Lecture 12 — Processes and Threads

Jeff Zarnett
jzarnett@uwaterloo.ca

Department of Electrical and Computer Engineering
University of Waterloo

April 6, 2024

Processes

Early computers did exactly one thing.

Or at least, exactly one thing at a time.

Now the OS supports multiple programs running concurrently.

To manage this complexity, the OS uses the process.

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.

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 up.

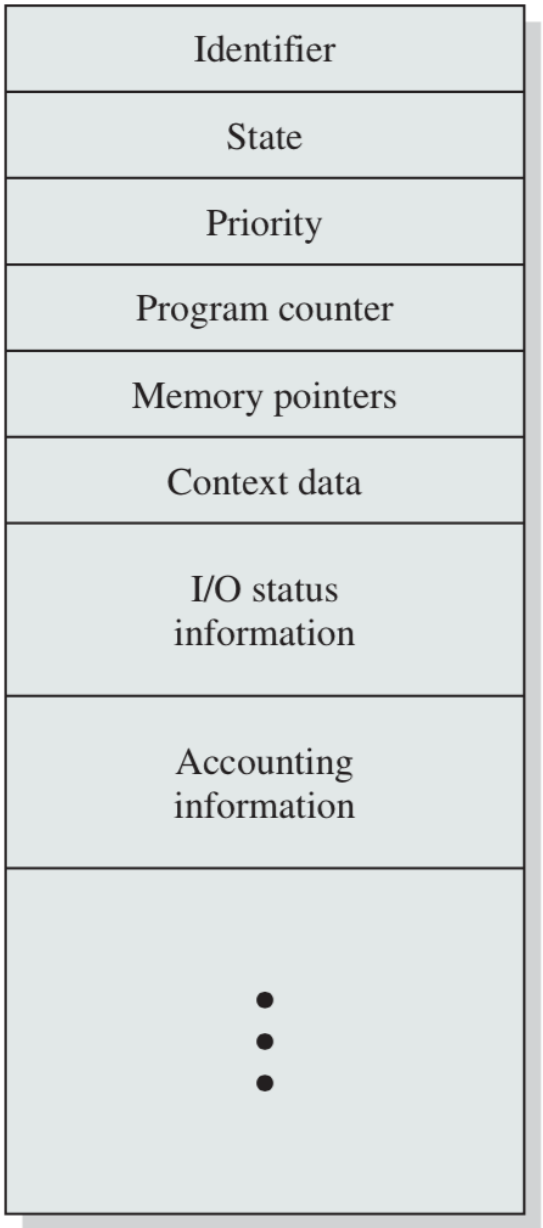
The blocks are held in memory and maintained in some container by the kernel.

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)



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.

The Circle of Life

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.

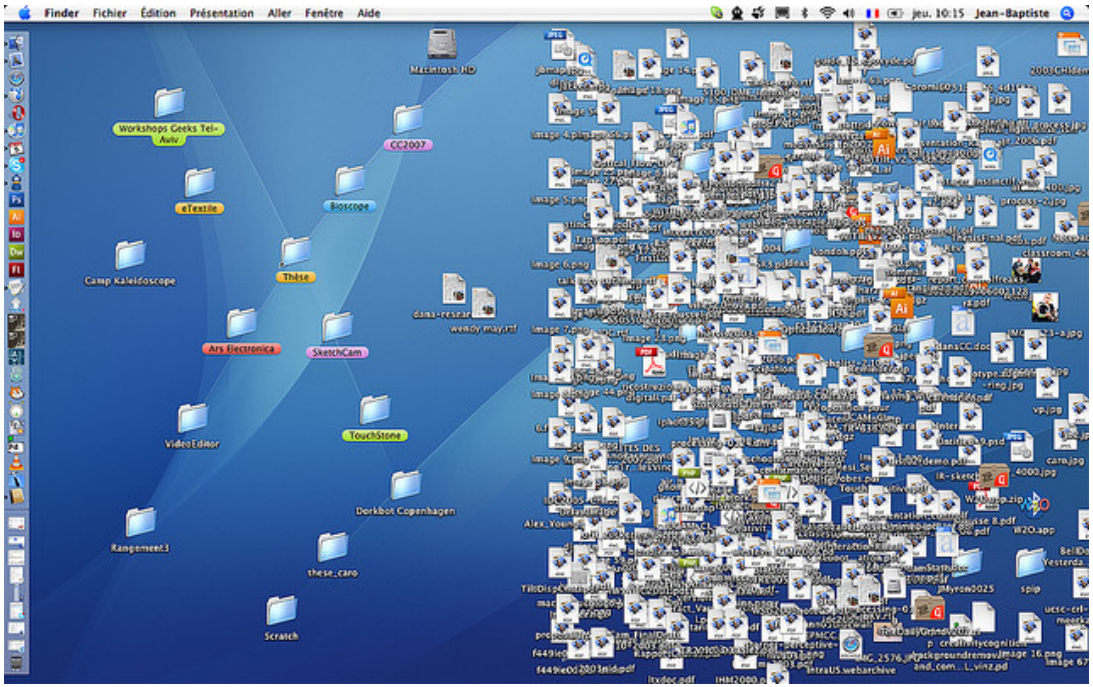


Image Credit: CS Tigers

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, do it only when needed.

Process Family Tree

In UNIX, but not in Windows, the relationship between the parent process and child process(es), if any, is maintained, forming a hierarchy.

A process, unlike most plants and animals, reproduces asexually.

A process has one parent; zero or more children.

A process and all its descendants form a process group.

Certain operations like sending a signal can be sent to a whole group.

Process Family Tree: UNIX

UNIX the first process created is called `init`.

It is the parent of all processes (eventually).
Like the `Object` class in Java is the superclass of all classes.

Thus in UNIX we may represent all processes as a tree structure.

Process Return Codes

When a process terminates, it does so with a return code.
    Just as a function often returns a value.

On the command line or double clicking an icon, return value is ignored.

In UNIX, a parent can get the code that process returns.

Usually, a return value of zero indicates success.
    Other values indicate an error of some sort.

Normally there is some sort of understanding between the parent and child processes about what a particular code means.

The Walking Dead

Aaah! Zombies! Run!

When a child process finishes, until the parent collects the return value, the child continues in a state of “undeath” we call a zombie.

This does not mean that the process then shuffles around the system attempting to eat the brains of other processes.

It just means that the process is dead but not gone.

There is still an entry in the PCB list.
And the process holds on to its allocated resources.

Only after the return value is collected can it be cleaned up.

Getting Rid of Zombies

Usually, a child process's result is eagerly awaited by its parent.

The `wait` call collects the value right away.

This allows the child to be cleaned up (or, more grimly, "reaped").

If there is some delay for some reason, the process is considered a zombie until that value is collected.

Orphans

If a parent dies before the child does, the child is called an orphan.

In UNIX any orphan is automatically adopted by the `init` process.
…making sure all processes have a good home.

By default, `init` will just `wait` on all its child processes
And do nothing with the return values.

A program can be intentionally orphaned: to run in the background.

This would be cruel, except that processes, as far as anyone knows, do not have feelings.

Five State Model

As you might imagine, at any given time, a process is running or not running.

The first two states of the model are therefore “Running” and “Ready”.

A program that requests a resource like I/O or memory may not get it right away.
This gives us the “Blocked” state.

Five State Model

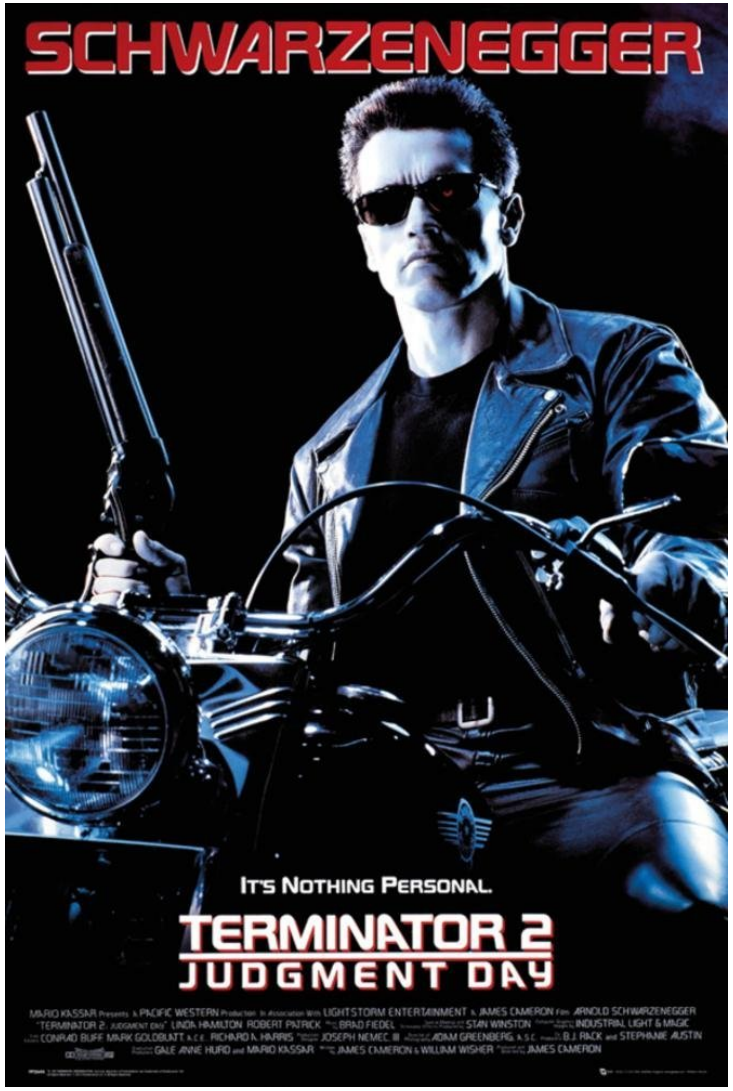
But we did not cover things like zombies.

Life pro tip: the character who doubts that zombies are real dies first.

A UNIX process may be finished but its value yet uncollected.

It is not ready to run, but not waiting for a resource either.

Come with me if you want to live



New state: Terminated.

Five State Model

That accounts for four states; what about the fifth?

The fifth is the “New” state: just created.

If the user creates a process, the OS has significant work to do.

- Define an identifier.
- Instantiate the PCB.
- Put the process in the New state.

The process is defined, but the OS has not started it yet.

Five State Model

Why bother with the “New” state?

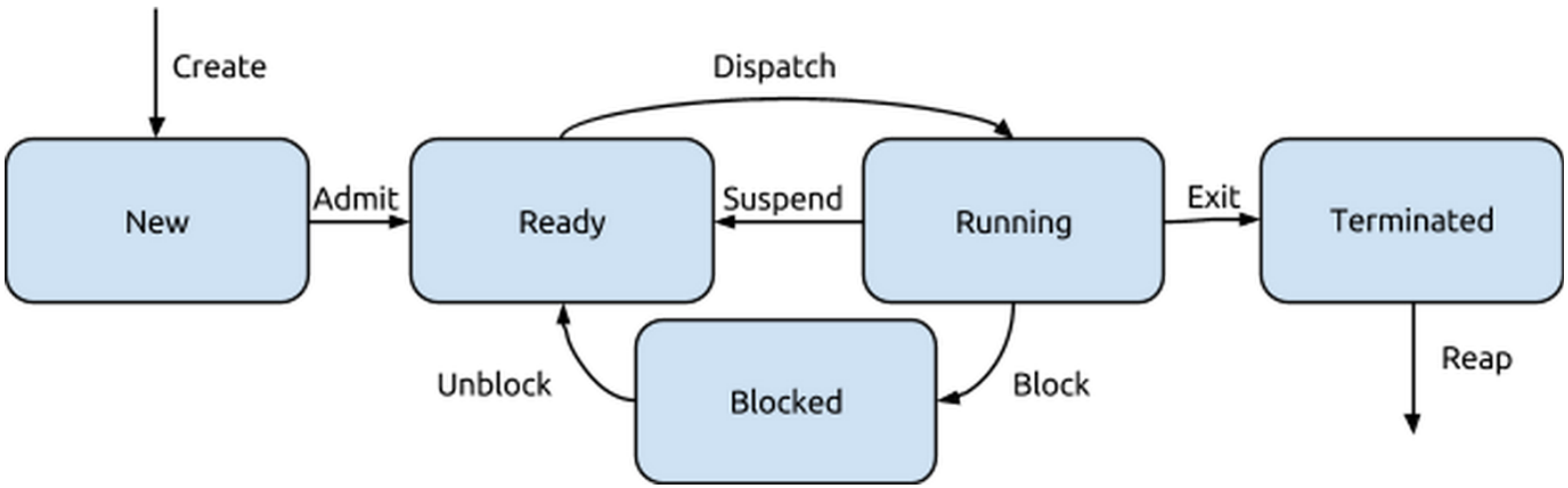
The system may limit the number of concurrent processes.

New processes are typically on disk and not in memory.

Thus, with the two new states added, the five states are:

1. **Running**
2. **Ready**
3. **Blocked**
4. **New**
5. **Terminated**

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.

Threads?

Although the concept of the process is important, most operating systems these days think about things at the level of the thread.

Almost everything we just learned about process management can apply to thread management!

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

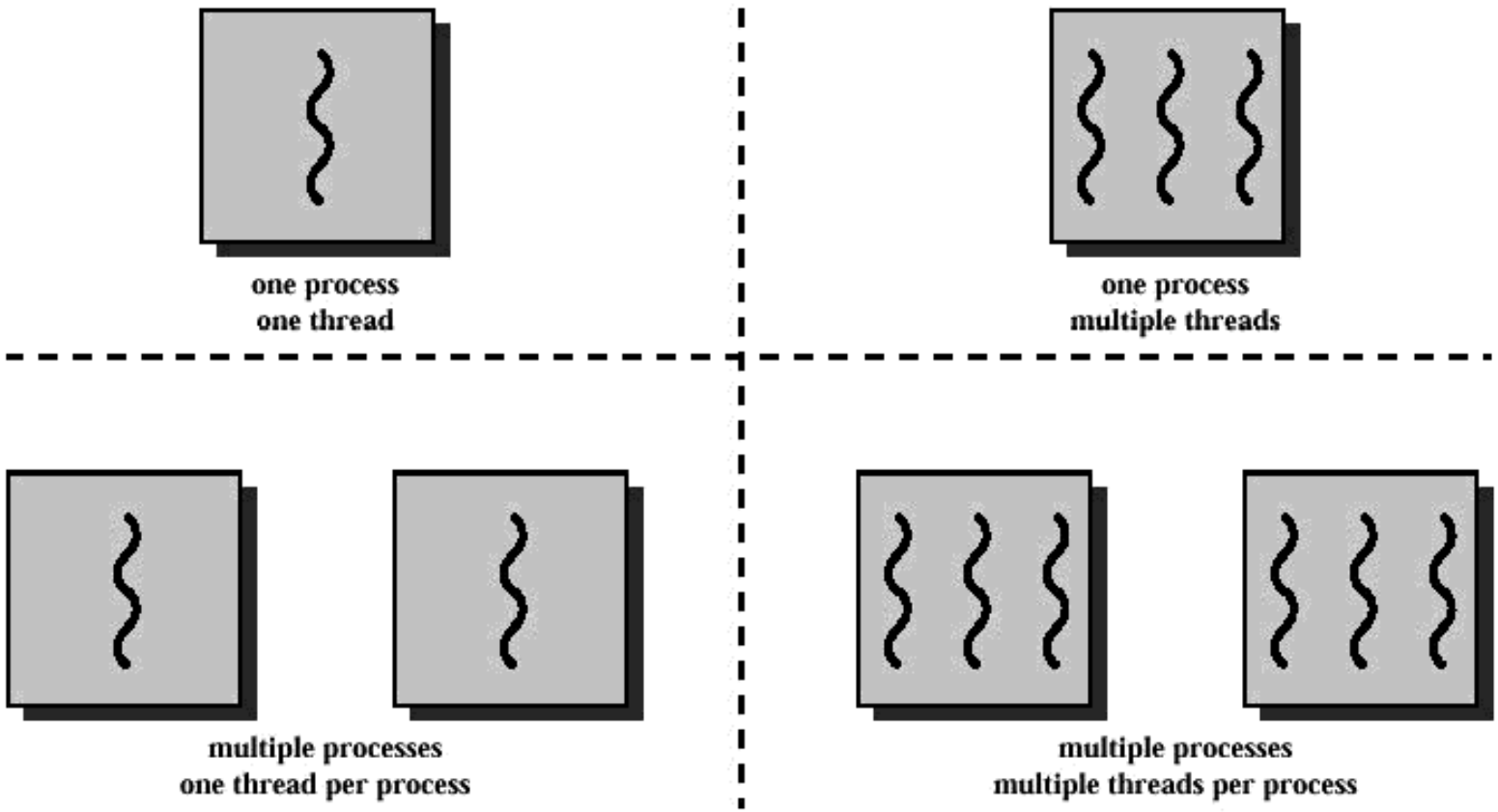
A multithreaded program uses more than one thread, (some of the time).

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

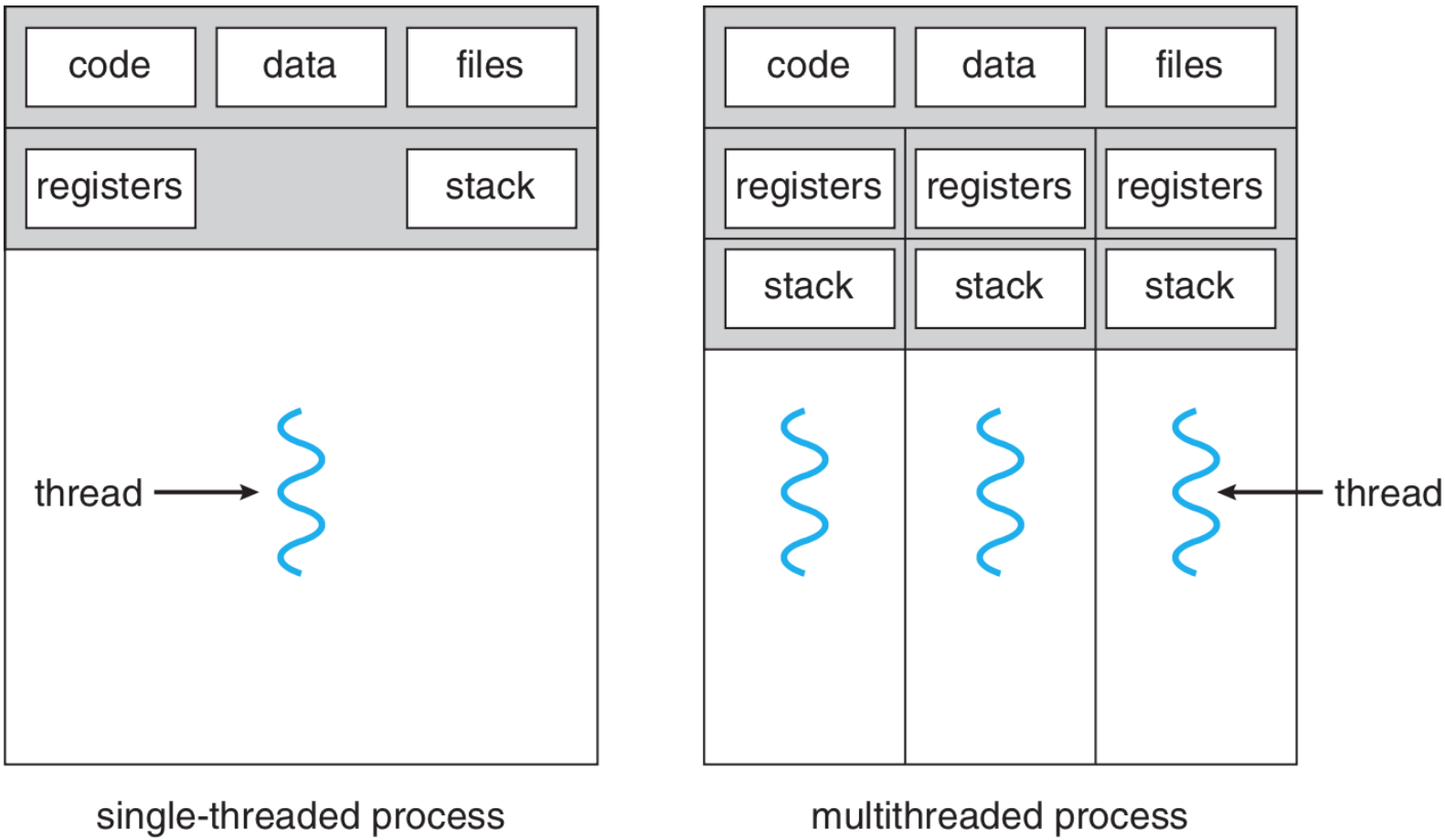


Thread Possessions

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.

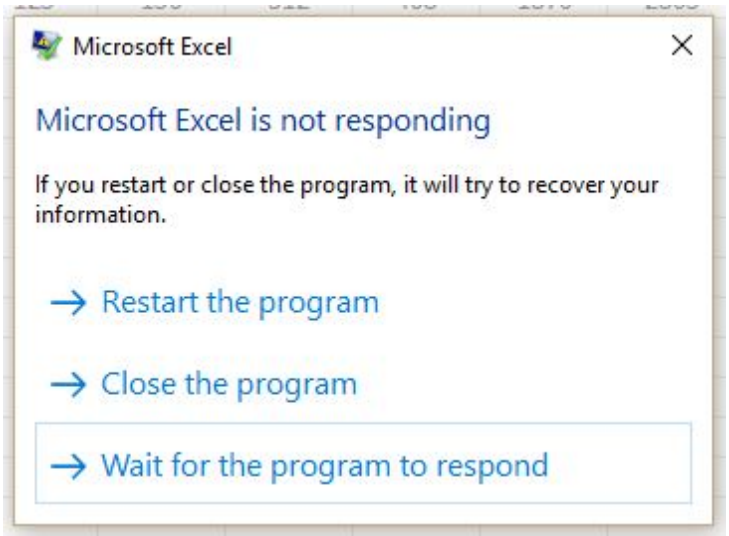
If a thread opens a file, other threads in that process can also access it.

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.

File transfer app: 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$\times$ 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.

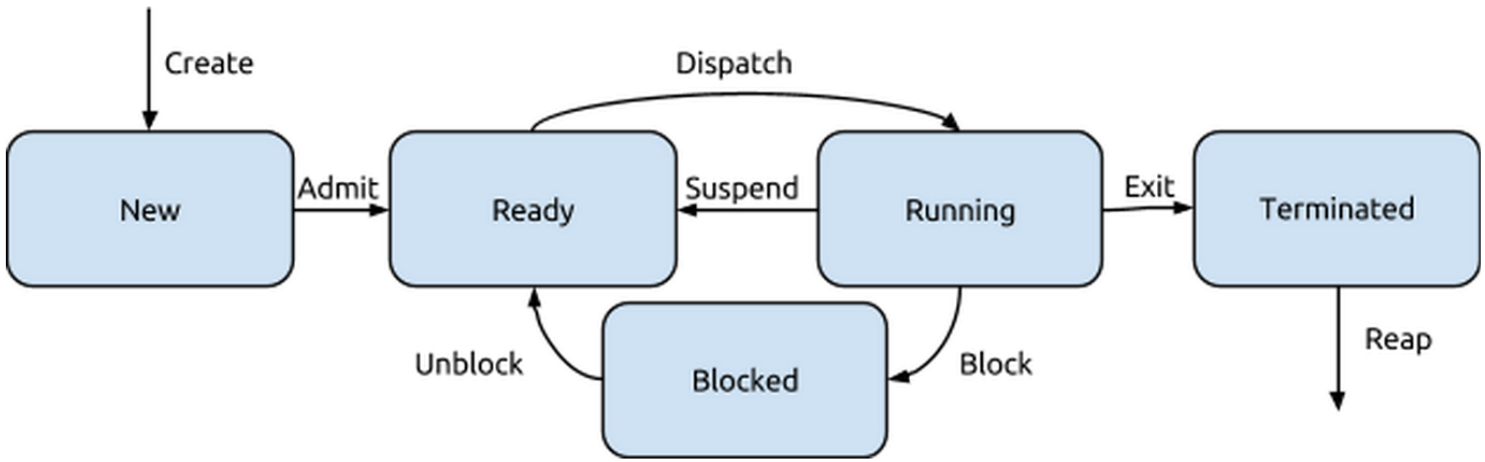
This once again brings us to the subject of co-ordination (for later).

If 1 thread encounters an error, the whole process will be terminated by the OS.

Thread States

Each individual thread will have its own state.

Still using the five state model:



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.

Multiprocessing

Not that long ago, a typical computer had one processor with one core.

It could accordingly do exactly one thing at a time.

1 processor: 1 general purpose processor that executes user processes.

There may be special-purpose processors in the system (RAID controller).

Only one general purpose processor so we call it a uniprocessor system.

Multiprocessing

Now, desktops, laptops, and even cell phones are using multi-core processors.

A quad-core processor may be executing four different instructions from four different threads at the same time.

In theory, multiple processors may mean that we can get more work done in the same amount of (wall clock) time, but this is not a guarantee.

Terminology Note

Terminology note: we often refer to a logical processing unit as a core.

CPU may refer to a physical chip that contains 1+ logical processing units.

As far as the operating system is concerned, it does not much matter if a system has four cores in four physical chips or four cores in one chip.

Either way, there are four units that can execute instructions.

Execution

1 process, 1 thread: it does not matter how many cores are available.
At most one core will be used to execute this task.

If there are multiple processes, each process can execute on a different core.

But what if there are more processes and threads than available cores?

We can hope that the processes get blocked frequently enough and long enough?

Chest Day Best Day

The line at my gym is actually longer 💪

"Can I work in with you?"

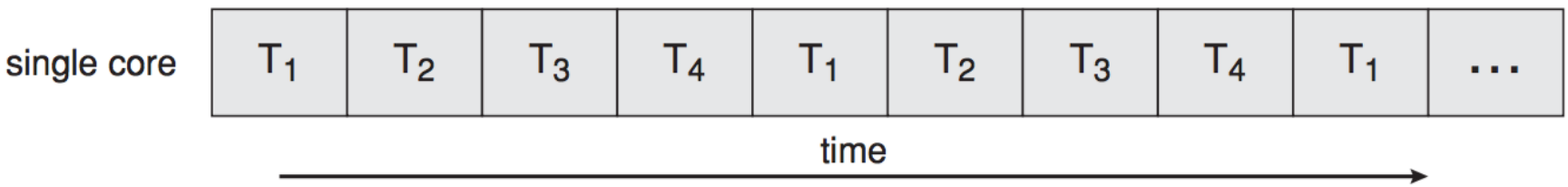
Execution

Switch between the different tasks via a procedure we call time slicing.

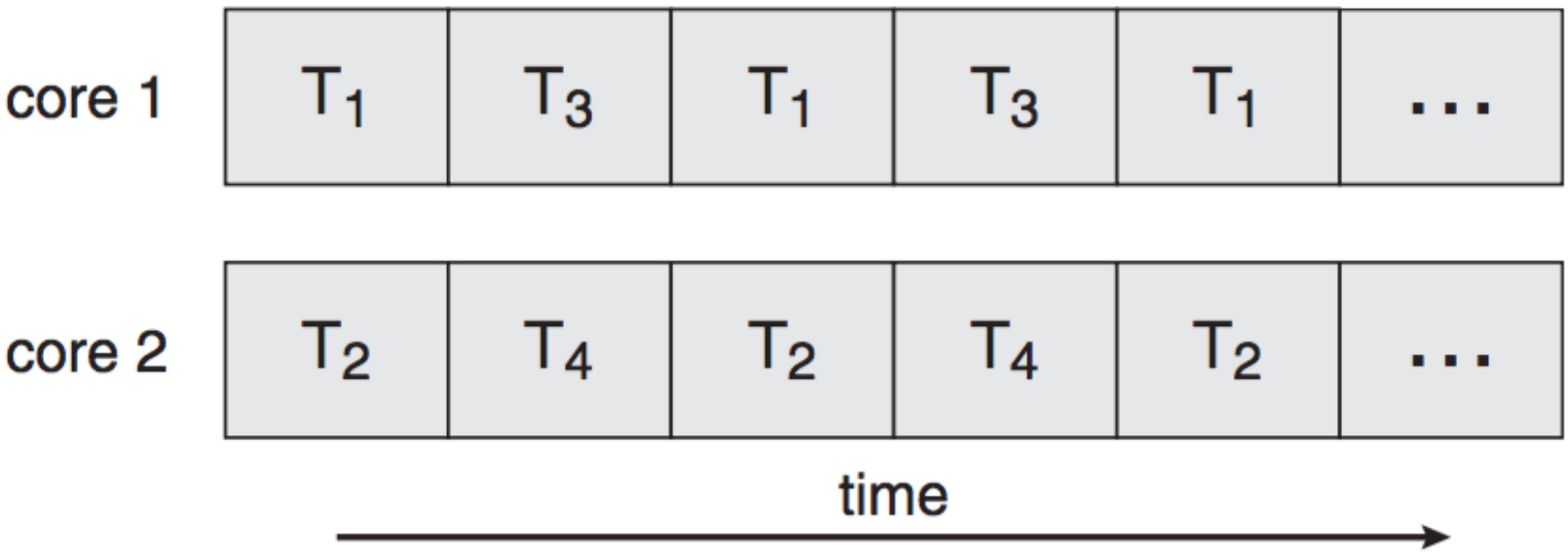
So thread 1 would execute for a designated period, such as 20 ms, then thread 2 for 20 ms, then thread 3 for 20 ms, then back to thread 1 for 20 ms.

To the user, it seems like threads 1, 2, and 3 are being executed in parallel.
20 ms is fast enough that the user does not notice the difference.

Single Core Execution



Time slicing will still occur, if necessary:



Multiple threads at the same time = tasks completed faster?

Parallelism and Speedup

Depends on the nature of the task!

Fully parallelized: $2 \times$ Threads = $2 \times$ Speed

Partially parallelized: $2 \times$ Threads = $(1 < n < 2) \times$ Speed

Cannot be parallelized: $2 \times$ Threads = $1 \times$ Speed

Heard, Chef!

Speedup Example

Suppose: a task that can be executed in 5 s, containing a parallelizable loop.

Initialization and recombination code in this routine requires 400 ms.

So with one processor executing, it would take about 4.6 s to execute the loop.

Split it up and execute on two processors: about 2.3 s to execute the loop.

Add to that the setup and cleanup time of 0.4 s and we get a total time of 2.7 s.

Completing the task in 2.7 s rather than 5 s represents a speedup of about 46%.

Amdahl's Law

Gene Amdahl came up with a formula for the general case of how much faster a task can be completed based on how many processors we have available.

Let us define S as the portion of the application that must be performed serially and N as the number of processing cores available.

Amdahl's Law:

$$\text{speedup} \leq \frac{1}{S + \frac{1-S}{N}}$$

Amdahl's Law

Take the limit as $N \rightarrow$ infinity and you will find the speedup converges to $\frac{1}{S}$.

The limiting factor on how much additional processors help is the size of S.

Matches our intuition of how it should work.

Amdahl's Law on the 5 s Task

Applying this formula to the example from earlier:

Processors	Run Time (s)
1	5
2	2.7
4	1.55
8	0.975
16	0.6875
32	0.54375
64	0.471875
128	0.4359375

Observations on the 5 s Task

1. Diminishing returns as we add more processors.

2. Converges on 0.4 s.

The most we could speed up this code is by a factor of $\frac{5}{0.4} \approx 12.5$.

But that would require infinite processors (and therefore infinite money).

Merge Sort Example

Recall from data structures and algorithms the concept of merge sort.

This is a divide-and-conquer algorithm like binary search.

Split the array of values up into smaller pieces, sort those, and then merge the smaller pieces together to have sorted data.

Merge Sort Example