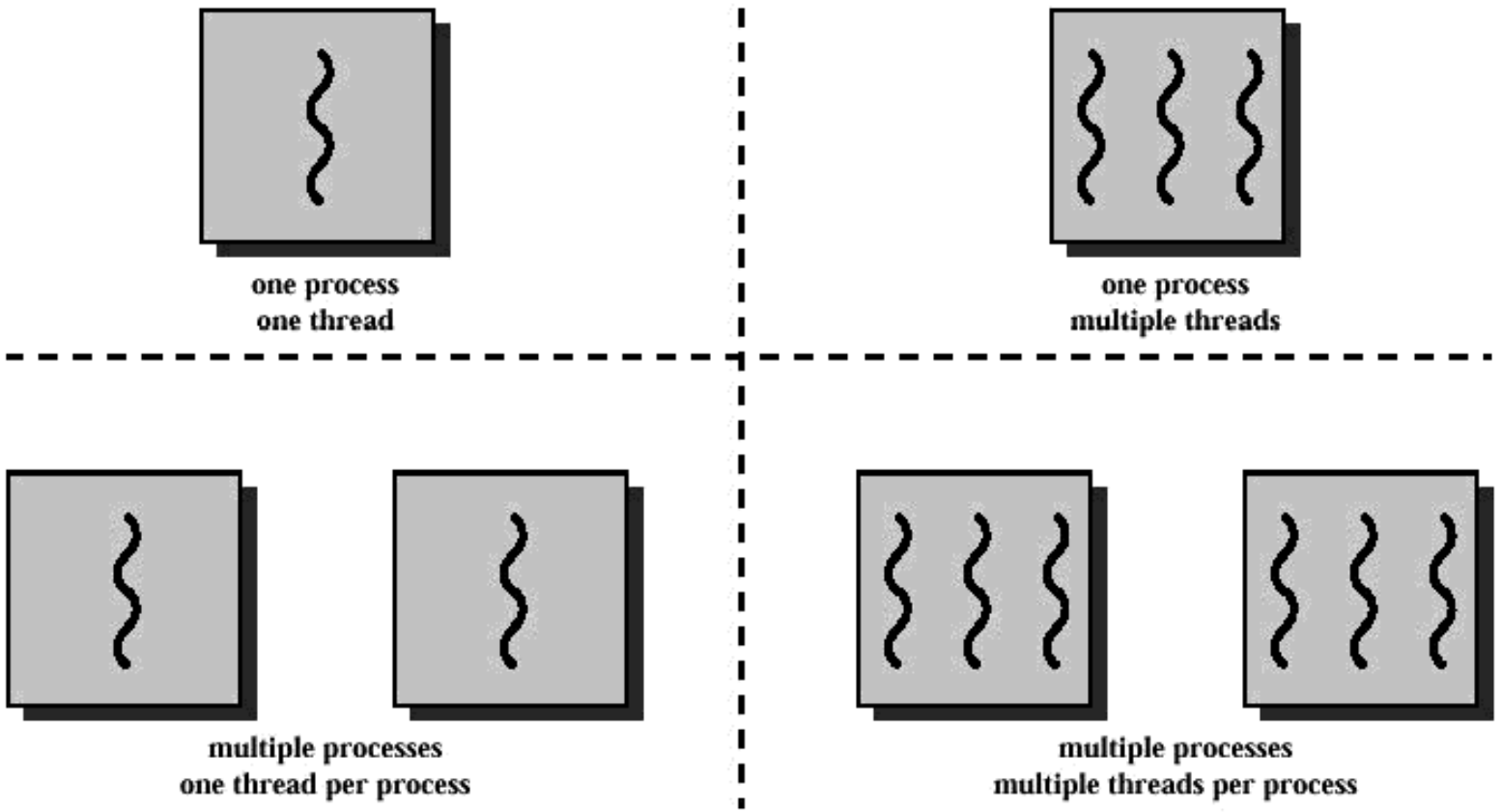
A multithreaded program uses more than one thread, (some of the time).

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

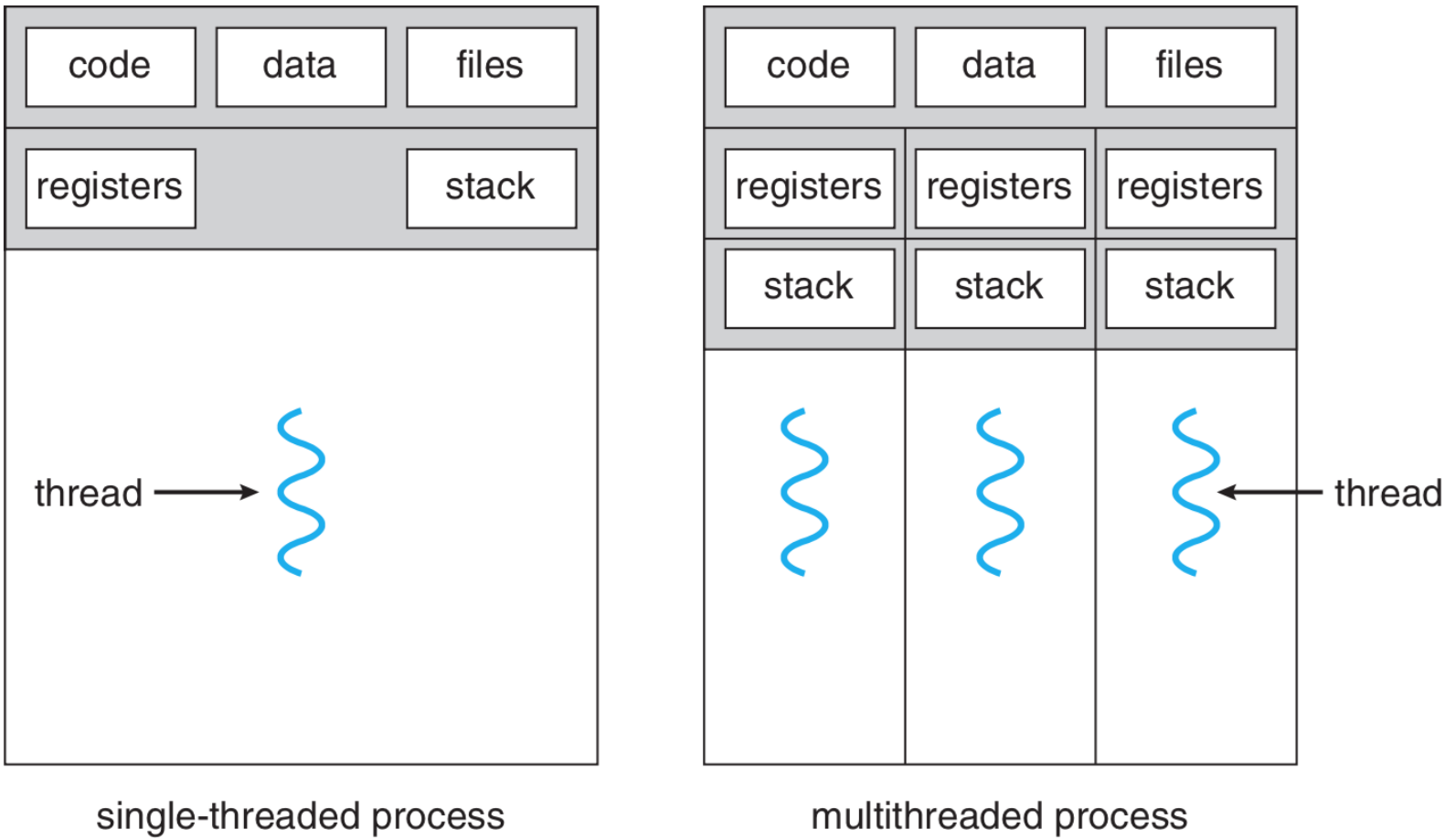


Thread Possessions

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.

Why Threads?

Remember there's a performance-risk tradeoff in threads.

If this didn't seem familiar to you, please go back and review ECE 252!