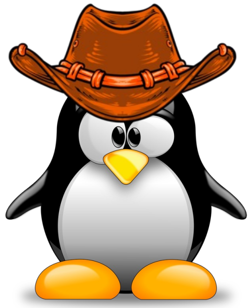


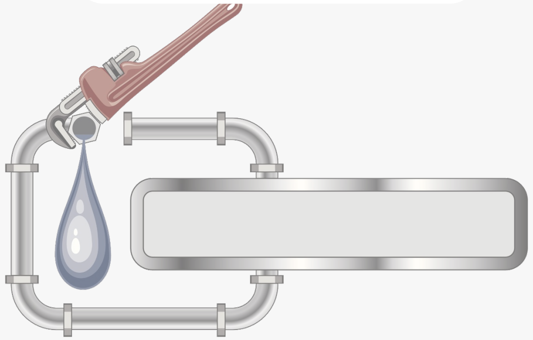
CSE 330: Operating Systems

Adil Ahmad

Lecture #2: Privileges, design, and general concepts

A modern OS wears many hats!

Plumbing



Bootloader/BIOS

Privilege management

Device drivers

MMIO/DMA interface

Managing

Memory management

Process loading

Process scheduling

Virtualization

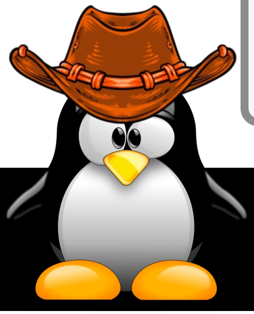
Policing

System auditing

Enclave computation

Defences, e.g., KASLR

Software isolation



What makes the OS so special that it can do its tasks?

First, let's take a closer look at CPUs

- Let's think of CPUs in *very naïve* terms as having the following:
  - **Instructions** with hardcoded operations (e.g., add)
  - **Registers** where operations (e.g., add) are performed

- Code: x = 10 + 20. CPU executes the following "instructions":
  - LOAD 10 from memory to a register (rax)
  - LOAD 20 from memory to a register (rbx)
  - ADD rbx to rax
  - STORE rax to memory

- LOAD, ADD, and STORE are examples of **memory and arithmetic instructions** implemented by your CPU

Some other instructions exposed by modern CPUs

- **Memory and arithmetic instructions**
  - ➢ LOAD (read data from memory into register)
  - ➢ ADD (add values in two registers)
- **Device access instructions**
  - IN → read data from device (e.g., SSD)
  - OUT → write data to device
- **CPU shutdown instructions**
  - HLT → stop the CPU from spinning
- … (many more)

Which of these instructions look "sensitive" to you?

- **Memory and arithmetic instructions**
  - LOAD (read data from memory into register)
  - ADD (add values in two registers)

- **Device access instructions**
  - IN → read data from device (e.g., SSD)
  - OUT → write data to device

Program can overwrite important files in SSD

- **CPU shutdown instructions**
  - HLT → stop the CPU from spinning

Program can shut down your computer on a "whim" or mistake

- … (many more)

CPU privilege modes **restrict** sensitive instructions

User mode
(U-mode)

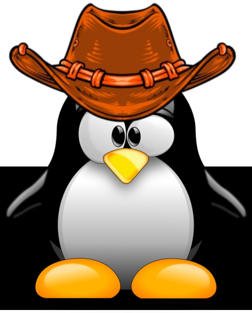
Only **non-sensitive (unprivileged)** instructions allowed:

- Memory access (e.g., load, store)
- Arithmetic (e.g., add, subtract)
- …

Supervisor mode
(S-mode)

Both **non-sensitive** and **sensitive (privileged)** instructions allowed:

- Memory access (e.g., load, store)
- Arithmetic (e.g., add, subtract)
- Device access (e.g., in, out)
- Shutdown (hlt), ….



What makes the OS so special that it can do its tasks?

Only OS executes at the S-mode and controls "privileged" functionality

The effective impact of user-mode on programs

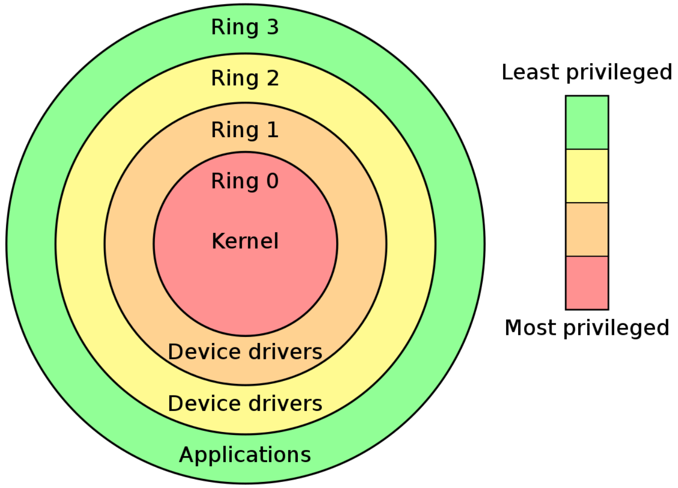
- Cannot directly read/write to storage
- Cannot access devices like hardware timers, USB, GPU, etc.
- Cannot arbitrarily change the memory regions allocated to program
- Cannot read/write other process' memory regions
- Cannot read/write kernel regions, etc...

Does not impact the program itself → limits changes a program can make outside of its own scope

Privilege modes can be *even more expressive!*

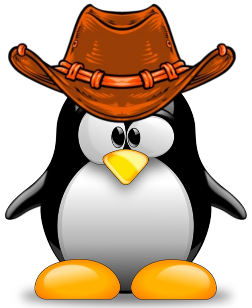
Each **higher privilege mode** allows **more control** over system functions

X86 privilege rings



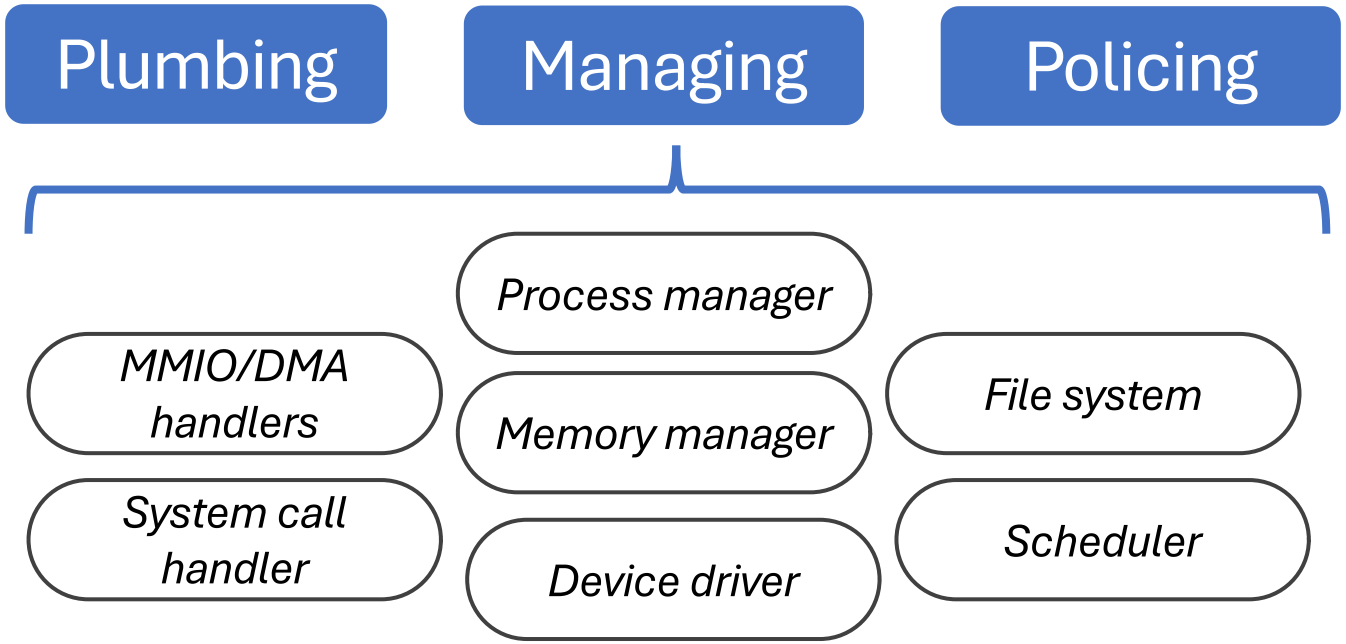
A software should only have as much privilege as it needs.

(The principle of least privilege)



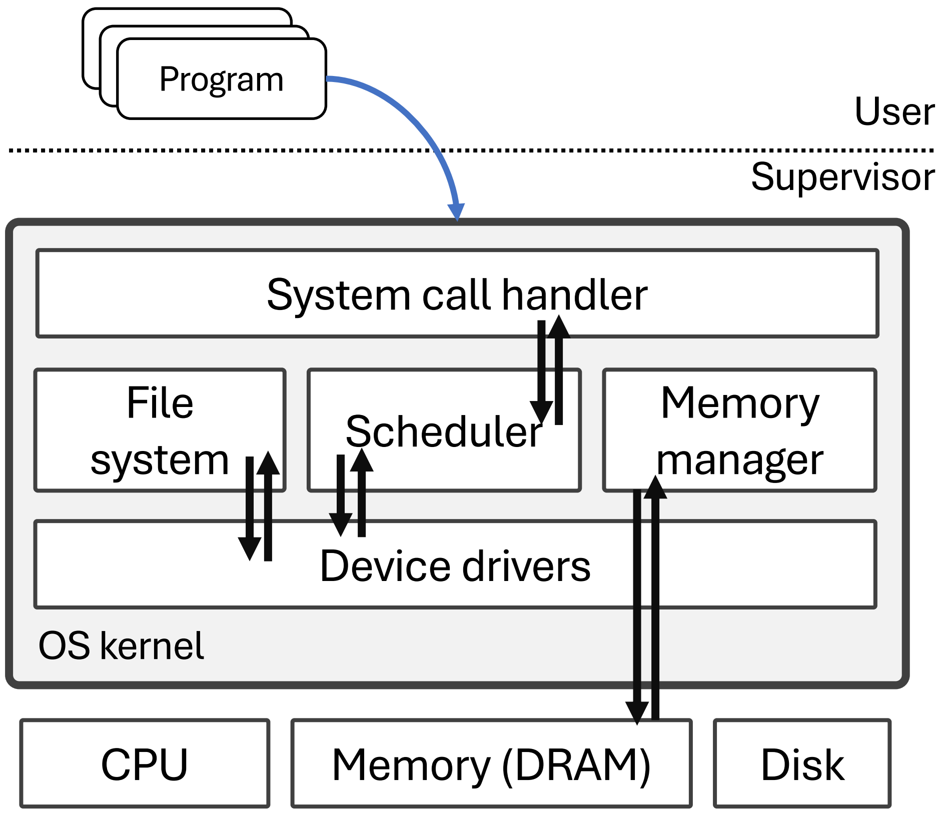
First topic: designing an OS

The three "hats" of the OS require lots of code



How should we combine the different code together?

Let's start with one large software: **a monolithic kernel**



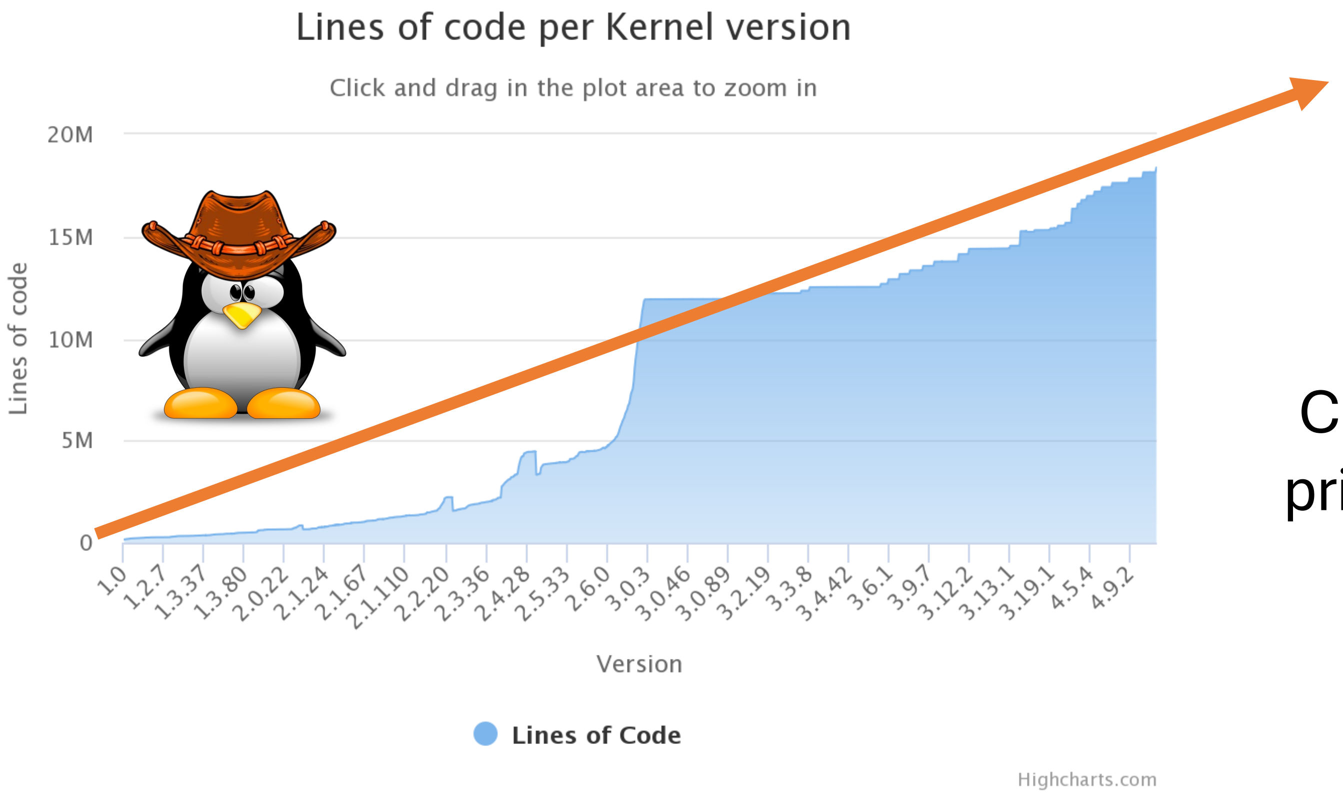
The OS is one "big" program that runs within the same (S-mode) privilege

Easier for kernel *sub-systems* to collaborate – no irritating boundaries

All kernel code runs within S-mode – no communication slowdown

Why not to keep a monolithic kernel?

A big kernel means complexity, which reduces innovation and changes

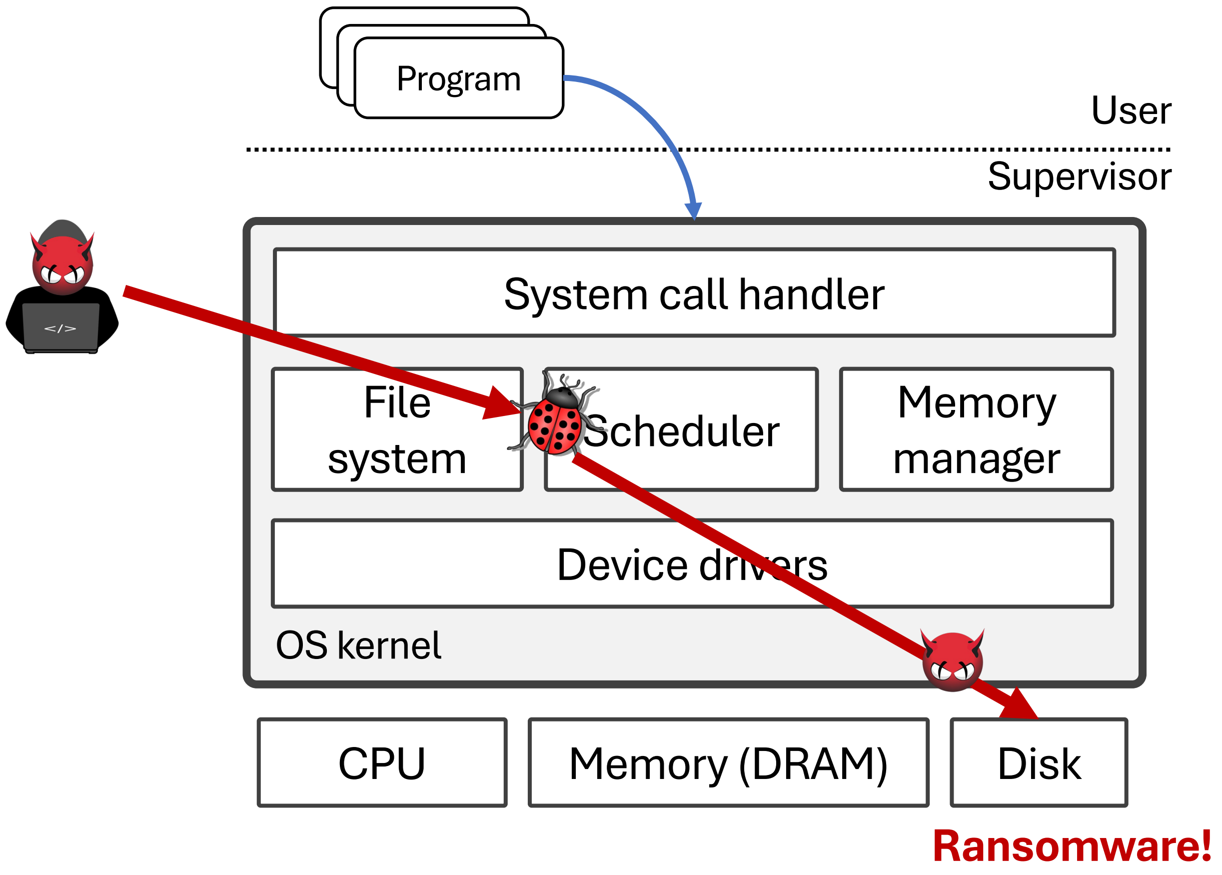


More than 31 million code lines today!

Changes must not break prior functionality → hard

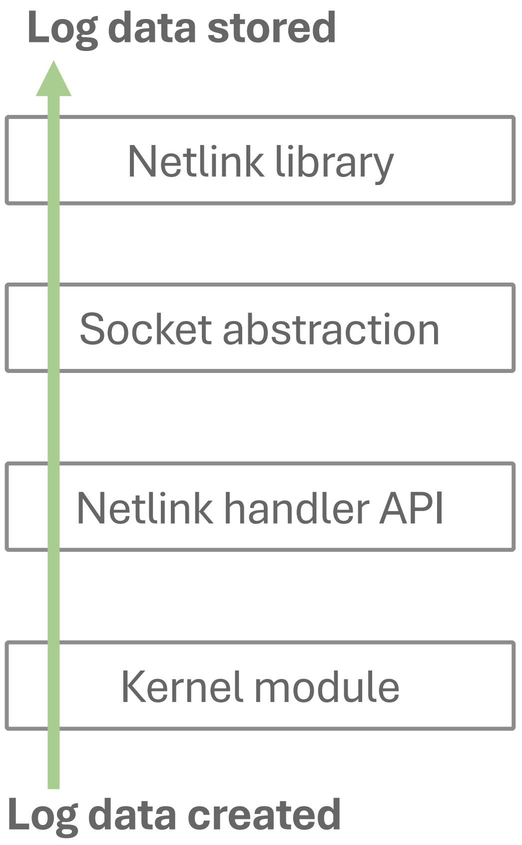
Why not to keep a monolithic kernel?

No internal enforcement means minor bugs → crashes or full compromise

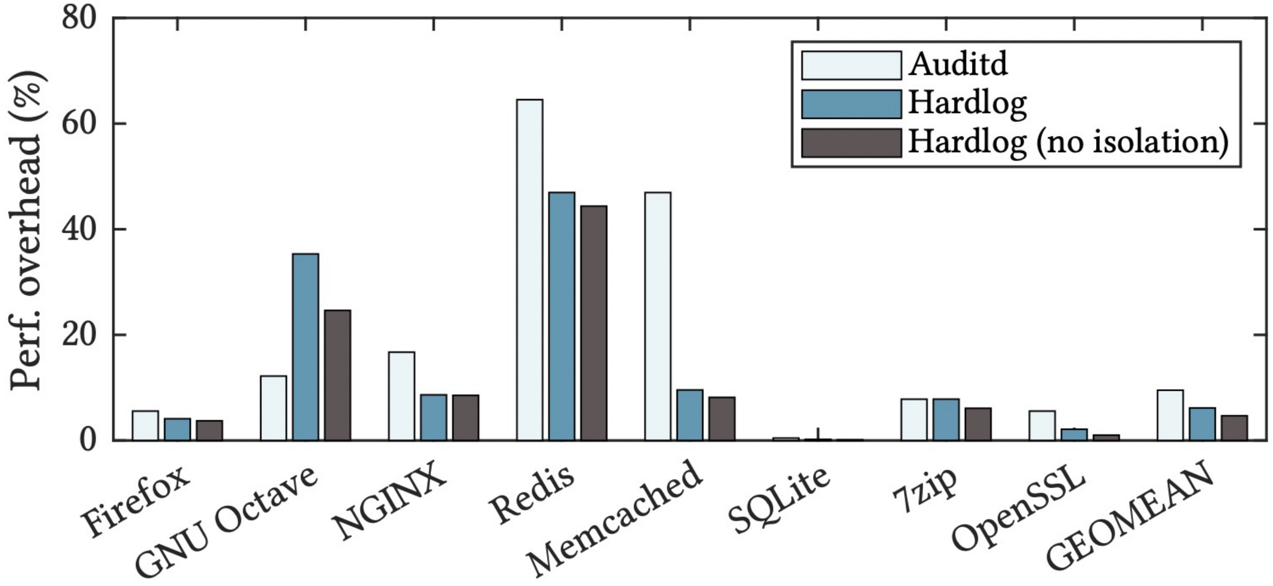


Why not to keep a monolithic kernel?

Over-general and emphasis on reusing abstractions, which can be inefficient

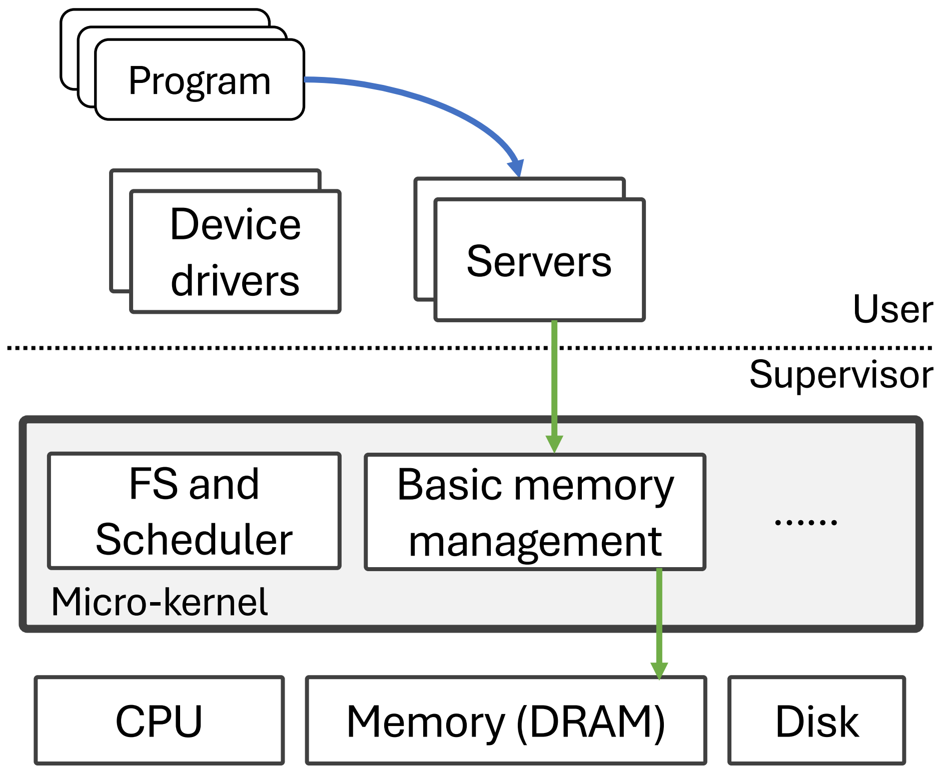


Avoiding different layers can improve performance significantly



Hardlog (Ahmad et. Al, Oakland '22)

Let’s look at the alternative: a *micro-kernel*



Move most of the OS’ functionality to user-space *servers* – *Small kernel*

Independent *servers* can use well-defined communication APIs – *Easy to update servers*

Minor bugs in *user-space servers* cannot compromise full system – *Better security*

Cardinal rule of designing a micro-kernel

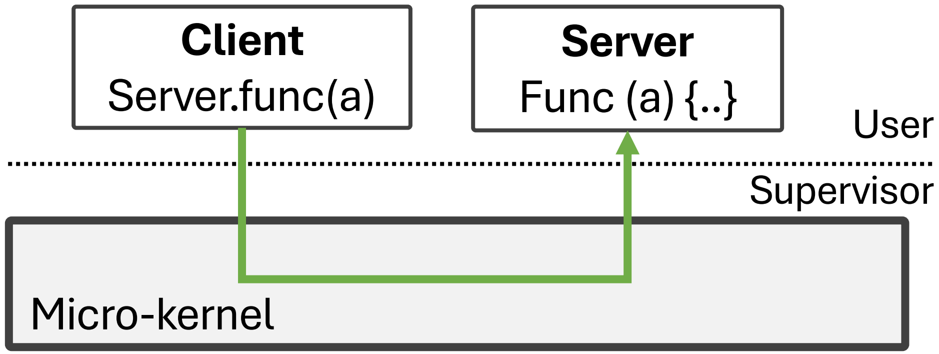
"A concept is **tolerated inside the microkernel** only if **moving it outside** the kernel, i.e., permitting competing implementations, **would prevent the implementation** of the system's required functionality."

(Jochen Liedtke, one of the inventors of the L4 micro-kernel)

Why not to keep a micro-kernel?

IPC makes **communication between components slow**

➢One of the biggest challenges older micro-kernels faced (e.g., pre-L4)



Protected procedural call (PPC)

PPC entails:

(a) Switching privilege
(b) Copying args/results
(c) Blocking

These aspects make communication slow

Why not to keep a micro-kernel?

- **Compatibility is also a problem**, since micro-kernels do not need to follow the widely-used APIs (communication protocols)
  - E.g., POSIX standards used by monolithic kernels
- Each version of a "user space" server can have its own API which can be evolved somewhat arbitrarily
- Programs must follow that API for execution

OS design (and life) is all about **trade-offs!**

- Monolithic has better performance and compatibility
- Micro-kernel has better modularity and security

Consumer OSs adopt a monolithic approach *with elements of micro-kernels*

- The kernel is kept *as small as possible*
- But most servers (e.g., device drivers) are in the privileged kernel space

Small security-critical systems (e.g., secure co-processors) use micro-kernels

Windows 11

Fuchsia by Google

seL4 Security. Performance. Proof.

My favorite design: a *nested kernel*

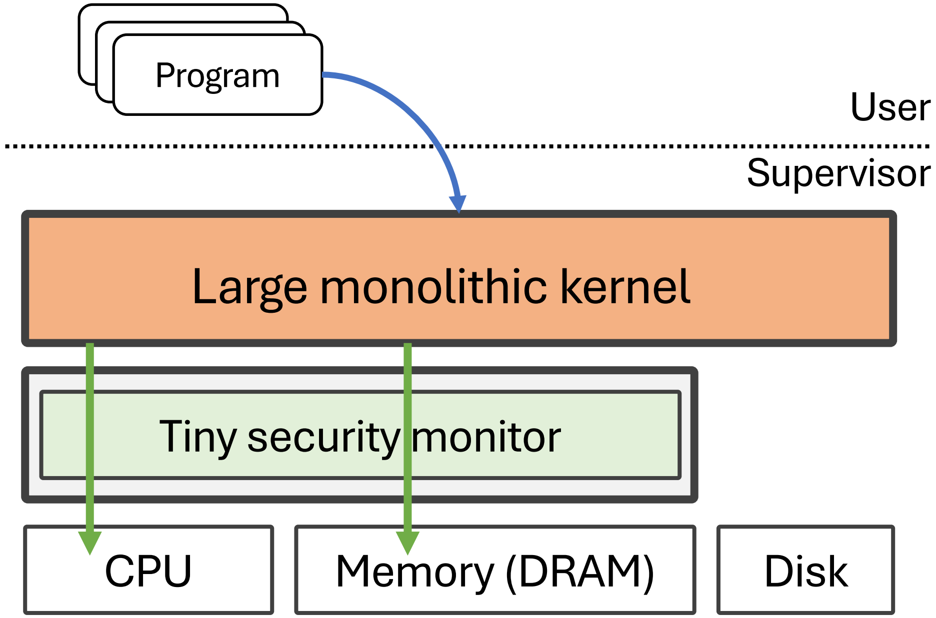
Small monitor *(the nested kernel)* that interposes HW and monolithic kernel

Benefits

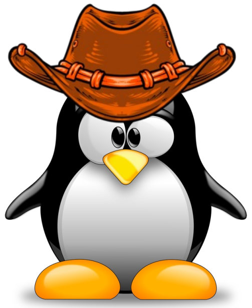
- Still a simple-ish design
- Nested kernel can enforce security policies on buggy monolithic kernels

Drawbacks

- Hard to enable isolation within same privilege level (*software tricks?*)
- Slower than a monolithic kernel



Nested Kernel (Dautenhahn et. Al, ASPLOS '15)



General OS concepts to get us started

What is the difference between the “OS” and “kernel”?

- Kernel is “core” part of the operating system that interacts with the hardware directly (e.g., for memory management, scheduling, etc.)
- The ”OS” is essentially a collection of the ”kernel” and pre-packaged user-level software. These software basically provide:
  - Graphical user input (GUI)
  - Window management
  - File explorers, etc.
- In this class, whenever I say OS, I really mean the “kernel” :p

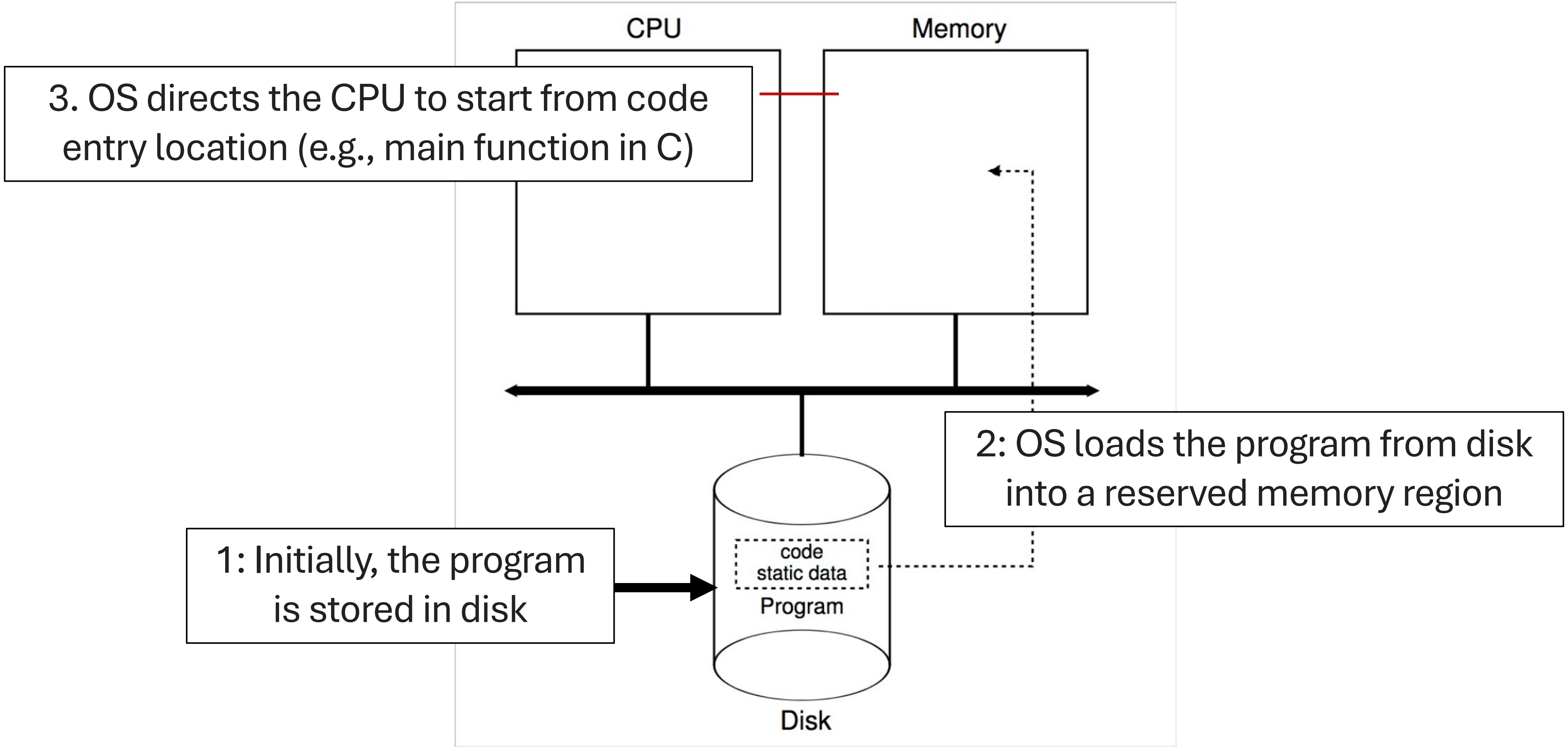
Integral concept of a **process** in OS

- *Programs* are a set of codes (static entity) or executable files
  - e.g., compile a “Hello World” C program → a.out
- *Processes* are the *running version of* programs
  - We can run the same program in multiple processes
  - One host can run multiple processes
- Let’s check a C++ analogy
  - Definition of a class→ “program”
  - Object instance of a class → “process”

What events result in process creation?

- Principle events that initiate the creation of processes
  - System initialization (e.g., *init* process)
  - Execution of a process creation system call by a running process
  - User request to create a process

An illustration of typical process creation



Each process has an (isolated) **address space**

- A process requires hardware resources to execute, including CPU and main memory (also called DRAM)
  - DRAM → Dynamic Random Access Memory

- OS does not show all system memory to a process, rather abstracts a portion of memory to each process

- View of memory shown to each process is called its “address space”
  - Typically, each process’ memory view is different from others

(More details about address spaces in lecture #8 onwards.)

Process can spawn other instances of itself

- Ask OS to create a completely different clone of the same program.
  - Clone will be a new "process" with isolated memory
  - E.g., running two instances of a web browser

- Ask OS to execute it *simultaneously* on multiple CPUs
  - Each context is what we call a **thread**
  - Threads share the same memory but can execute different functions independently of each other
    - Useful for information sharing
  - E.g., two tabs within a web browser (*not always true* but for illustrative purpose!)

- We will learn all about "multi-processing" and "multi-threading"!

Recall that processes (and their threads) are *unprivileged*

Will the process ever require access to “privileged” functionality?

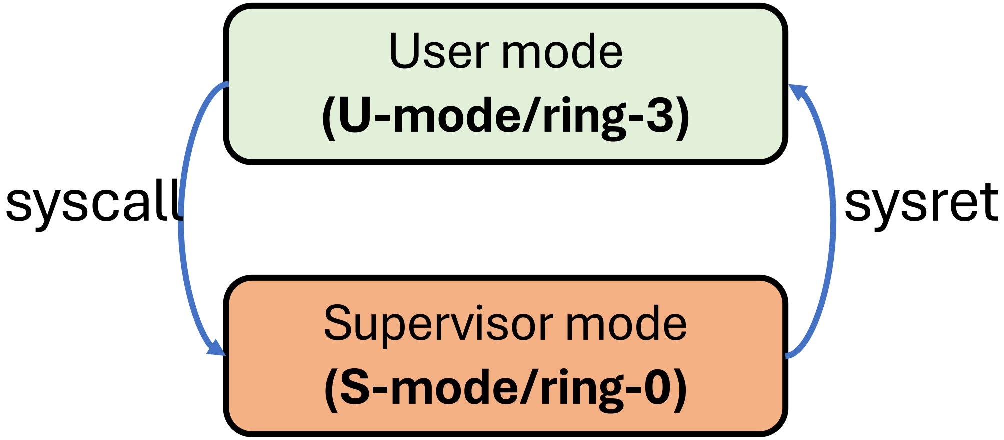
- **Yes!**
  - Reading a file from the SSD
  - Writing to a file in USB storage
  - Sending data through the network
  - Reading a hardware provided timer, etc.

- We **need a mechanism to ask the operating system kernel** to help us with privileged tasks
  - OS kernels can check if the process is “authorized” or not

Process interacts with the kernel using system calls

Requests made to the OS from a user process for a certain operation (e.g., file system access, device access, etc.)

Made using special instructions in modern CPUs (e.g., SYSCALL in x86)



System calls require:

(a) Switching privilege/address spaces

(b) Copying arguments/results

(More details about transitions in lecture #9 onwards.)

Some examples of famous system calls in Linux

fd = open(file, permissions, ...)

- *Open a file for reading, writing, or both*

s = close(file)

- *Close an open file*

n = read(fd, buf, nbytes)

- *Read data from a file into a buffer*

n = write(fd, buf, nbytes)

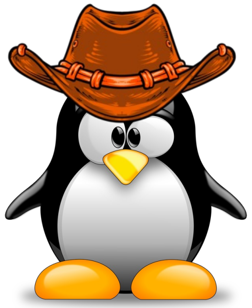
- *Write data from a buffer into a file*

pos = lseek(fd, offset, whence)

- *Move the file pointer*

s = stat(name, &buf)

- *Get a file's status info (e.g., size, etc.)*



That’s all, folks. See you in the next class!