The OS, Privileged Mode & Exceptional Control Flow

Anne Bracy

CS 3410

Computer Science

Cornell University

The slides were originally created by Deniz ALTINBUKEN.

P&H Chapter 4.9, pages 445–452, appendix A.7

Clicker Question

To what extent does the clicker grade component affect your class attendance?

A) The clickers do not affect my class attendance.
B) I attend this class slightly more often because of the clickers.
C) If there were no clickers, I would be here **way** less often.
D) My clicker is answering this question, because my friend is holding my clicker. I am still in bed.
E) None of these describes me.

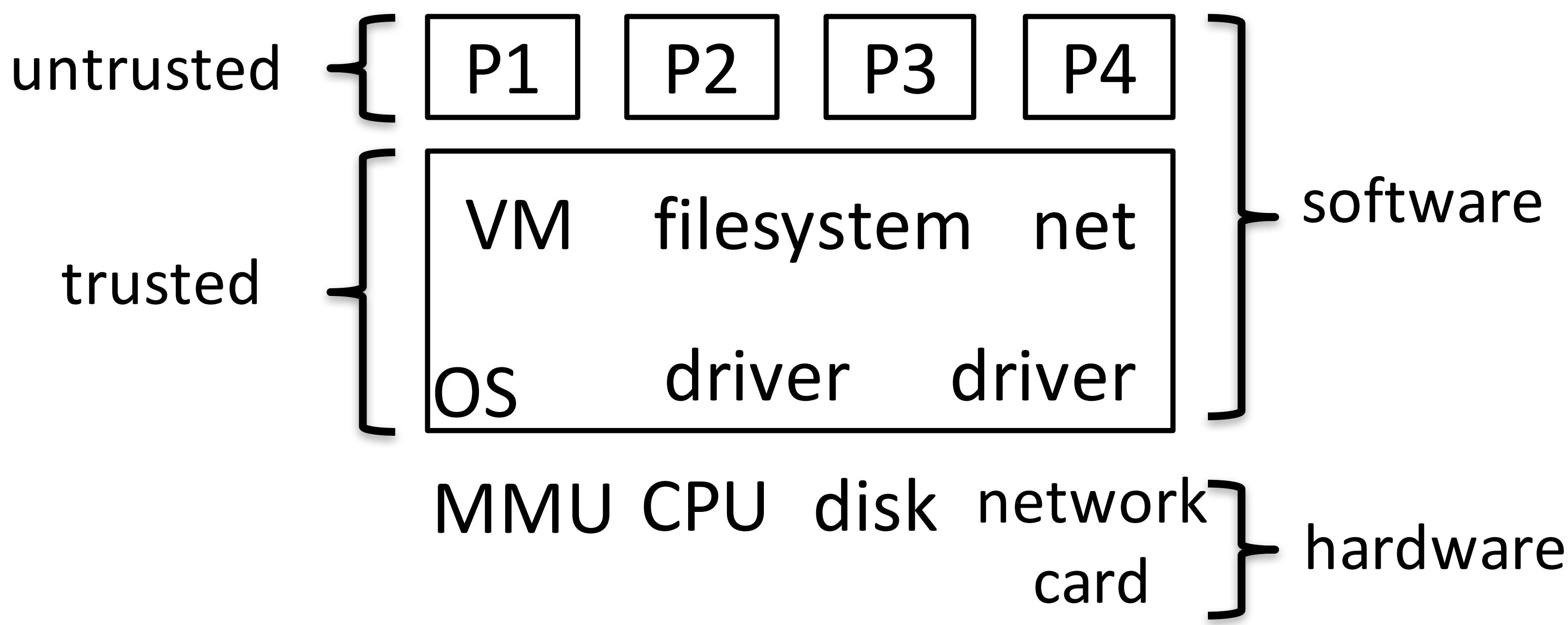
Operating System

- Manages all of the software and hardware on the computer
- Many processes running at the same time, requiring resources
  - CPU, Memory, Storage, *etc.*

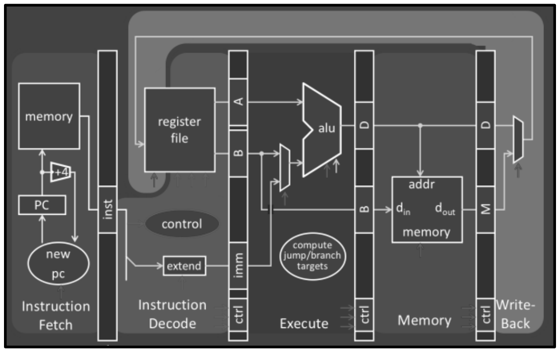
- The Operating System multiplexes these resources amongst different processes, and isolates and protects processes from one another!

Operating System

- Operating System (OS) is a trusted mediator:
  - *Safe control transfer between processes*
  - *Isolation (memory, registers) of processes*



One Brain, Many Personalities



Brain

You are what you execute.

Personalities:

hailstone_recursive

Microsoft Word

Minecraft

Linux ← *yes, this is just software like every other program that runs on the CPU*

Are they all equal?

Trusted vs. Untrusted

- Only trusted processes should access & change important things
  - Editing TLB, Page Tables, OS code, OS \$sp, OS \$fp…

- If an untrusted process could change the OS' \$sp/\$fp/\$gp/*etc.*, OS would crash!

Privileged Mode

CPU Mode Bit in Process Status Register

- Many bits about the current process
- Mode bit is just one of them

- Mode bit:
    - 0 = user mode = untrusted:
      "Privileged" instructions and registers are disabled by CPU
    - 1 = kernel mode = trusted
      All instructions and registers are enabled

Privileged Mode at Startup

1. Boot sequence
   - load first sector of disk (containing OS code) to predetermined address in memory
   - Mode ← 1; PC ← predetermined address

2. OS takes over
   - initializes devices, MMU, timers, etc.
   - loads programs from disk, sets up page tables, *etc.*
   - Mode ← 0; PC ← program entry point
     – User programs regularly yield control back to OS

Users need access to resources

If an untrusted process does not have privileges to use system resources, how can it

- Use the screen to print?
- Send message on the network?
- Allocate pages?
- Schedule processes?

Solution: System Calls

System Call Examples

`putc()`: Print character to screen

- Need to multiplex screen between competing processes

`send()`: Send a packet on the network

- Need to manipulate the internals of a device

`sbrk()`: Allocate a page

- Needs to update page tables & MMU

`sleep()`: put current prog to sleep, wake other

- Need to update page table base register

System Calls

System call: **Not** just a function call

- Don't let process jump just anywhere in OS code
- OS can't trust process' registers (sp, fp, gp, etc.)

SYSCALL instruction: safe control transfer to OS

MIPS system call convention:

- Exception handler saves temp regs, saves ra, …
- $v0 = system call number, which specifies the operation the application is requesting

Libraries and Wrappers

Compilers do not emit SYSCALL instructions

- Compiler doesn’t know OS interface

Libraries implement standard API from system API

libc (standard C library):

- gets() → getc()
- getc() → syscall
- sbrk() → syscall
- printf() → write()
- write() → syscall
- malloc() → sbrk()
- …

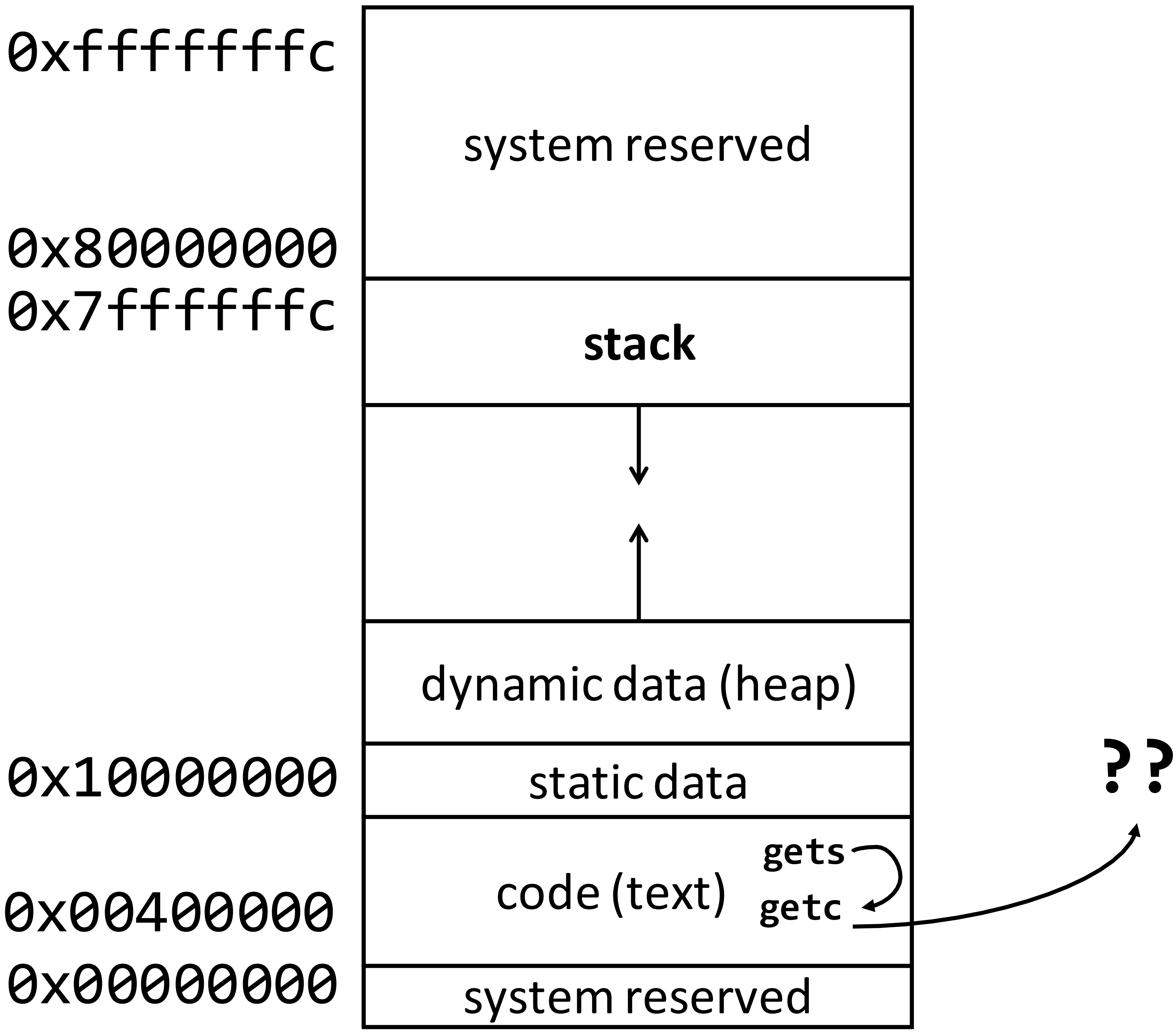
Invoking System Calls

```
char *gets(char *buf) {
  while (...) {
    buf[i] = getc();
  }
}

int getc() {
  asm("addiu $v0, $0, 4");
  asm("syscall");
}
```

4 is number for getc syscall

Anatomy of a Process, v1



Where does the OS live?

In its own address space?

- Syscall has to switch to a different address space
- Hard to support syscall arguments passed as pointers

. . . So, NOPE

In the same address space as the user process?

- Protection bits prevent user code from writing kernel
- Higher part of virtual memory
- Lower part of physical memory

. . . Yes, *this is how we do it.*

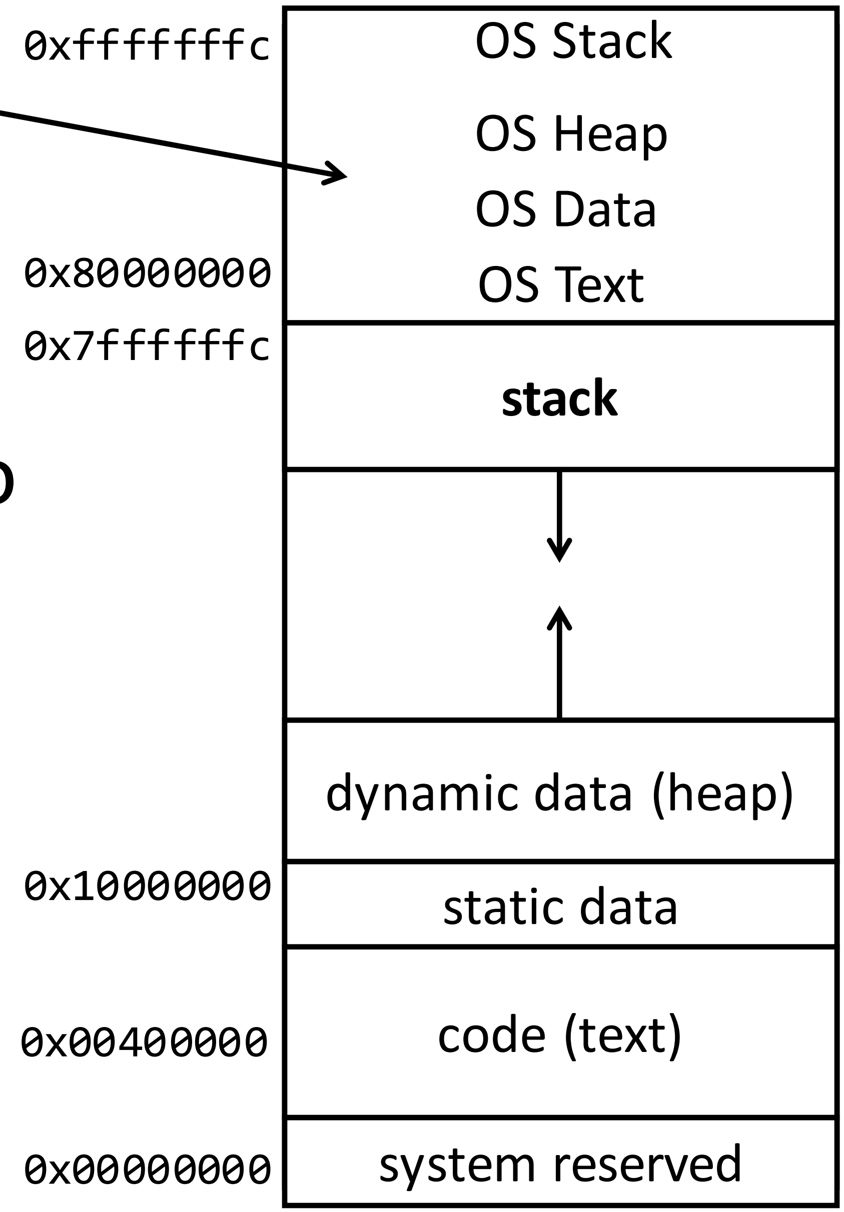
Full System Layout

All kernel text & most data:

- At same virtual address in every address space

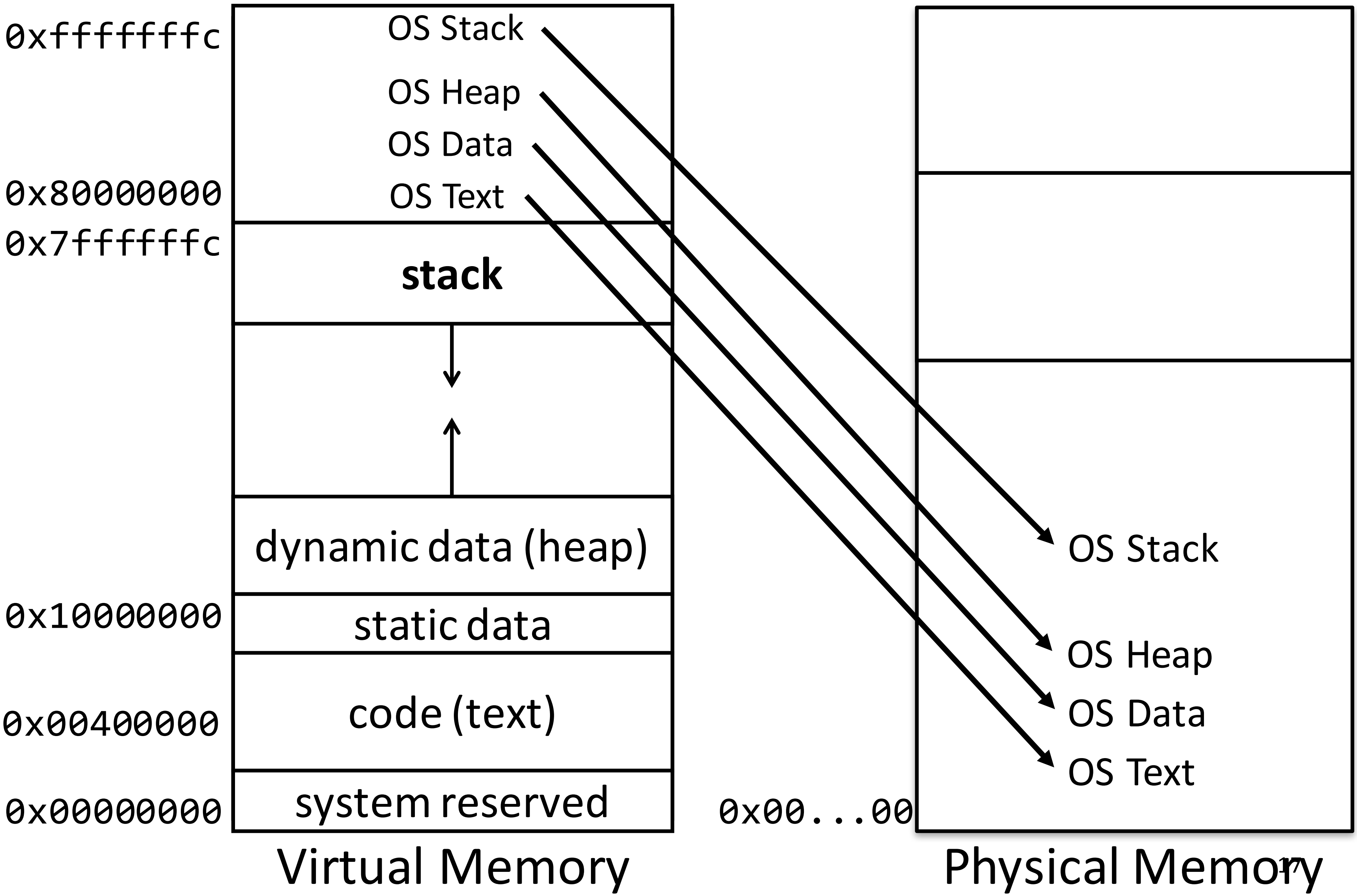
OS is omnipresent, available to help user-level applications

- Typically in high memory

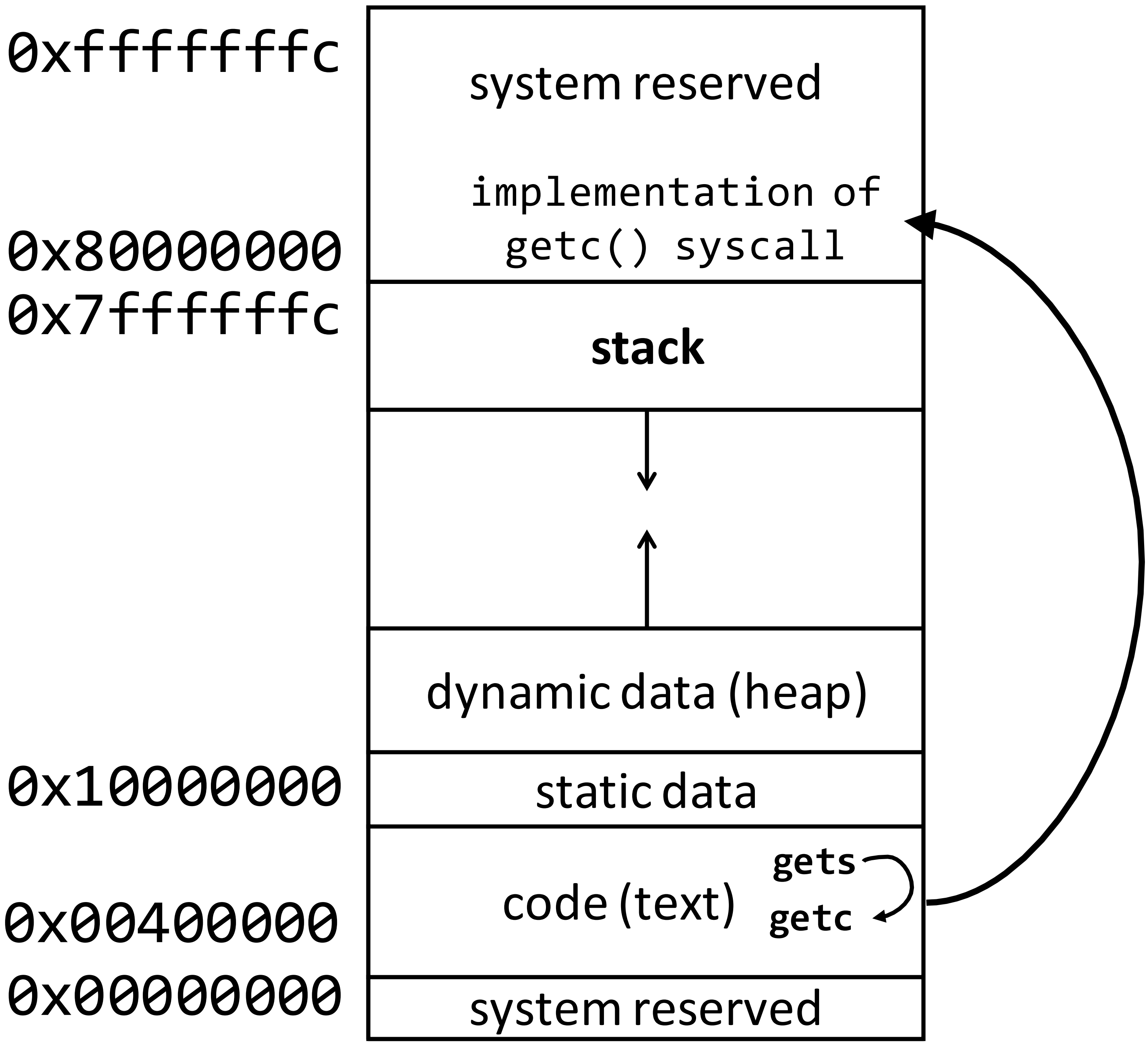


Virtual Memory

Full System Layout



Anatomy of a Process, v2



Clicker Question

Which statement is FALSE?

A) OS manages the CPU, Memory, Devices, and Storage.
B) OS provides a consistent API to be used by other processes.
C) The OS kernel is always present on Disk.
D) The OS kernel is always present in Memory.
E) Any process can fetch and execute OS code in user mode.

Inside the SYSCALL instruction

SYSCALL instruction does an atomic jump to a controlled location (i.e. MIPS 0x8000 0180)

- Switches the sp to the kernel stack
- Saves the old (user) SP value
- Saves the old (user) PC value (= return address)
- Saves the old privilege mode
- Sets the new privilege mode to 1
- Sets the new PC to the kernel syscall handler

Inside the SYSCALL implementation

Kernel system call handler carries out the desired system call

- Saves callee-save registers
- Examines the syscall number
- Checks arguments for sanity
- Performs operation
- Stores result in v0
- Restores callee-save registers
- Performs a “return from syscall” (ERET) instruction, which restores the privilege mode, SP and PC

Exceptional Control Flow

Anything that *isn't* a user program executing its own user-level instructions.

System Calls:

- just one type of exceptional control flow
- Process requesting a service from the OS
- Intentional – *it's in the executable!*

Software Exceptions

Trap
Intentional
Examples:
System call
(OS performs service*)*
Breakpoint traps
Privileged instructions

Fault
Unintentional but
Possibly recoverable
Examples:
Division by zero
Page fault

Abort
Unintentional
Not recoverable
Examples:
Parity error

Hardware support for exceptions

Exception program counter (EPC)

- 32-bit register, holds addr of affected instruction
- Syscall case: Address of SYSCALL

Cause register

- Register to hold the cause of the exception
- Syscall case: 8, Sys

Special instructions to load TLB

- Only do-able by kernel

Precise Exceptions

Hardware guarantees

- Previous instructions complete
- Later instructions are flushed
- EPC and cause register are set
- Jump to prearranged address in OS
- When you come back, restart instruction

- Disable exceptions while responding to one
  - Otherwise can overwrite EPC and cause

Exceptional Control Flow

AKA Exceptions

Hardware interrupts

Asynchronous

= caused by events external to CPU

Maskable

Can be turned off by CPU

Example: alert from network device that a packet just arrived, clock notifying CPU of clock tick

Unmaskable

Cannot be ignored

Example: alert from the power supply that electricity is about to go out

Software exceptions

Synchronous

= caused by CPU executing an instruction

Interrupts & Unanticipated Exceptions

No SYSCALL instruction. Hardware steps in:

- Saves PC of exception instruction (EPC)
- Saves cause of the interrupt/privilege (Cause register)
- Switches the sp to the kernel stack
- Saves the old (user) SP value
- Saves the old (user) PC value
- Saves the old privilege mode
- Sets the new privilege mode to 1
- Sets the new PC to the kernel ~~syscall hander~~ interrupt/exception handler

SYSCALL

Inside Interrupts & Unanticipated Exceptions

interrupt/exception handler handles event
Kernel ~~system call handler carries out system call~~

all

- Saves ~~callee-save~~ registers
- Examines the ~~syscall number~~ cause
- ~~Checks arguments for sanity~~
- Performs operation
- ~~Stores result in v0~~ all
- Restores ~~callee-save~~ registers
- Performs a ERET instruction (restores the privilege mode, SP and PC)

Clicker Question

What other task requires both Hardware and Software?

A) Virtual to Physical Address Translation

B) Branching and Jumping

C) Clearing the contents of a register

D) Pipelining instructions in the CPU

E) What are we even talking about?

Address Translation: HW/SW Division of Labor

Virtual → physical address translation!

Hardware

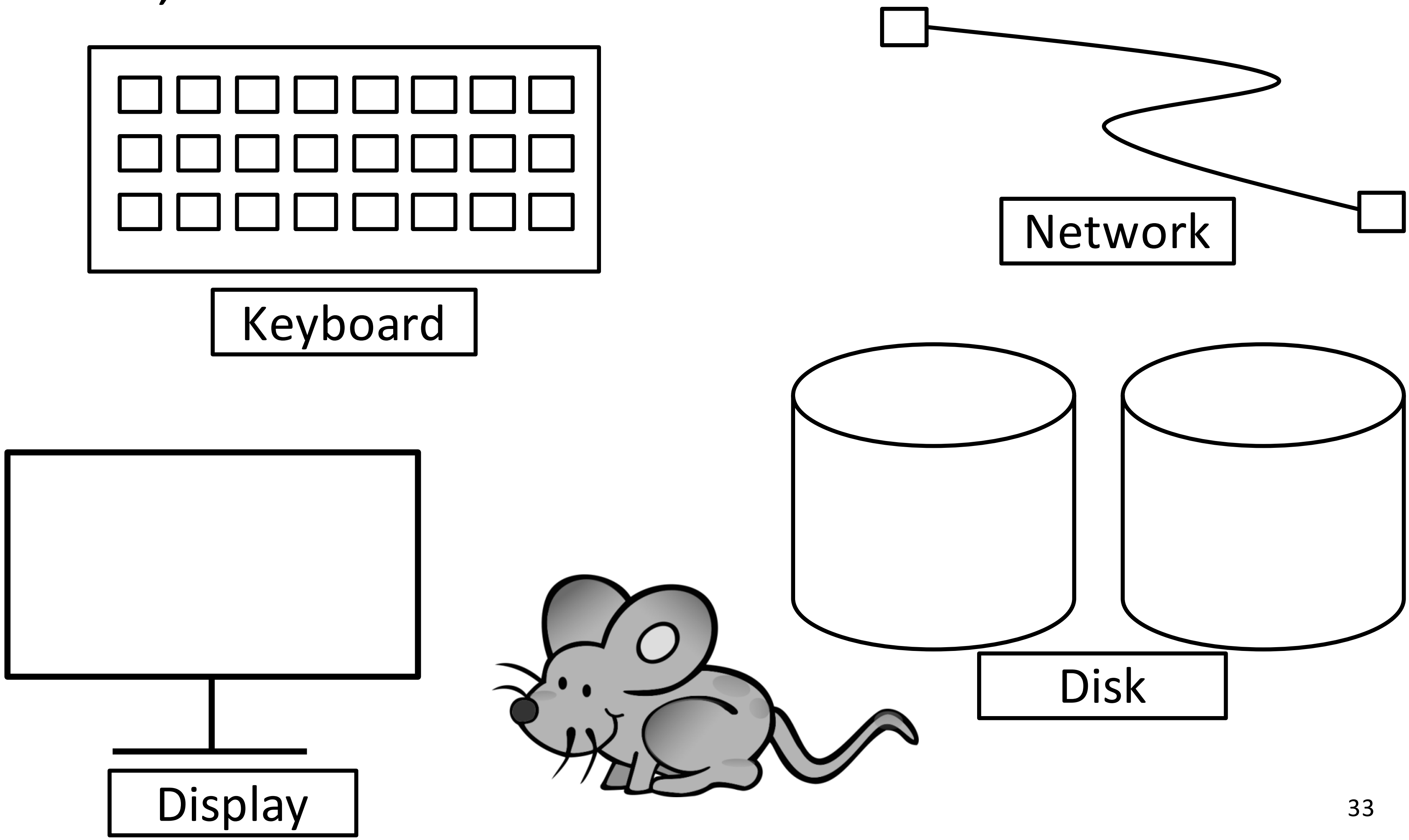
- has a concept of operating in physical or virtual mode
- helps manage the TLB
- raises page faults
- keeps Page Table Base Register (PTBR) and ProcessID

Software/OS

- manages Page Table storage
- handles Page Faults
- updates Dirty and Reference bits in the Page Tables
- keeps TLB valid on context switch:
  - Flush TLB when new process runs (x86)
  - Store process id (MIPS)

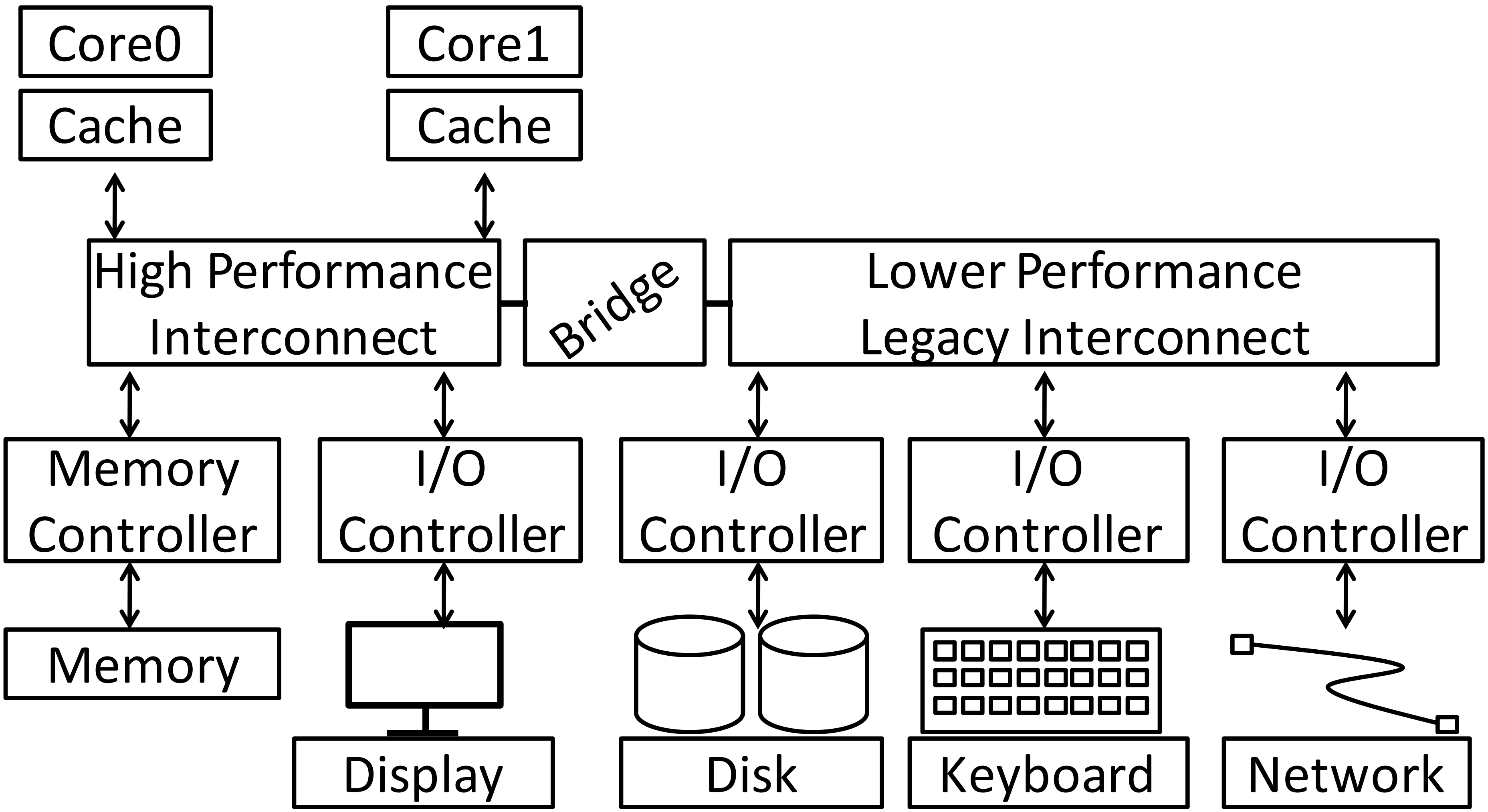
Interacting with the environment

I/O Devices: monitor, disk, keyboard, network, mouse, etc.

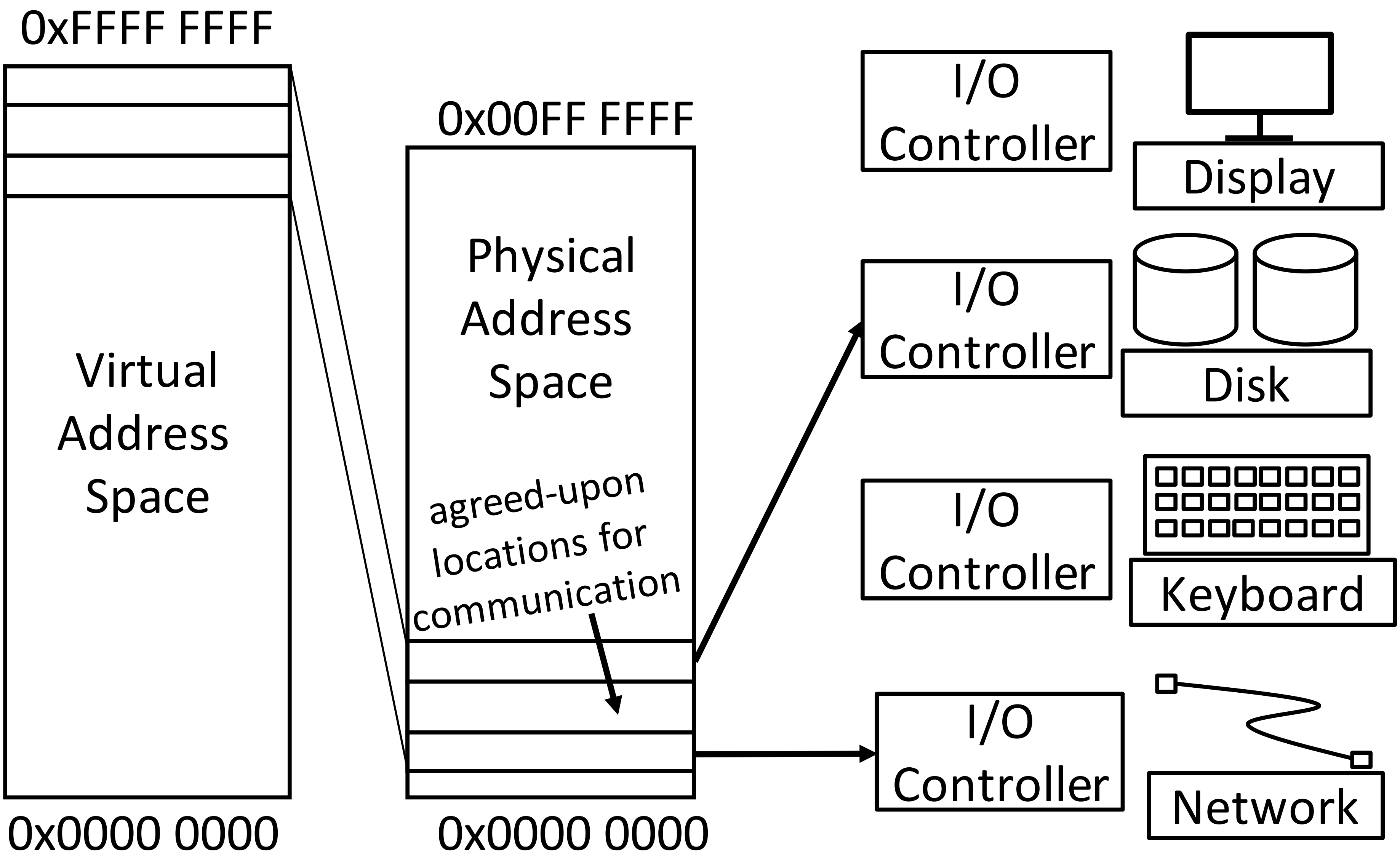


I/O Controllers + Bridge

Modern systems separate high-performance processor, memory, display interconnect from lower-performance interconnect



Aside: Memory-Mapped I/O



Less-favored alternative = Programmed I/O:

- Syscall instructions that communicate with I/O
- Communicate via special device registers

Programmed I/O vs Memory Mapped I/O

Programmed I/O

- Requires special instructions
- Can require dedicated hardware interface to devices
- Protection enforced via kernel mode access to instructions
- Virtualization can be difficult

Memory-Mapped I/O

- Re-uses standard load/store instructions
- Re-uses standard memory hardware interface
- Protection enforced with normal memory protection scheme
- Virtualization enabled with normal memory virtualization scheme

Polling vs. Interrupts

How does program learn device is ready/done?

1. Polling: Periodically check I/O status register
   - Common in small, cheap, or real-time embedded systems
   - \+ Predictable timing, inexpensive
   - – Wastes CPU cycles
2. Interrupts: Device sends interrupt to CPU
   - Cause register identifies the interrupting device
   - Interrupt handler examines device, decides what to do
   - \+ Only interrupt when device ready/done
   - – Forced to save CPU context (PC, SP, registers, *etc.*)
   - – Unpredictable, event arrival depends on other devices' activity

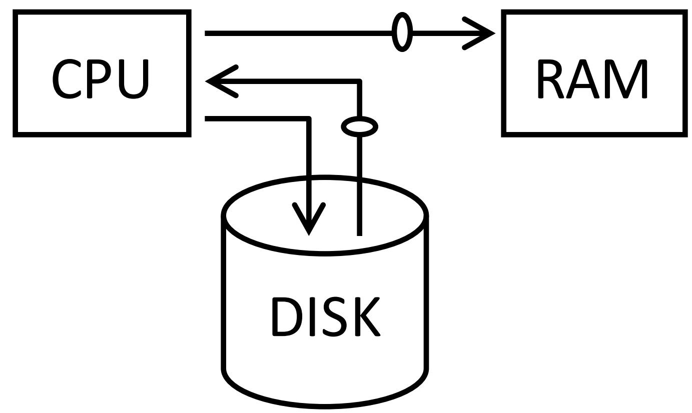
Which one is the winner? Which one is the loser?

Data Transfer

1. Programmed I/O: Device ←→ CPU ←→ RAM

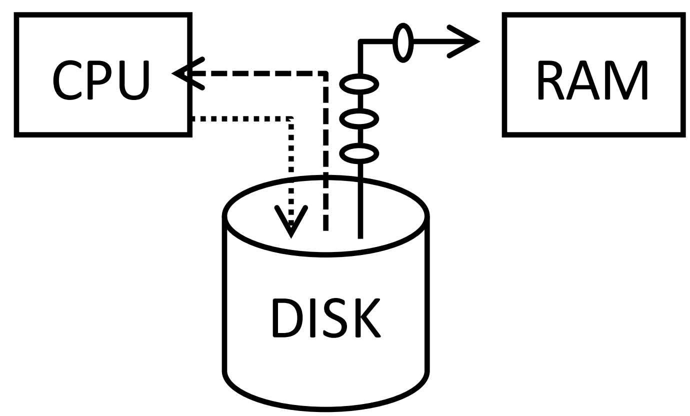
   for (i = 1 .. n)

   - CPU issues read request
   - Device puts data on bus & CPU reads into registers
   - CPU writes data to memory



2. Direct Memory Access (DMA): Device ←→ RAM

   - CPU sets up DMA request
   - for (i = 1 ... n)
     Device puts data on bus
     & RAM accepts it
   - Device interrupts CPU after done



Which one is the winner? Which one is the loser?

I/O Takeaways

Diverse I/O devices require hierarchical interconnect which is more recently transitioning to point-to-point topologies.

Memory-mapped I/O is an elegant technique to read/write device registers with standard load/stores.

Interrupt-based I/O avoids the wasted work in polling-based I/O and is usually more efficient.

Modern systems combine memory-mapped I/O, interrupt-based I/O, and direct-memory access to create sophisticated I/O device subsystems.