

# **Syscalls, exceptions, and interrupts, ...oh my!**

**Hakim Weatherspoon**

**CS 3410**

Computer Science

Cornell University

The slides are the product of many rounds of teaching CS 3410 by Deniz Altinbuken, Professors Weatherspoon, Bala, Bracy, and Sirer.

# Announcements

- C practice assignment
  - Due Monday, April 23rd
- P4-Buffer Overflow is due tomorrow
  - Due Wednesday, April 18th
- P5-Cache Collusion!
  - Due Friday, April 27th

# Outline for Today

- How do we protect processes from one another?
  - Skype should not crash Chrome.
- Operating System
- How do we protect the operating system (OS) from other processes?
  - Chrome should not crash the computer!
- Privileged Mode
- How does the CPU and OS (software) handle exceptional conditions?
  - Division by 0, Page Fault, Syscall, etc.
- Traps, System calls, Exceptions, Interrupts

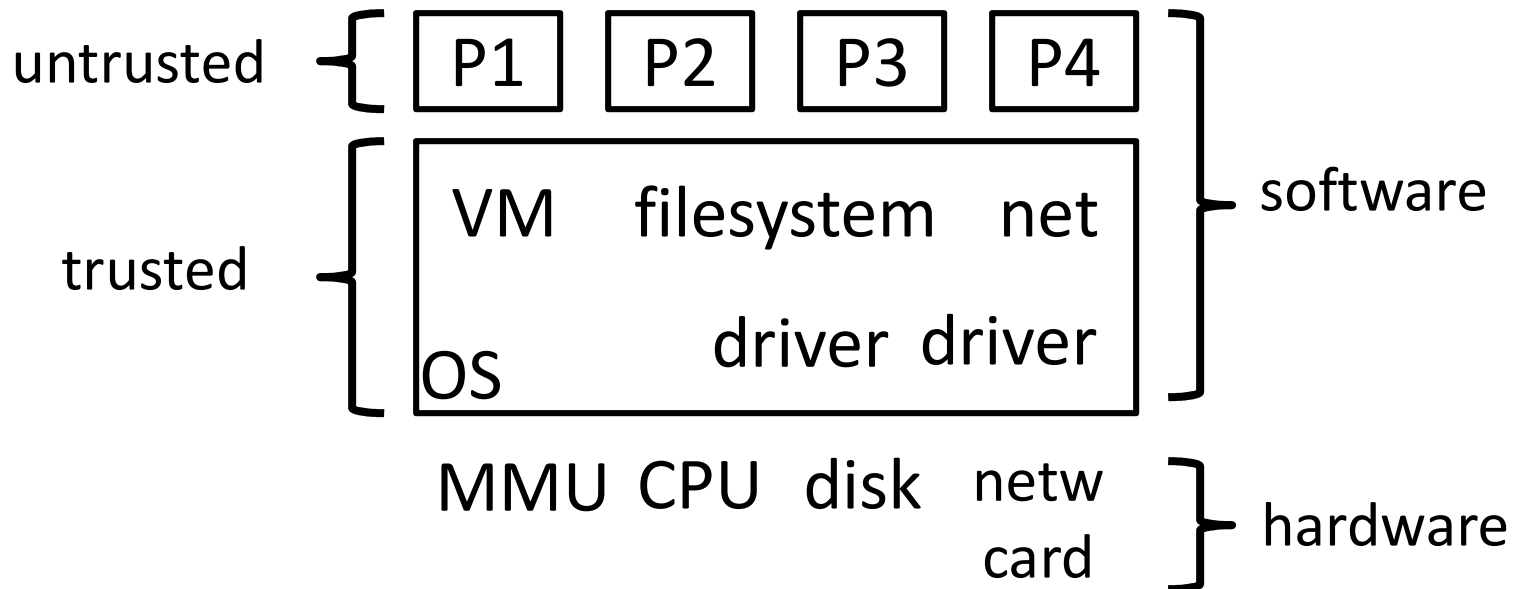
# Operating System

# 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*



# Outline for Today

- How do we protect processes from one another?
  - Skype should not crash Chrome.
- Operating System
- How do we protect the operating system (OS) from other processes?
  - Chrome should not crash the computer!
- Privileged Mode
- How does the CPU and OS (software) handle exceptional conditions?
  - Division by 0, Page Fault, Syscall, etc.
- Traps, System calls, Exceptions, Interrupts

Privileged (Kernel) Mode



# 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
- $\text{Mode} \leftarrow 1$ ;  $\text{PC} \leftarrow$  predetermined address

## 2. OS takes over

- initializes devices, MMU, timers, etc.
- loads programs from disk, sets up page tables, *etc.*
- $\text{Mode} \leftarrow 0$ ;  $\text{PC} \leftarrow$  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?

# 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

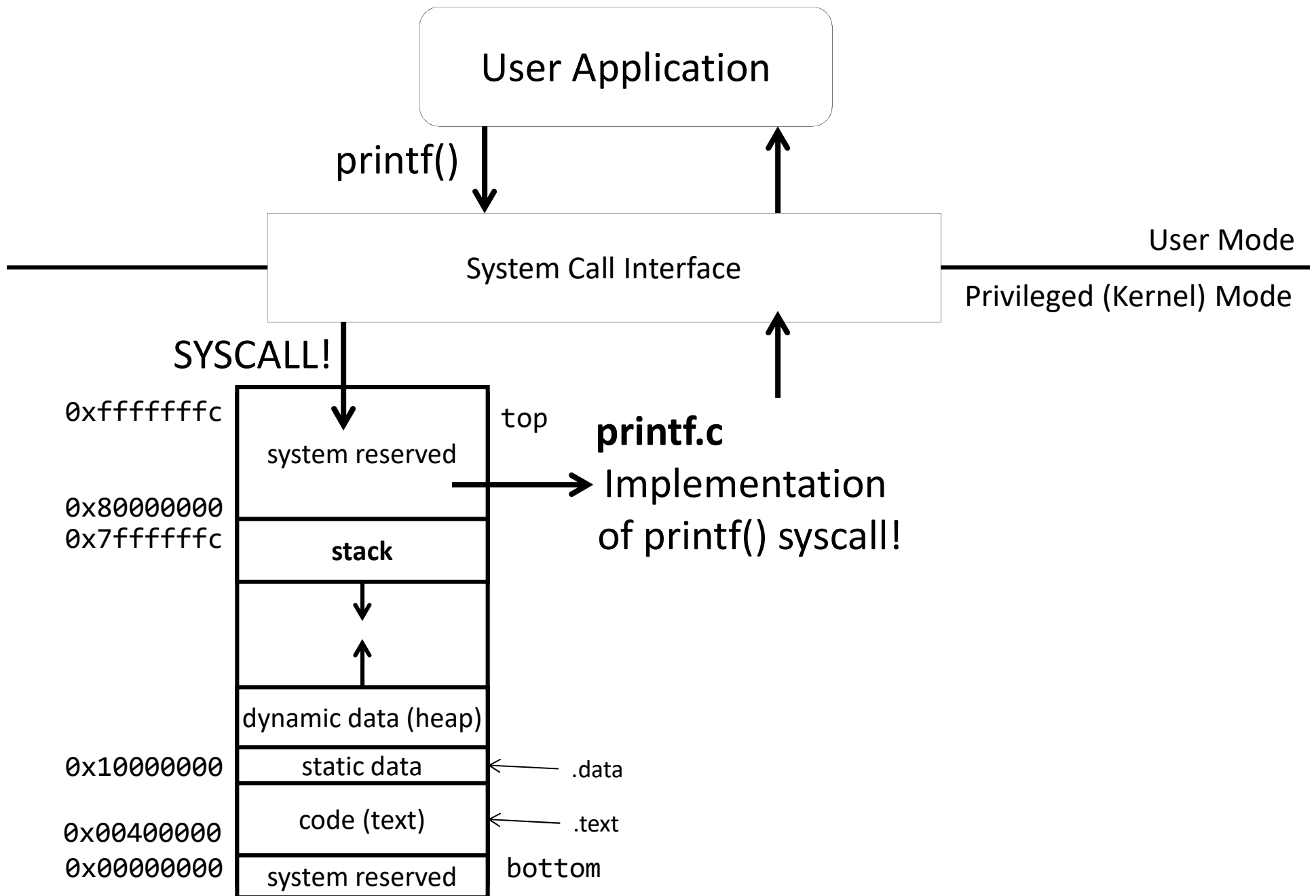
System call: Not just a function call

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

SYSCALL instruction: safe transfer of control to OS

MIPS system call convention:

- Exception handler saves temp regs, saves ra, ...
- but: \$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):

- `getc()` → `syscall`
- `sbrk()` → `syscall`
- `write()` → `syscall`
- `gets()` → `getc()`
- `printf()` → `write()`
- `malloc()` → `sbrk()`
- ...



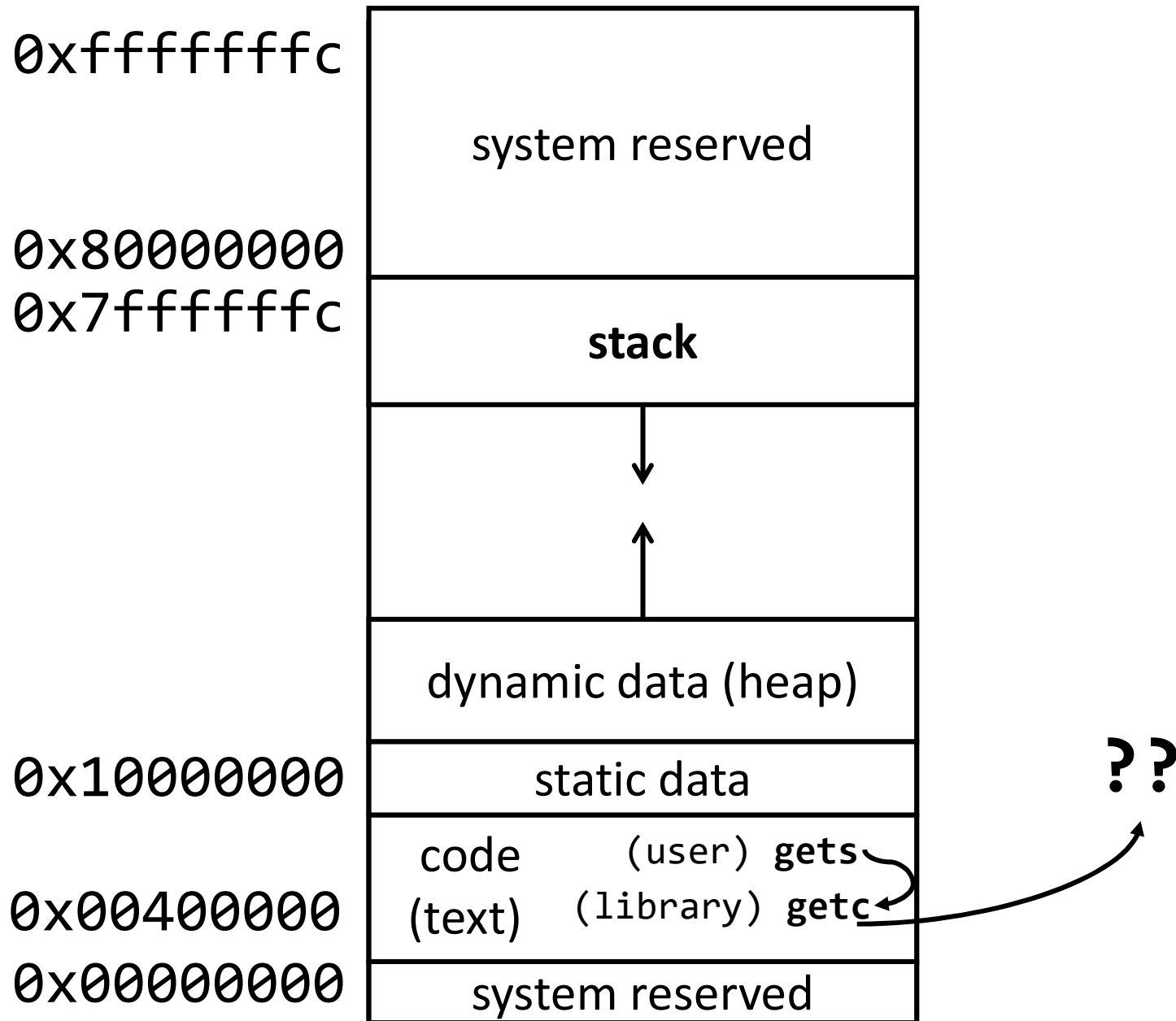
# Invoking System Calls

```
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.*

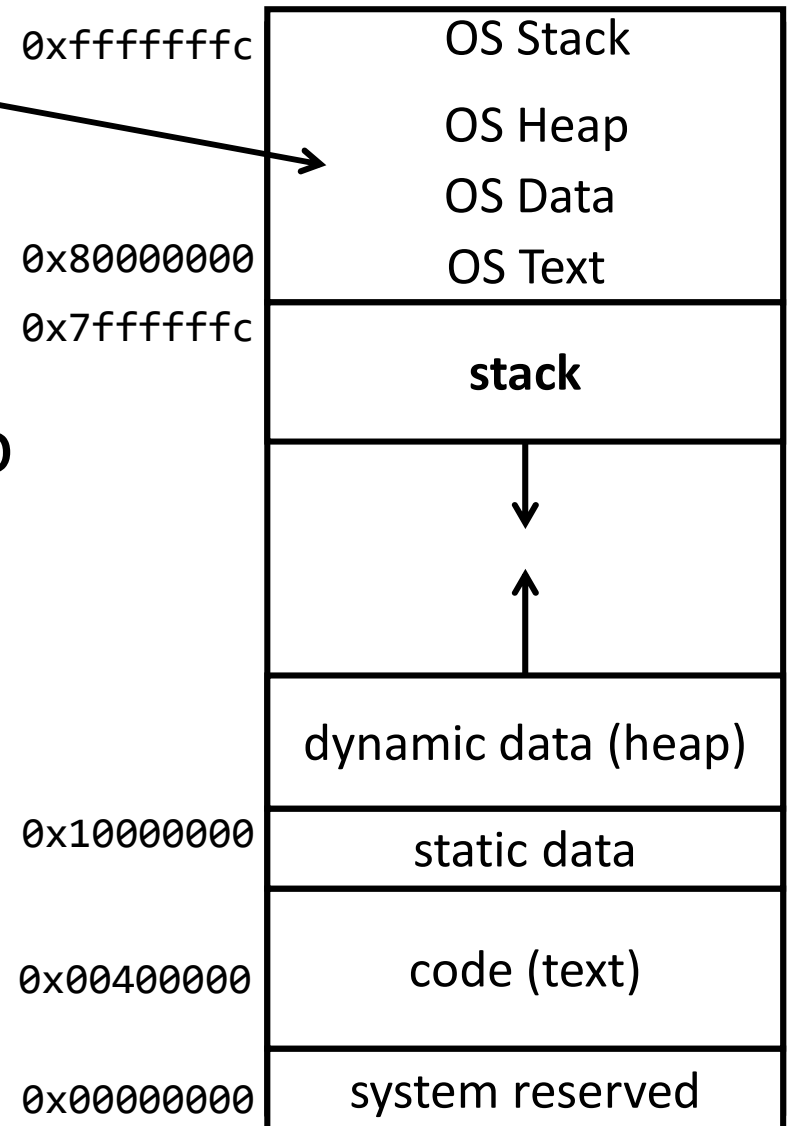
# Full System Layout

All kernel text & most data:

- At same virtual address in every address space

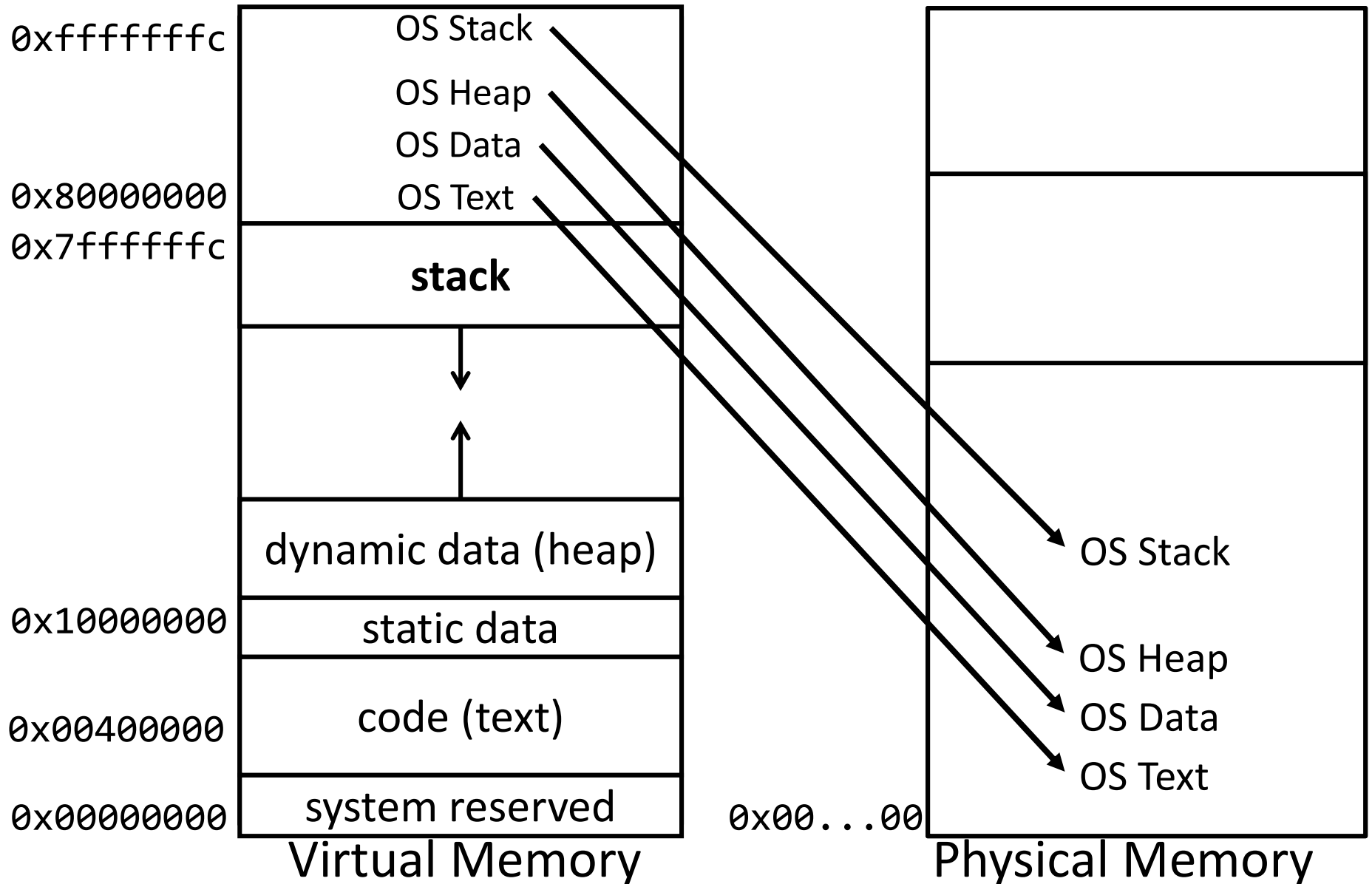
OS is omnipresent, available to help user-level applications

- Typically in high memory

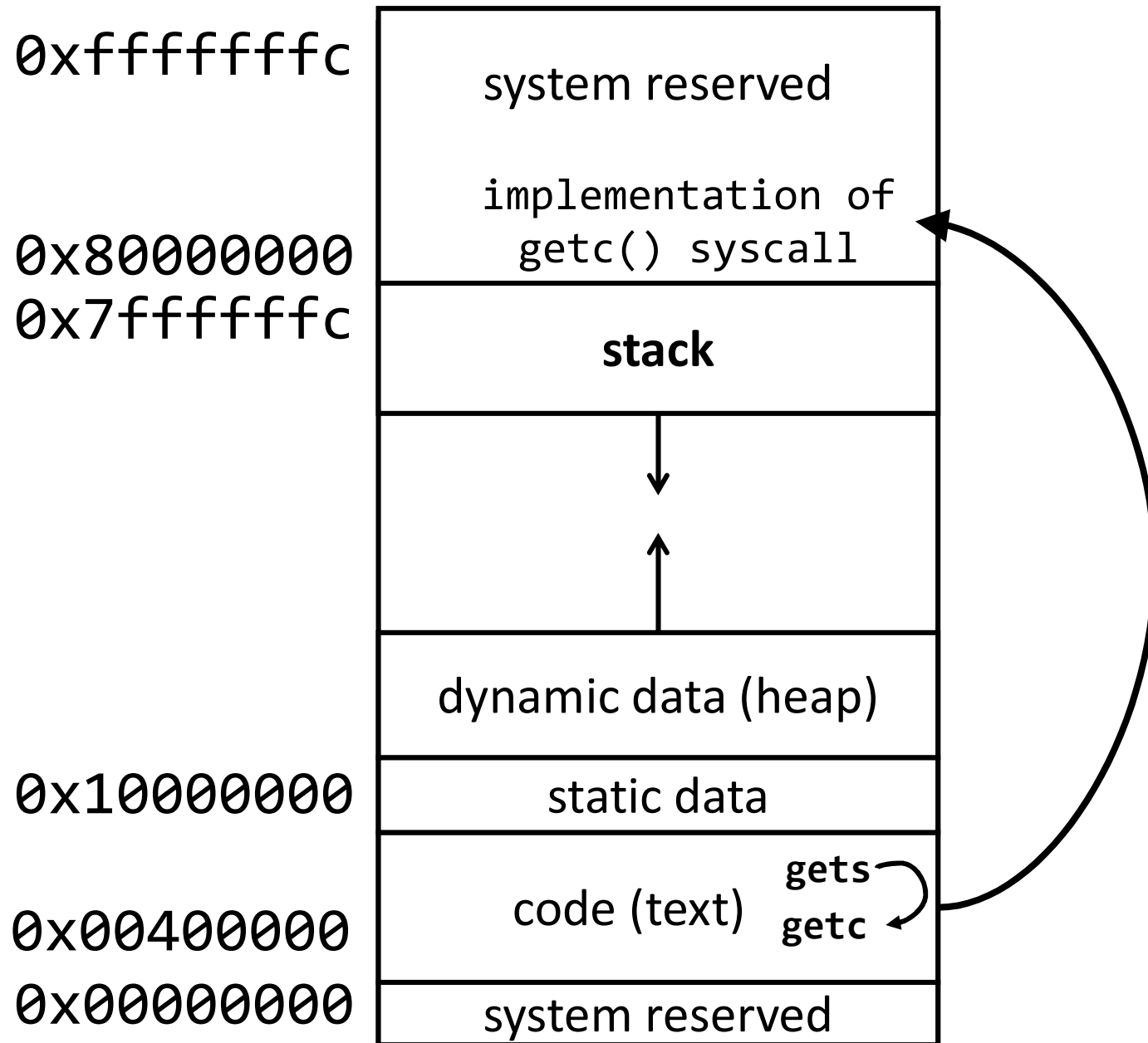


Virtual Memory<sup>20</sup>

# Full System Layout



# Anatomy of a Process, v2



# 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



# Takeaway

- It is necessary to have a privileged (kernel) mode to enable the Operating System (OS):
  - provides isolation between processes
  - protects shared resources
  - provides safe control transfer

# Outline for Today

- How do we protect processes from one another?
  - Skype should not crash Chrome.
- Operating System
- How do we protect the operating system (OS) from other processes?
  - Chrome should not crash the computer!
- Privileged Mode
- How does the CPU and OS (software) handle exceptional conditions?
  - Division by 0, Page Fault, Syscall, etc.
- Traps, System calls, Exceptions, Interrupts

# 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

```
graph TD; A[Software Exceptions] --> B[Trap]; A --> C[Fault]; A --> D[Abort];
```

## **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

*One of **many** ontology / terminology trees.*

# Terminology

Trap: Any kind of a control transfer to the OS

Syscall: Synchronous and planned, process-to-kernel transfer

- SYSCALL instruction in MIPS (various on x86)

Exception: Synchronous but unplanned, process-to-kernel transfer

- exceptional events: div by zero, page fault, page protection err, ...

Interrupt: Asynchronous, device-initiated transfer

- e.g. Network packet arrived, keyboard event, timer ticks

# Hardware support for exceptions

## Exception program counter (EPC)

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

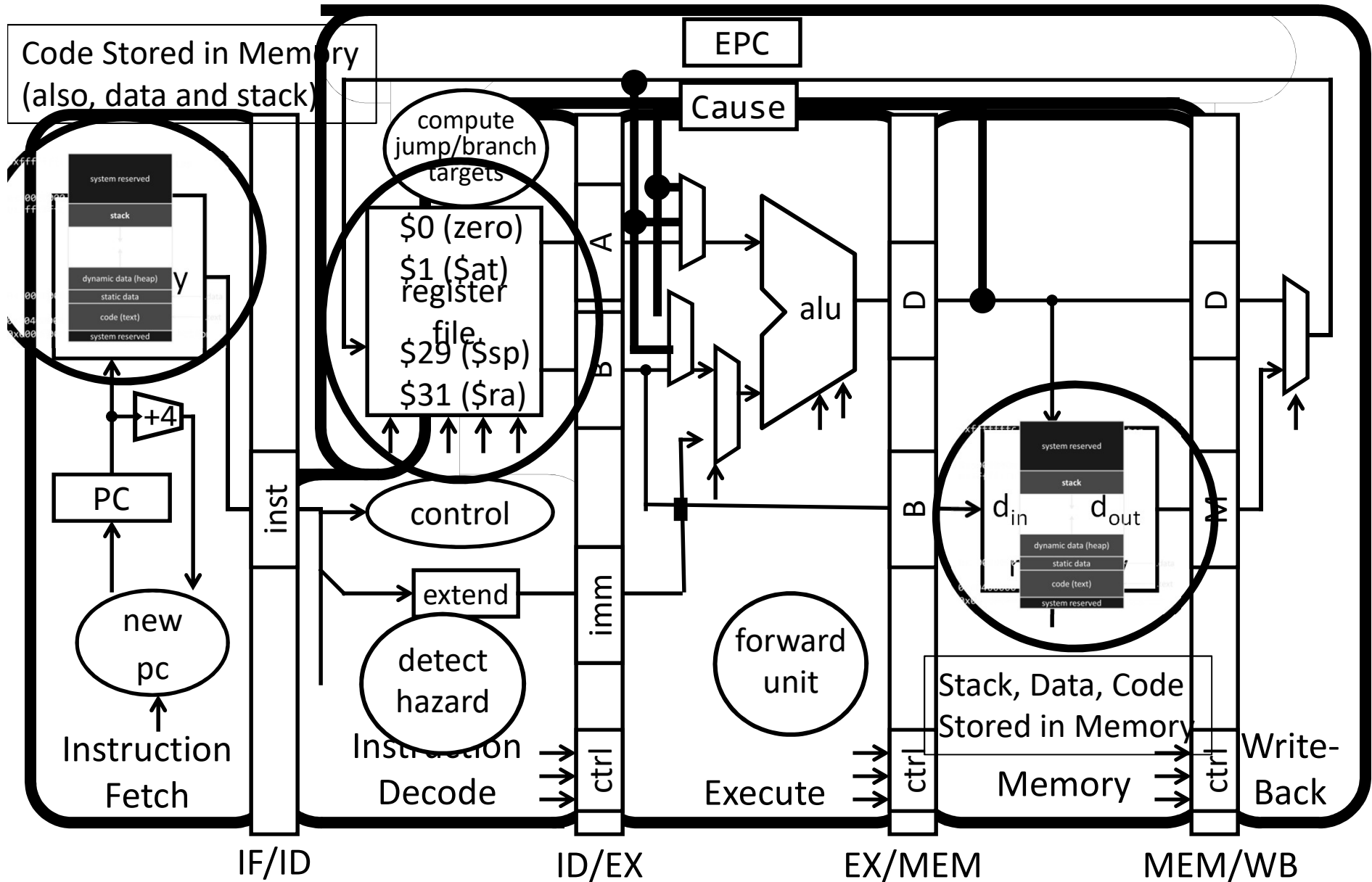
## Cause register

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

## Special instructions to load TLB

- Only do-able by kernel

# Hardware support for exceptions



# Hardware support for exceptions

Precise exceptions: Hardware guarantees

(similar to a branch)

- 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

Software exceptions

*Synchronous*

= caused by CPU  
executing an instruction

Maskable

*Can be turned off by CPU*

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

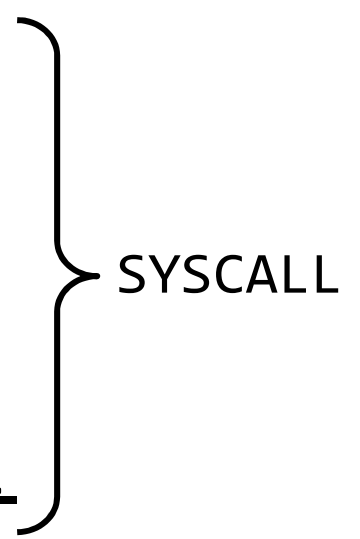
Unmaskable

*Cannot be ignored*

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

# 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 handler~~ 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)

# 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)

# Demand Paging on MIPS

1. TLB miss
2. Trap to kernel
3. Walk Page Table
4. Find page is invalid
5. Convert virtual address to file + offset
6. Allocate page frame
  - Evict page if needed
7. Initiate disk block read into page frame
8. Disk interrupt when DMA complete
9. Mark page as valid
10. Load TLB entry
11. Resume process at faulting instruction
12. Execute instruction