Operating Systems

Beep Boop

A Little History

Early computers were initially single user systems and evolved through

- hard-wired programs
- punched cards
- magnetic tape

and other methods of feeding a new program into the computer system.

A Little History (2)

Using a computer meant signing up for a time slot: Actually walking up to the computer center at the appointed time.

- Each user was allocated a minimum 15-minute slot
- They usually spent 10 minutes just setting up the equipment to do their computation ...
- You may have had only 5 minutes or less of actual computation — wasting two thirds of the time slot!

Historical Costs

The economic cost of all the setup time was staggering.

- $300 per hour for computer time on an IBM 701 in 1954 dollars
- that's $3,613/hr in 2026 dollars!

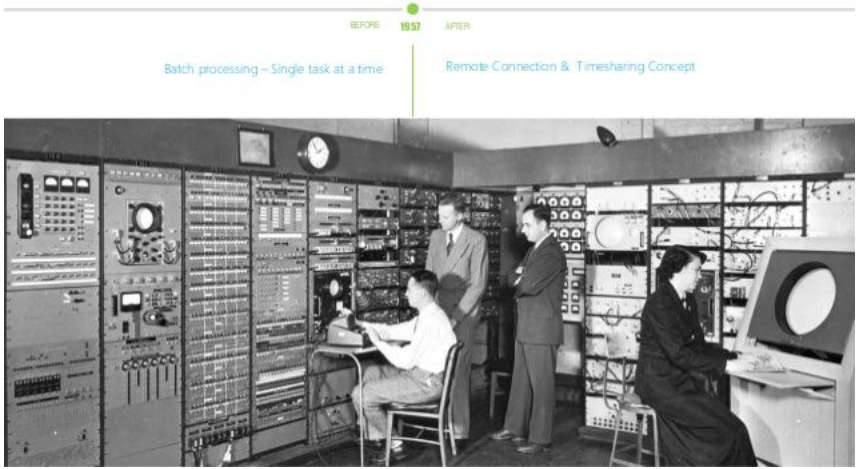
The first **operating systems** were designed to fix this problem.

- Can we automate loading the next task when the current ends?

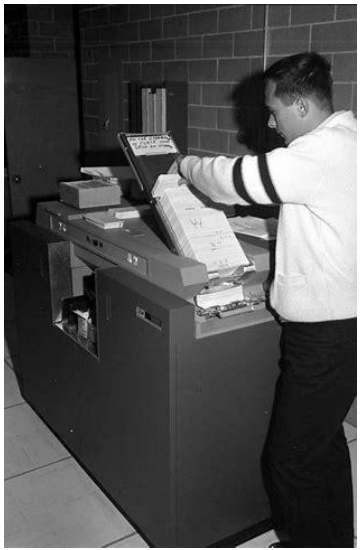
This evolved into a *revolutionary* OS mechanism called **Batch Processing.**

Batch Processing (1956)

Still not multitasking, but allowed people to queue up jobs without reconfiguring the whole machine for each







Batch Processing: Not quite OS, but…

It introduces a **higher authority** that manages *what task gets access to the hardware at what time!*

In this case: program 1 gets unrestricted access to all hardware, **then** program 2 gets unrestricted access to all hardware, etc…

Problem: This is still not the most efficient use of resources. *Programs don't need ALL hardware at ALL times.*

Multiprogramming and Timesharing (1960s)

Multiprogramming

- Multiple programs loaded into memory at the same time.
- CPU switches between programs when one has to wait; usually on disk, memory, or user input.
- Main goal is to **reduce CPU idle time.**

Timesharing

- Allowed multiple users to access the same “mainframe” computer simultaneously from **remote** terminals.
- CPU **rapidly switches** between users via **time slices**.

Multiprogramming and Timesharing Continued

Problem: Each user is still locked into one task at a time.

Why not just make a bunch of users on the old timesharing system?

- For a while, that's exactly what we did.
- It was annoying.
- Still not efficient: time slices were **equally allocated.**

True Multi-tasking (1971, Unix)

Introducing **Preemptive Multitasking**

- OS handles multiple user programs at the same time.
- Achieved using rapid **context switching.**
- The user experiences as if the computer is doing several things at once.
- Time slices or **quanta** are allocated to **threads** via:
  - **Priority**
  - CPU **scheduling** (via a **scheduler**)

Whoops that's a lot

We'll come back to all that mumbo jumbo.

The main point is, a CPU isn't truly doing more than one thing at a time. It is just very rapidly alternating between all running tasks so quickly that it feels simultaneous.

Running an instruction, pausing the thread, saving its state, loading the state of another thread, running an instruction, repeat.

The addition of more cores doesn't change the need for **context switching.**

The Rest is History

1983: Networking (TCP/IP - The Internet)

1984s: Graphical User Interfaces (GUIs)

- Apple Macintosh (1984) and Microsoft Windows (1985)

1990s: Linux introduces open-source development

2000s: Mobile device OS’, Virtualization and Cloud infra

Operating System: Briefly Defined

---

1. A **computer program** that
2. **multiplexes hardware resources** and
3. **implements useful abstractions**.

The OS is just another computer program.

Multiplexing allows multiple people or programs to use the same set of hardware resources safely and efficiently.

Abstractions simplify the usage of hardware resources by organizing information or implementing new capabilities.

Why Study OS in Architecture?

Reality: As an engineer you should know how computers really work.

Ubiquity: operating systems are everywhere and you are likely to eventually encounter them or their limitations.

Beauty: operating systems are examples of mature solutions to difficult design and engineering problems. Studying them will improve your ability to design and implement abstractions.

Why Study OS in Architecture?

Design: Operating Systems stress careful design and specification before coding begins.

Difficulty: Operating Systems are large existing code bases where new solutions have stringent performance requirements.

Unfortunately (or fortunately*) we don’t have much time to go deep into it, but its basic concepts will recur.

*OS is life-ruiningly hard

Operating System Abstractions

Abstractions simplify application design by:

- Hiding undesirable properties,
- Adding new capabilities, and
- organizing information.

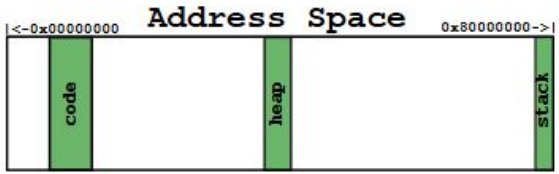
Abstractions provide an **interface** to application programmers that separates **policy** from **mechanism.**

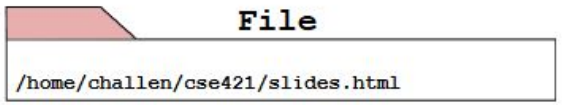
- **Policy:** *what* the interface commits to accomplishing.
- **Mechanism:** *how* the interface is implemented.

Operating System Abstractions (Examples)

- **Threads** abstract the CPU.
- **Address spaces** abstract memory.
- **Files** abstract the disk.







The Kernel

The core component of the Operating System, often used as a synonym for it, but without any of the bundled user apps.

The kernel implements all of these abstractions and more.

The kernel is what talks directly to the hardware.

- User programs are no longer responsible for this.
- This comes with safety and security guarantees*.

*in theory, hello crowdstrike

And also Windows drivers in general

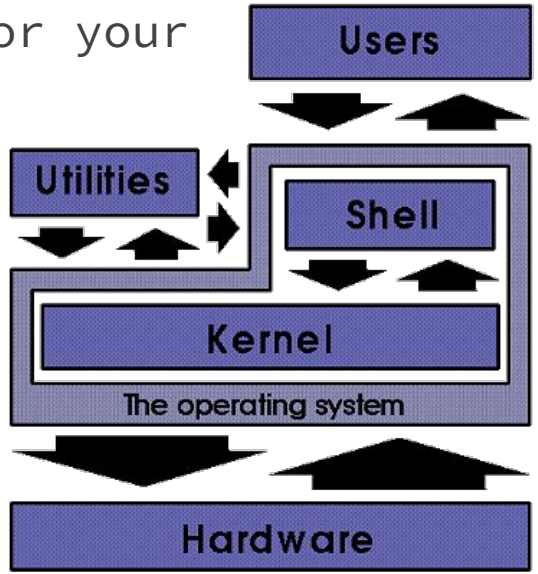
Types of Operating Systems

Whatever the user model, or commercial model, the Operating System drives the capabilities of the computer system.

The OS capabilities often set boundaries for your application architecture.

Types of OS architectures

- Monolithic systems
- Layered systems
- Microkernels

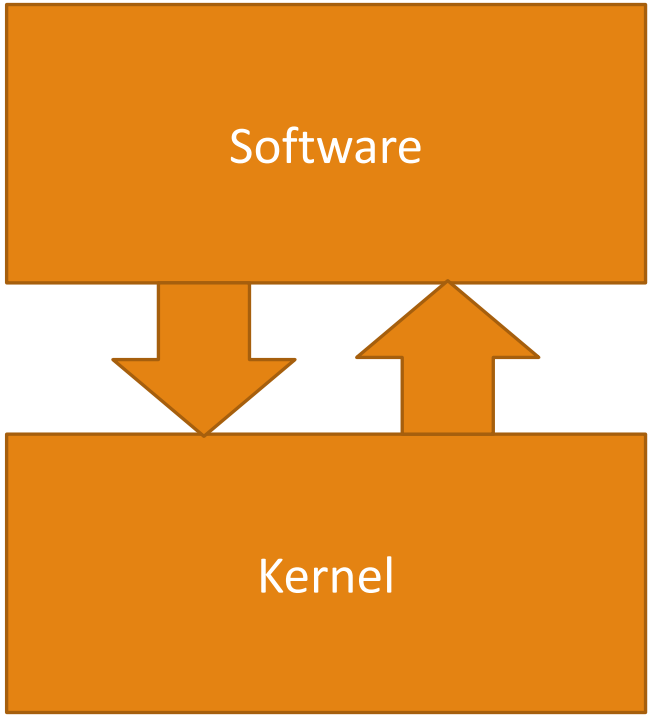


Unix Anatomy

Monolithic

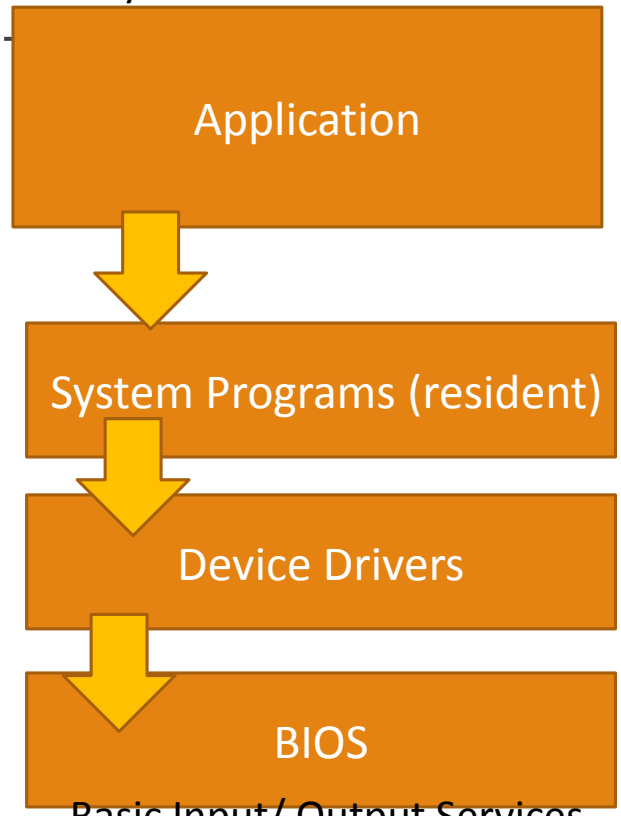
Early computer systems

Basic embedded controllers circa 1980s



(Semi)-Layered

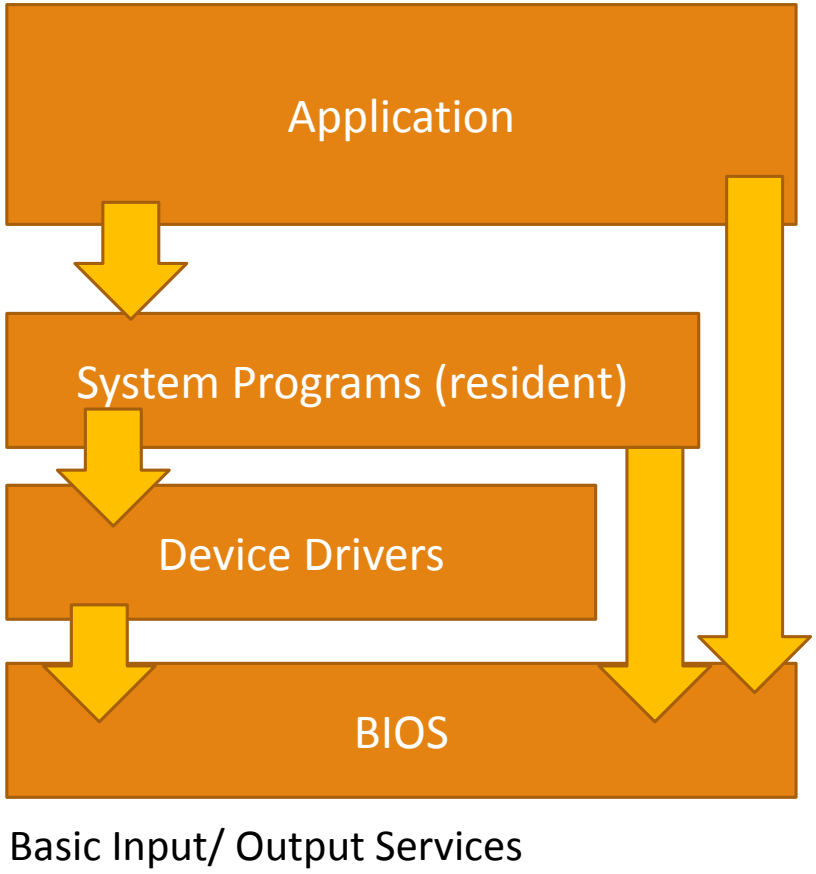
Theory



Each layer talks to the next layer

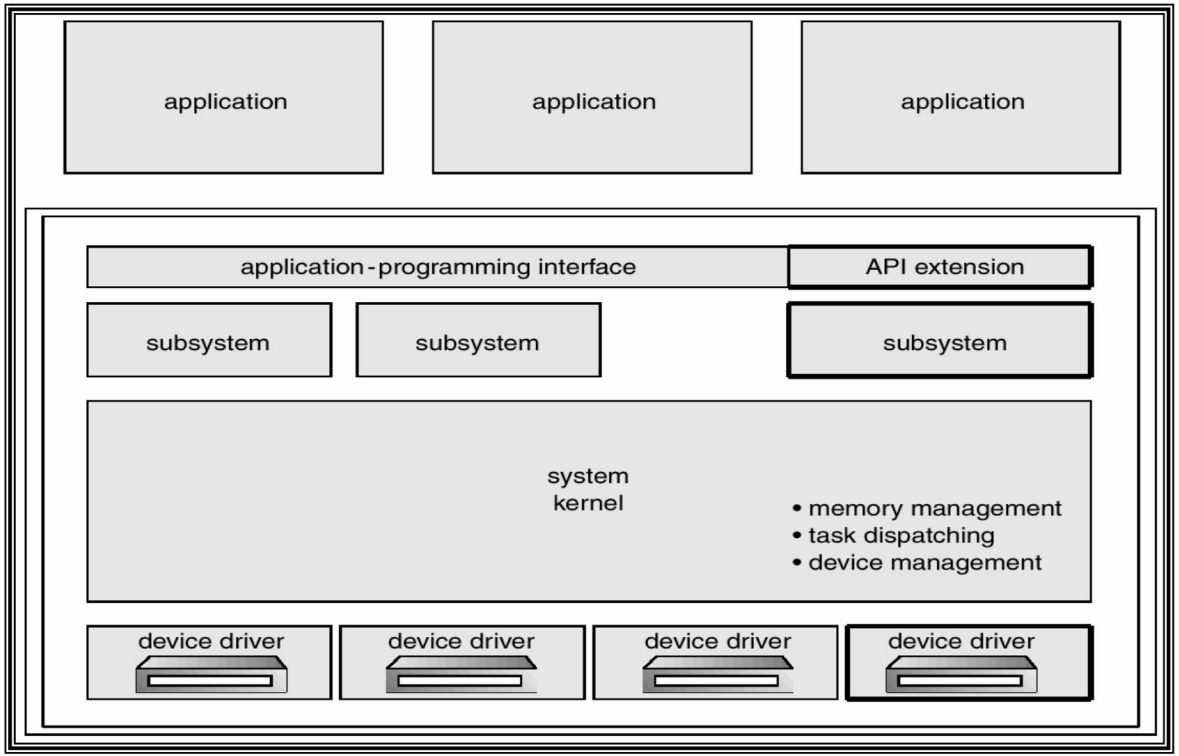
'BIOS' is in chips, and 'boots' the OS

DOS



Improved OS Architectures (OS/2 1987)

OS/2 Layer Structure



Enforced layers.

Advantages?

- Isolation
- Stability

Kernel Space and User Space

User Space

- Applications run here, must ask kernel for things.

Kernel Space

- Protected, low-level OS functions, full hw access.

Modern OS’s eliminate dangers of programmer error by preventing Applications from getting to Kernel Space.

User Space

Only sees memory and resources the kernel wants it to see.

User threads must raise an **exception** if they want something from the kernel.

- Then they must wait for the kernel to validate everything and return the results.

When application code crashes, only the application crashes.

Applications are **isolated** from each other.

- The kernel manages this via **virtual memory**.

Kernel Space

Full access to hardware and all operations performed on it.

Loaded prior to full OS initialization.

If your software runs in the kernel it can **see and do anything on your entire system.**

When kernel code crashes, the whole system crashes.

This is where device drivers run:

- In Linux, they are **built into kernel**, all are tested & maintained with each kernel release.
- In Windows, they are **installed manually**, most untested/unmaintained - Primary cause of BSOD

Always try to find another way...

Code should only run in kernel mode if it **absolutely has to.**

The Linux ecosystem is a lot more strict about this!

Crowdstrike Falcon

Antimalware for the server:

- Analyzes application behaviors to predict novel attacks.

To see everything it needed to, it ran in the kernel.

- Unfettered access to system data structures and services.

Code written as a device driver (despite no device) to get **Ring 0** access.

- Bypassed WHQL certification to run code not signed by Microsoft in Kernel Mode.
- Marked itself a boot-start driver (Windows won’t boot w/o it!)

Separating Kernel Space and User Space

Memory protection creates isolation per process.

- Keeps user programs from crashing *each other* **and** the operating system itself.

Two hardware-supported mechanisms:

- Address translation (Memory)
- Dual-mode operation (CPU)

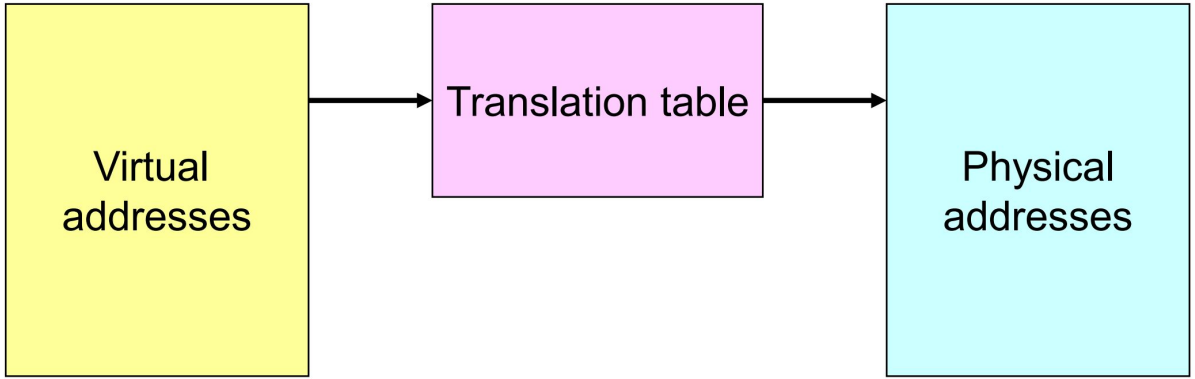
How Memory Protection Works

Each process is associated with and **address space**. This is all the physical addresses a process can touch.

However, each process *believes* that it owns the entire memory, starting with **virtual address** zero.

A **translation table** translates every memory references from a virtual address to a physical address. This is a core operating system function.

Address Translation (Virtual Memory)



- Translation provides protection
  - Processes cannot talk about other processes' addresses, nor about the OS addresses
- OS uses physical addresses directly
  - No translations

Why do you get kernel panic and blue-screens?

Virtual Memory

What is swap space?

The Process

Processes are the most fundamental operating system abstraction.

- Organize information about other abstractions (a **process** is given an **address space**)
- Represent a single thing that the computer is “doing.”
- A typical user application is a process.

Organizing Information

Unlike threads, address spaces, and files, processes are **not tied to a hardware component.** Instead, they contain other abstractions.

Processes contain:

- One or more **threads**
- An **address space**
- Zero or more open **file handles**

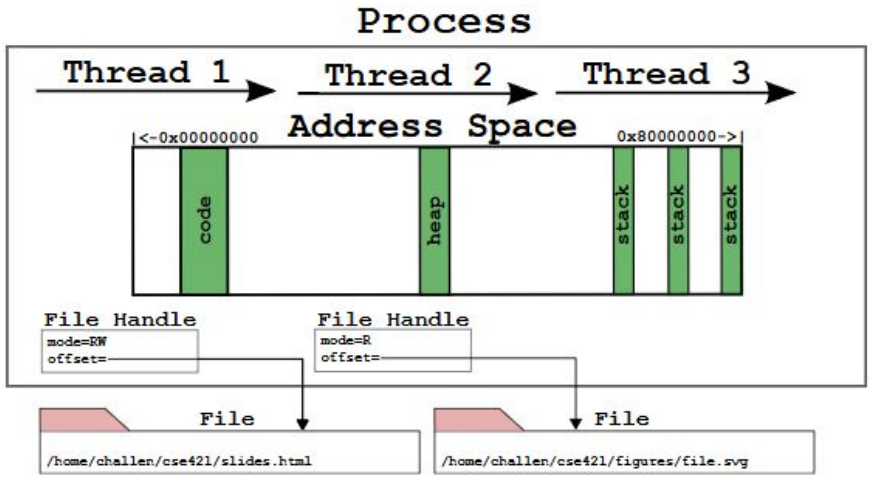
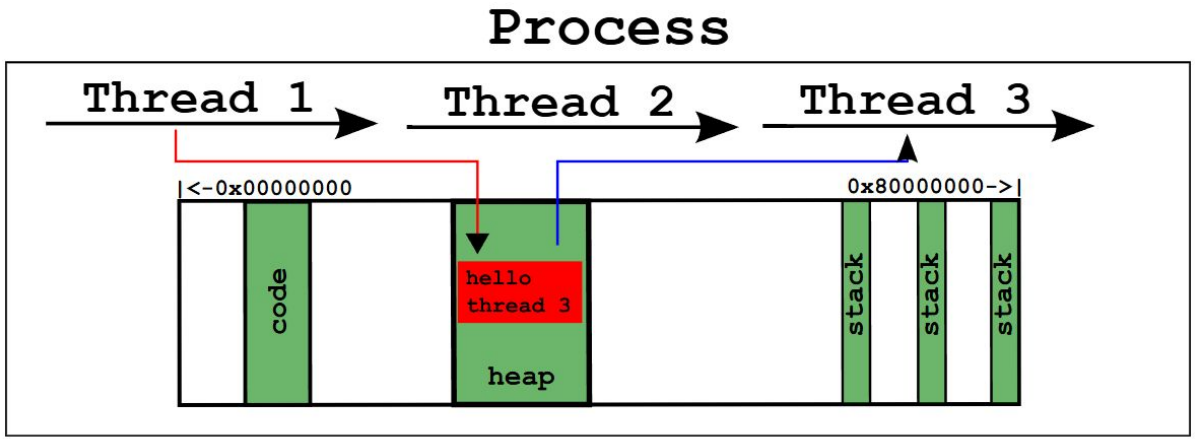


Figure 2. The Process

Intra-Process Communication: Easy

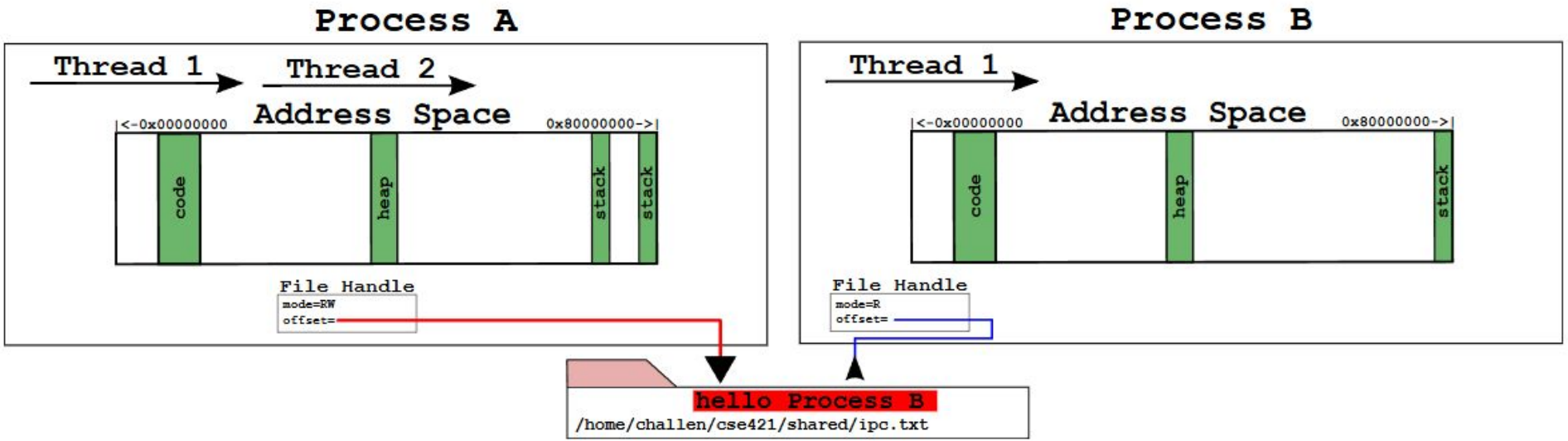
Communication between multiple threads in a process can be accomplished via shared memory.

Threads within a process also share open file handles and both static and dynamically allocated global variables.



Inter-Process Communication: Harder

Requires coordination. We will discuss this next time.



Processes vs Threads 1

We can describe both a process and a thread as **running.**

A process requires multiple resources: the CPU, memory, files, etc.

A thread of execution **abstracts** CPU state.

Processes contain threads; threads belong to a process.

A process is considered to be running when one or more of its threads are running.

Processes vs Threads 2

Threads are an operating system **primitive**.

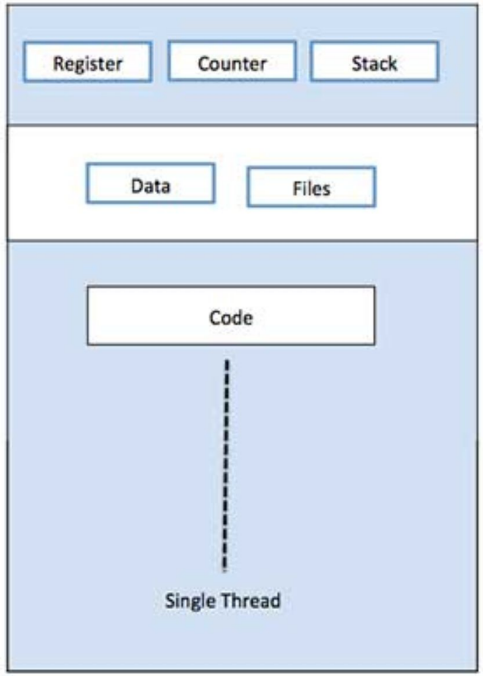
- Describe the CPU's place in the execution of a particular sequence of instructions, which can be paused, stored, and suspended. This allows for **context switching**.

Processes are an operating system **higher construct**.

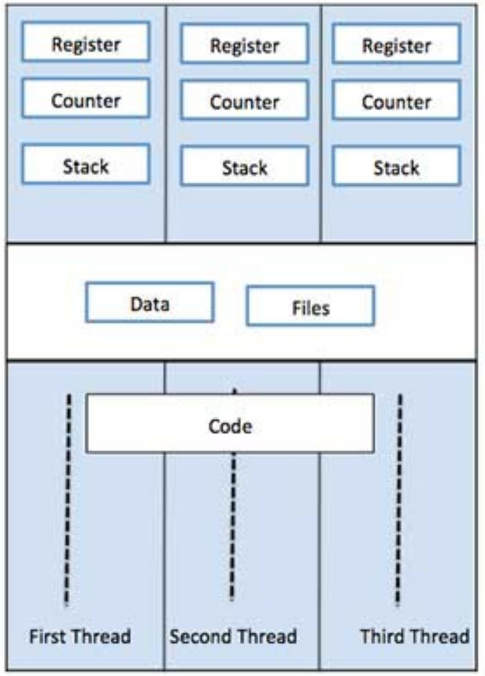
- Contain threads, manage their access to memory and files.
- Contain information about threads therein.
- Can be seen as a **memory management concept**.

Process vs Thread Visualized

- Processes are heavyweight **encapsulations**.
- Threads have access to other threads' data within the same process.
- Processes are operating system **structures**.
- Threads are **sequences of instructions** executing on the CPU.



Single Process P with single thread



Single Process P with three threads

Process Example: Firefox

———

Firefox has multiple threads. What are they doing?

- Processing interface events: mouse clicks, keyboard input, etc.
- Redrawing the screen as in response to user input, web page loading, etc.
- Loading web pages—usually multiple parts in parallel to speed things up.

Firefox is using memory. For what?

- Firefox.exe: the executable code of Firefox itself.
- Shared libraries for web page parsing, security, etc.
- Stacks storing local variables for running threads.
- A heap storing dynamically-allocated memory.

Firefox has files open. Why?

- Configuration files.
- Fonts.

Note on Multiprocessing

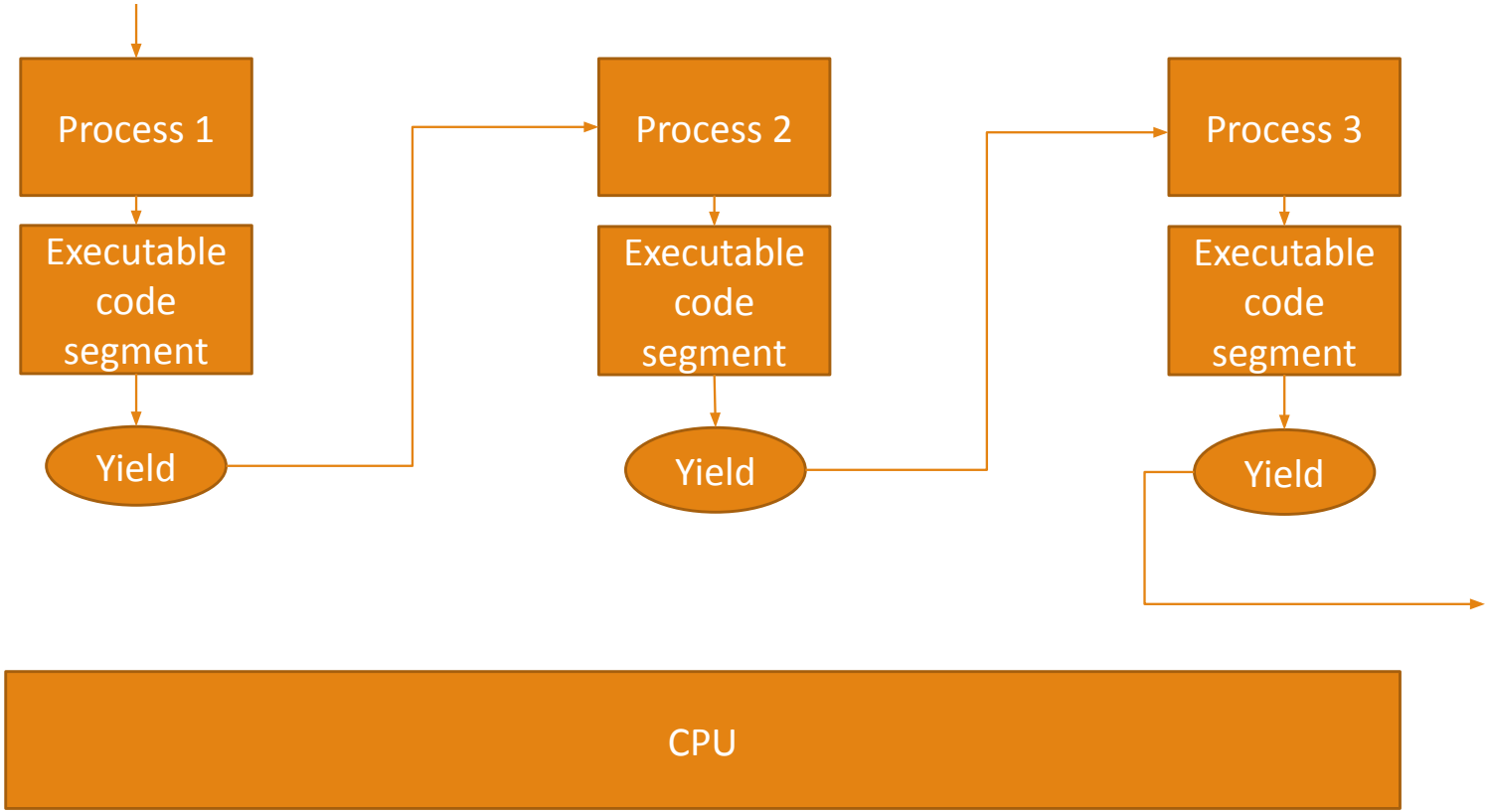
It is technically possible to implement multiprocessing without address space isolation, but it is very dangerous.

There are other ways of programming multiprocessing without these OS concepts, such as **Cooperating Multitasking.**

Let’s take a look at some visualizations.

Cooperative Multitasking

- All processes are memory resident
- Running process has full control of CPU
- It must explicitly 'yield' control to allow another process to run



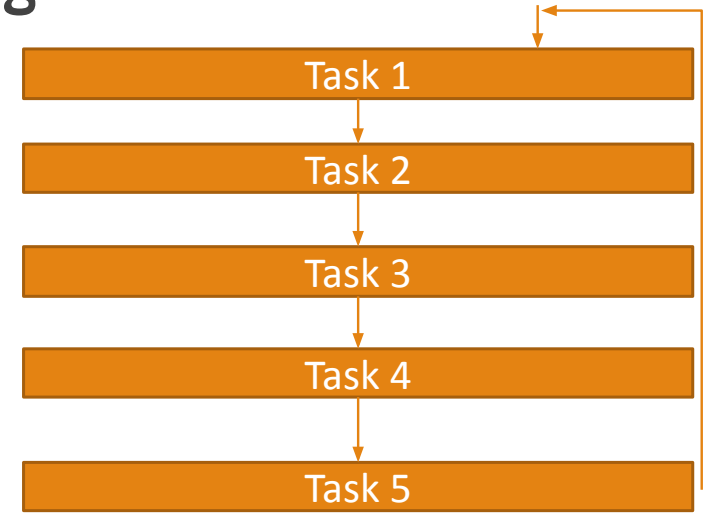
e.g. Windows pre-Windows 95

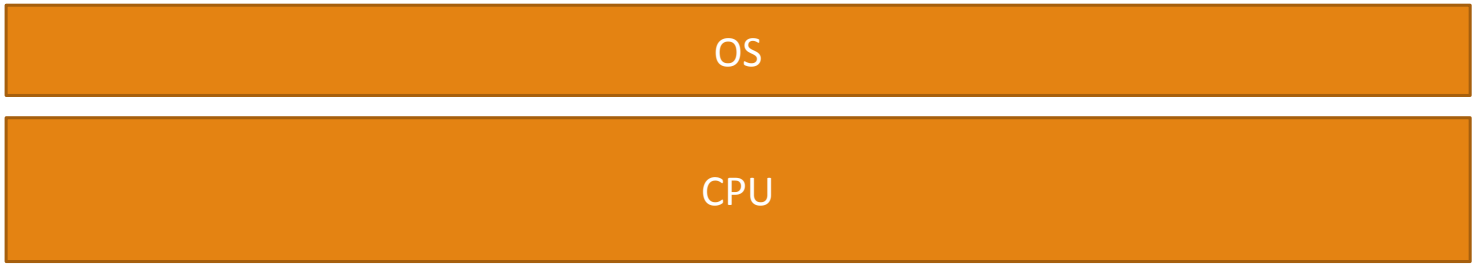
Cooperative Multitasking

- Also called “non-preemptive multitasking.”
- OS never initiates a context switch.
- Processes must **yield** to each other.
- Not used in modern operating systems, but still used in memory constrained **embedded systems**.
- Programming for this is similar to programming for multi-threaded applications today.

Time-sharing Multitasking

- Processes (tasks) are assigned a ‘time to run’ (sometimes by priority)
- OS Schedules a time slot for each
- Runs tasks until time expires
- Switches context and runs next priority process





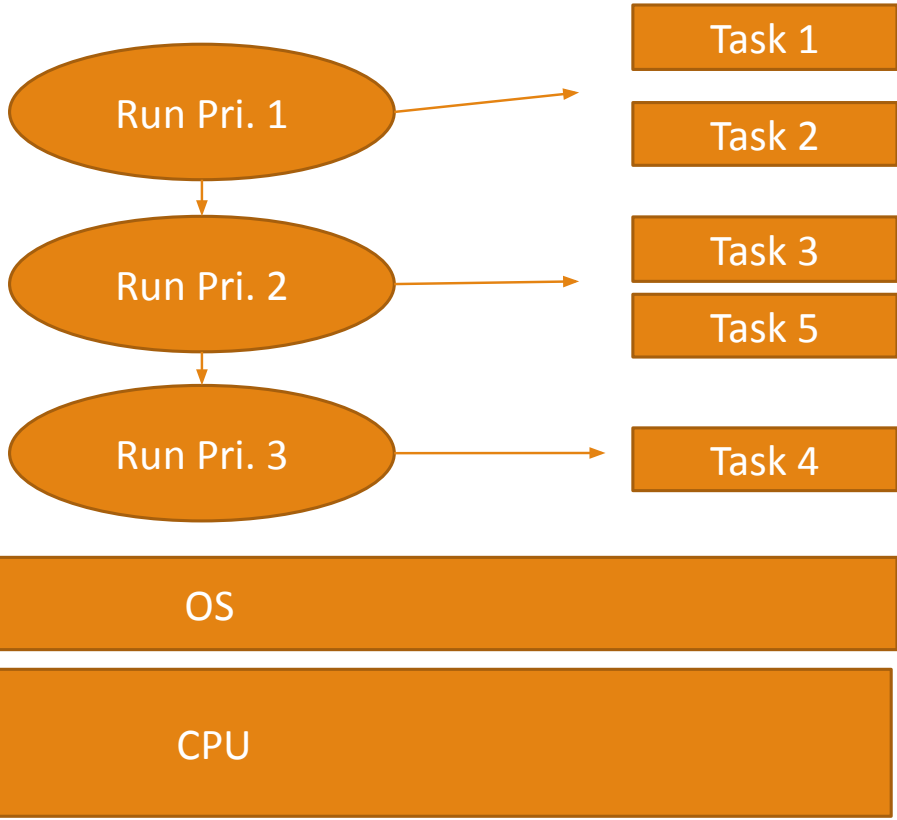
Time-sharing Multitasking

- OS schedules **time slots.**
- Task runs until time expires.
- State of process p1 is suspended.
- **Context switch** to another process, loading p2's previously suspended **state.**
- **Scheduler** prevents **starvation:**
  - makes sure no process is "starved" of ability to make progress, will always come back around.

Priority Preemptive Multitasking (Modern)

- Processes (tasks) are assigned a priority
- OS Schedules a time slot for each
- Runs tasks until time (per priority) expires
- Interrupt triggers OS
- Switches context (under control of OS) and runs next priority process (pre-empts process, even in the middle of execution)

Task ID	Priority
1	1
2	1
3	2
4	3
5	2



Priority Preemptive Multitasking

- Time-sharing with **preemption.**
  - Time slices can be “preempted” if the OS judges something else is more urgent.
- Preemption means that the OS can end a process’ time slice even in the middle of execution and resume it later
- **Interrupts** trigger context switches.
- Follows **priority queue** logic PLUS preemption.

Priority Queue Logic

- Highest priority tasks are scheduled first, first-come-first-serve internally.
- OS will preempt a running process if a higher priority one arrives.
- Whereas a typical priority queue would wait for the lower one to complete (non-preemptive).

A Note on Microkernels

We discussed older monolithic kernels and modern layered systems, but what about Microkernels?

Microkernels aim to run only the most **essential** functions in kernel space:

- Inter-Process Communication (IPC)
- Memory Management (Allocation/Deallocation)
- Address Space Management (virtual memory or other)

Other OS services, like file systems, devices, and networking run in user space.

Key Features of Microkernels

- **Modularity:** Components can be updated more easily without affecting the entire system.
- **Fault Isolation:** A failure in one services doesn’t necessarily crash the entire system
- **Security:** Smaller attack surface since less code runs in privileged kernel mode

What is this useful for? **Embedded systems** for small resource footprint. **Distributed systems** with reliance on **microservices** and other independent plug-and-play services.

Reminders

New Quiz Out Today

- Computer Arch and OS 1 (Due 9/18)

Active Assignments:

- SAiP Chapter 1 Discussion, Question 2 (Due Mon 9/07 @ 11:59pm)

Next Time: IPC (Inter-Process Communication)