Traps, Exceptions, System Calls, & Privileged Mode

Prof. Kavita Bala and Prof. Hakim Weatherspoon

CS 3410, Spring 2014

Computer Science

Cornell University

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

Heartbleed Security Bug

Heartbleed Security Bug

Heartbleed is a security bug in the open-source OpenSSL cryptography library, widely used to implement the Internet's Transport Layer Security (TLS) protocol.

"…worst vulnerability found since commercial traffic began to flow over the internet." Forbes, "massive Internet Security Vulnerability—Here's where you need to do," Apr 10 2014

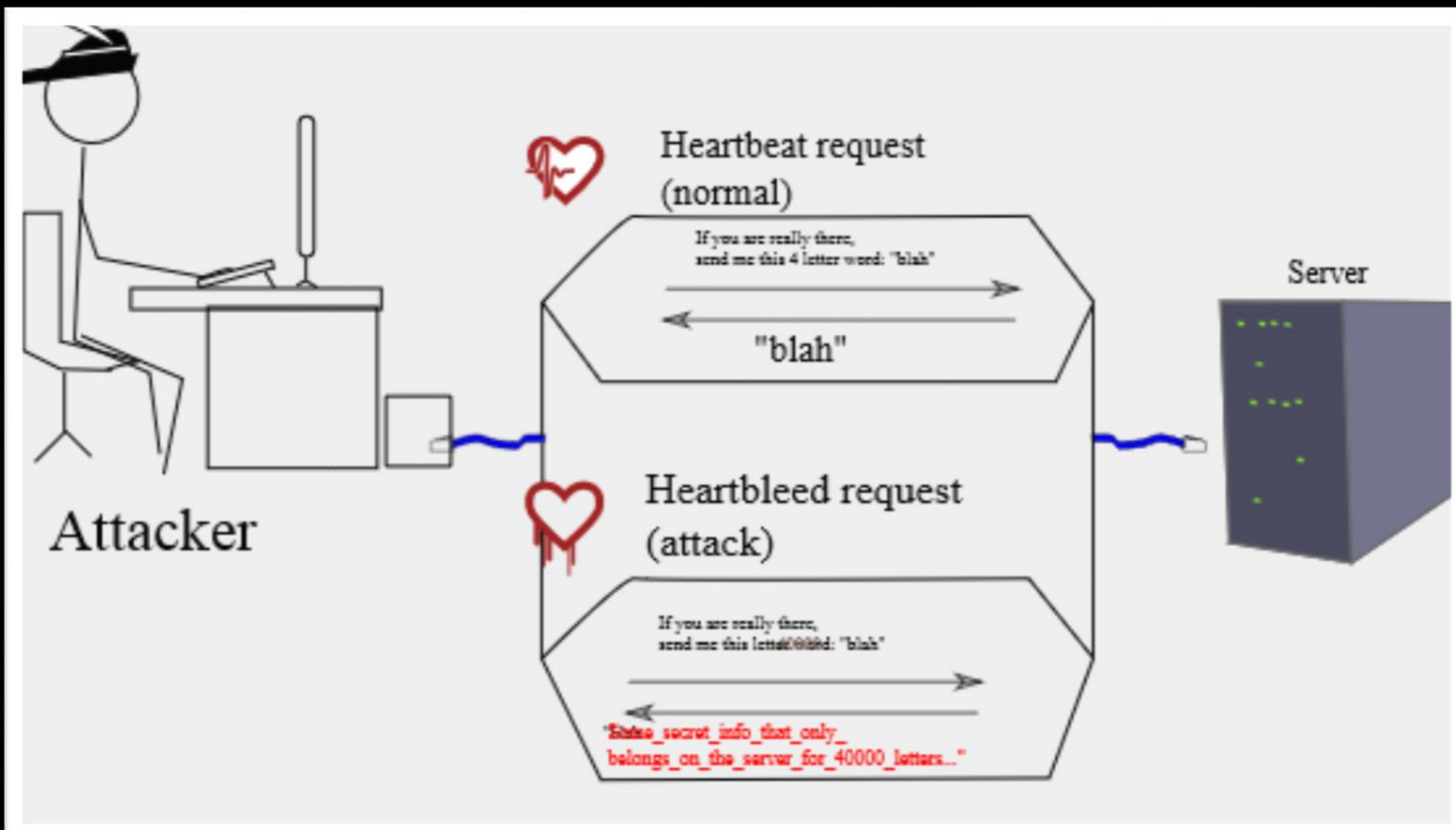
17% (0.5million) secure web servers vulnerable to bug—Netcraft, Ltd, Apr 8, 2014

- Amazon, Akamai, GitHub, Wikipedia, etc

Heartbleed Security Bug

How does it work?

- Lack of bounds checking
- "Buffer over-read"



http://en.wikipedia.org/wiki/Heartbleed

Heartbleed Security Bug

How does it work?

- Lack of bounds checking
- “Buffer over-read”
- SW allows more data to be read than should be allowed
- Malloc/Free did not clear memory
- Req with a large “length” field could return sensitive data
- Unauthenticated user can send a “heartbeat” and receive sensitive data

Heartbleed Security Bug

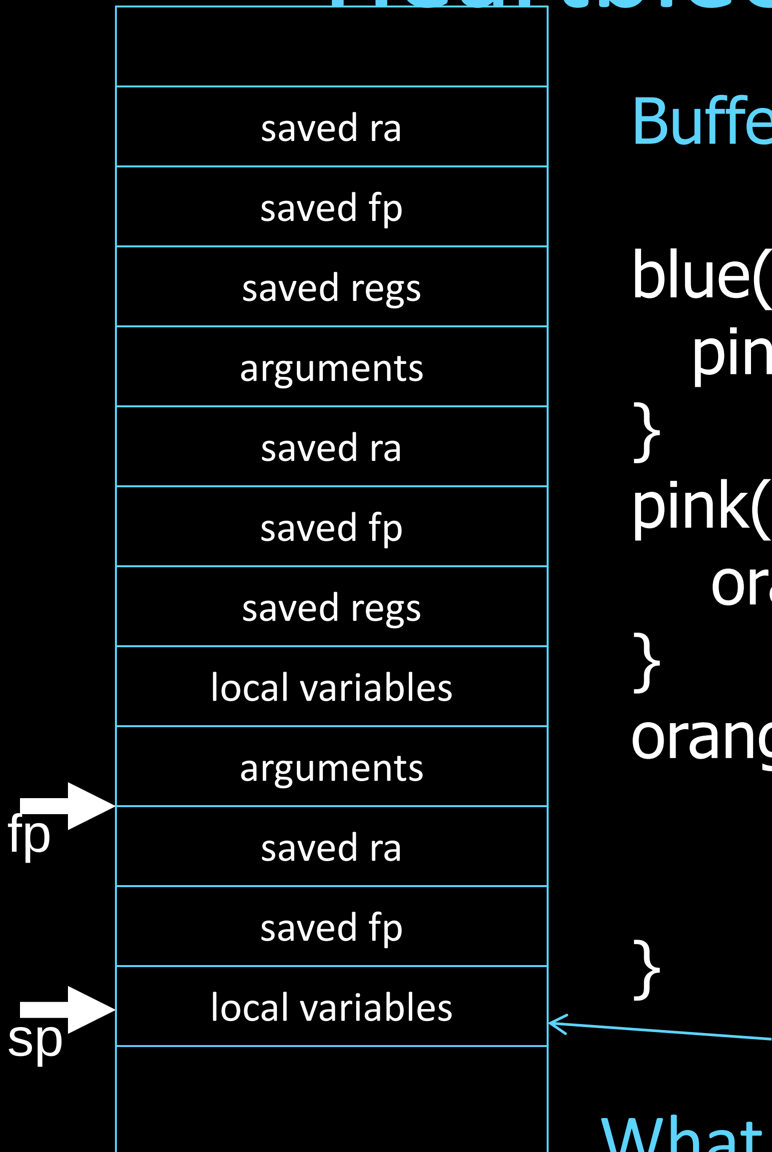
How does it work?

- Lack of bounds checking
- "Buffer over-read"

Similar bug/vulnerability due to "Buffer overflow"

- Lab3
- Browser implementation lacks bounds checking

Heartbleed Security Bug



Buffer Overflow from lec12 and lab3

```
blue() {
   pink(0,1,2,3,4,5);
}
pink(int a, int b, int c, int d, int e, int f) {
    orange(10,11,12,13,14);
}
orange(int a, int b, int c, int, d, int e) {
         char buf[100];
         gets(buf); // read string, no check!
}
```

buf[100]

What happens if more than 100 bytes is written to buf?

Takeaway

Worst Internet security vulnerability found yet due systems practices 101 that we learn in CS3410, lack of bounds checking!

Big Picture

How do we protect programs from one another?
How do we protect the operating system (OS) from programs?

How does the CPU (and software [OS]) handle exceptional conditions. E.g. Div by 0, page fault, syscall, etc?

Goals for Today

Operating System

Privileged mode

Hardware/Software Boundary

Exceptions vs Interrupts vs Traps vs Systems calls

Next Goal

How do we protect programs from one another?

How do we protect the operating system (OS) from programs?

Privileged Mode
aka Kernel Mode

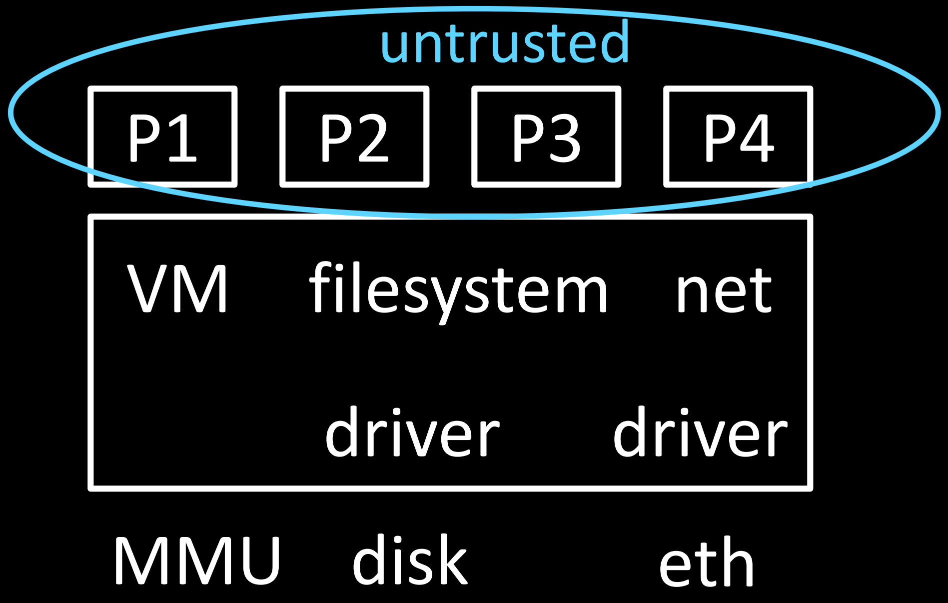
Operating System

Some things not available to untrusted programs:

- MMU instructions, Exception registers, HALT instruction, talk to I/O devices, OS memory, ...

Need trusted mediator: Operating System (OS)

- *Safe control transfer*
- *Data isolation*



Operating System

Trusted mediator is useless without a "privileged mode":

- Any program can muck with TLB, PageTables, OS code…
- A program can intercept exceptions of other programs
- OS can crash if program messes up $sp, $fp, $gp, …

Wrong: Make these instructions and registers available only to "OS Code"

- "OS Code" == any code above 0x80000000
- Program can still JAL into middle of OS functions
- Program can still muck with OS memory, pagetables, …

Privilege Mode

CPU Mode Bit / Privilege Level Status Register

Mode 0 = untrusted = user domain

- "Privileged" instructions and registers are disabled by CPU

Mode 1 = trusted = kernel domain

- All instructions and registers are enabled

Boot sequence:

- load first sector of disk (containing OS code) to well known address in memory
- Mode $\leftarrow$ 1; PC $\leftarrow$ well known address

OS takes over…

- initialize devices, MMU, timers, etc.
- loads programs from disk, sets up pagetables, etc.
- Mode $\leftarrow$ 0; PC $\leftarrow$ program entry point

(note: x86 has 4 levels x 3 dimensions, but only virtual machines uses any the middle)

Terminology

Trap: Any kind of a control transfer to the OS

Syscall: Synchronous (planned), program-to-kernel transfer

- SYSCALL instruction in MIPS (various on x86)

Exception: Synchronous, program-to-kernel transfer

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

Interrupt: Aysnchronous, device-initiated transfer

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

* real mechanisms, but nobody agrees on these terms

Sample System Calls

System call examples:

`putc()`: Print character to screen

- Need to multiplex screen between competing programs

`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 program jump just anywhere in OS code
- OS can’t trust program’s registers (sp, fp, gp, etc.)

SYSCALL instruction: safe transfer of control to OS

- Mode ← 0; Cause ← syscall; PC ← exception vector
- In MIPS, jump to 0x8000 0180 for an exception or 0x8000 0000 a TLB miss

MIPS system call convention:

- user program mostly normal (save temps, save ra, …)
- but: $v0 = system call number, which specifies the operation the application is requesting

Invoking System Calls

```
int getc() {
  asm("addiu $2, $0, 4");
  asm("syscall");
}

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

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()
- …

Where does OS live?

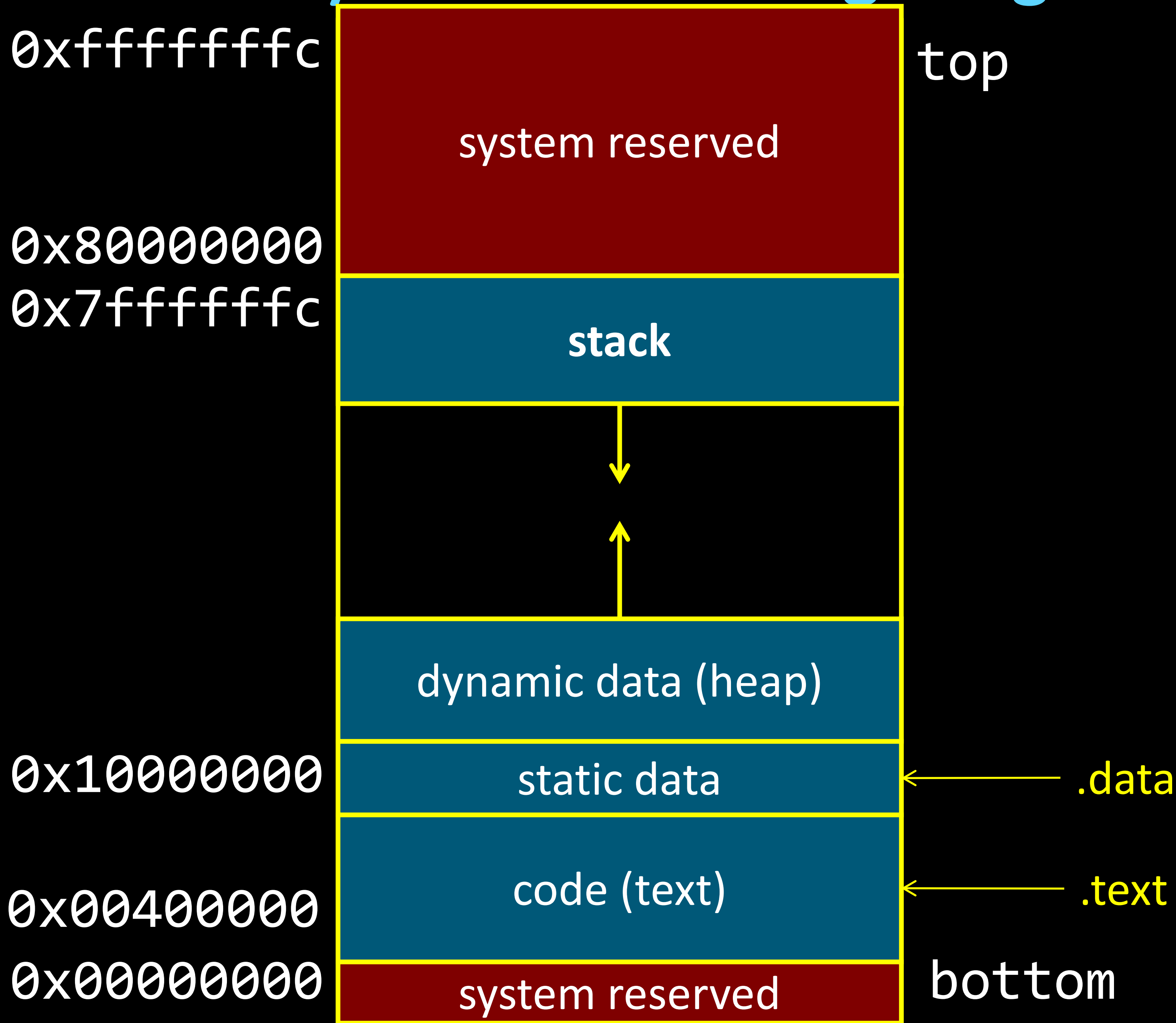
In its own address space?

- But then syscall would have to switch to a different address space
- Also harder to deal with syscall arguments passed as pointers

So in the same address space as process

- Use protection bits to prevent user code from writing kernel
- Higher part of VM, lower part of physical memory

Anatomy of an Executing Program



Full System Layout

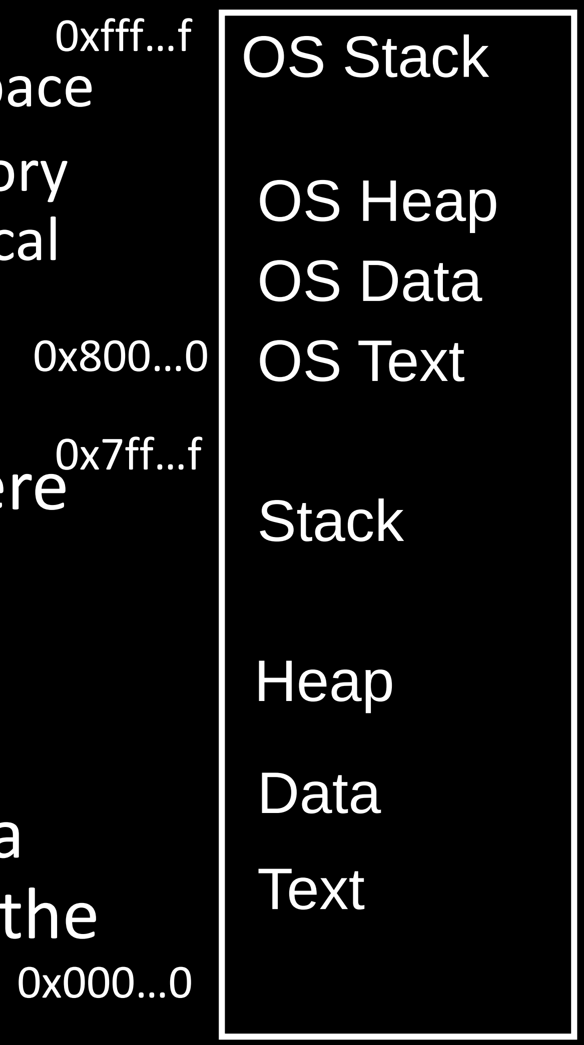
Typically all kernel text, most data

- At same Virtual Addr in every address space
- Map kernel in contiguous physical memory when boot loader puts kernel into physical memory

The OS is omnipresent and steps in where necessary to aid application execution

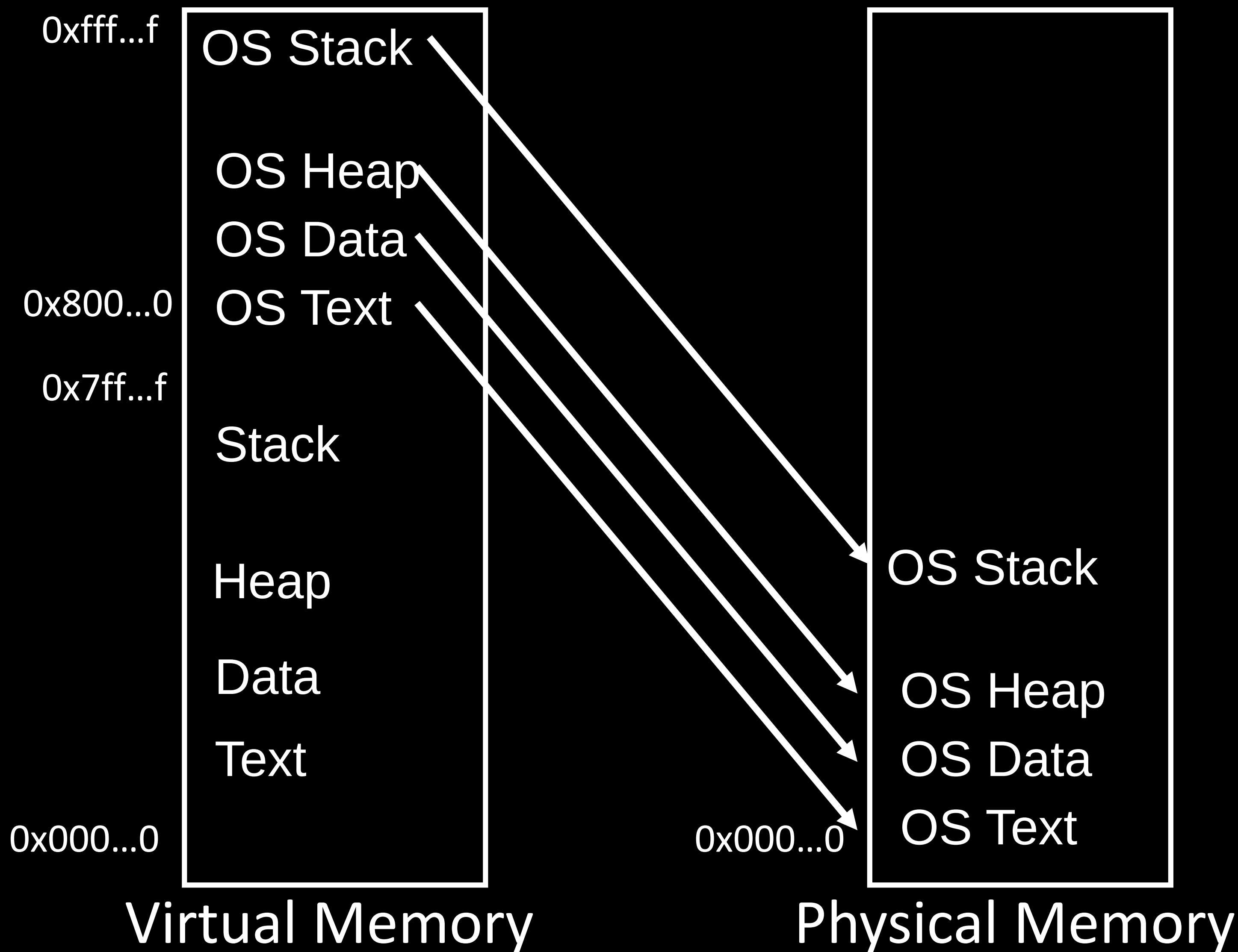
- Typically resides in high memory

When an application needs to perform a privileged operation, it needs to invoke the OS



Virtual Memory

Full System Layout



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

SYSCALL instruction

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

Worst Internet security vulnerability found yet due systems practices 101 that we learn in CS3410, lack of bounds checking!

It is necessary to have a privileged mode (aka kernel mode) where a trusted mediator, the Operating System (OS), provides isolation between programs, protects shared resources, and provides safe control transfer.

Next Goal

How do we protect programs from one another?
How do we protect the operating system (OS) from programs?

How does the CPU (and software [OS]) handle ***exceptional*** conditions? E.g. syscall, Div by 0, page fault, etc?

What are exceptions and how are they handled?

Exceptions

Exceptions are any unexpected change in control flow.

Interrupt -> cause of control flow change external

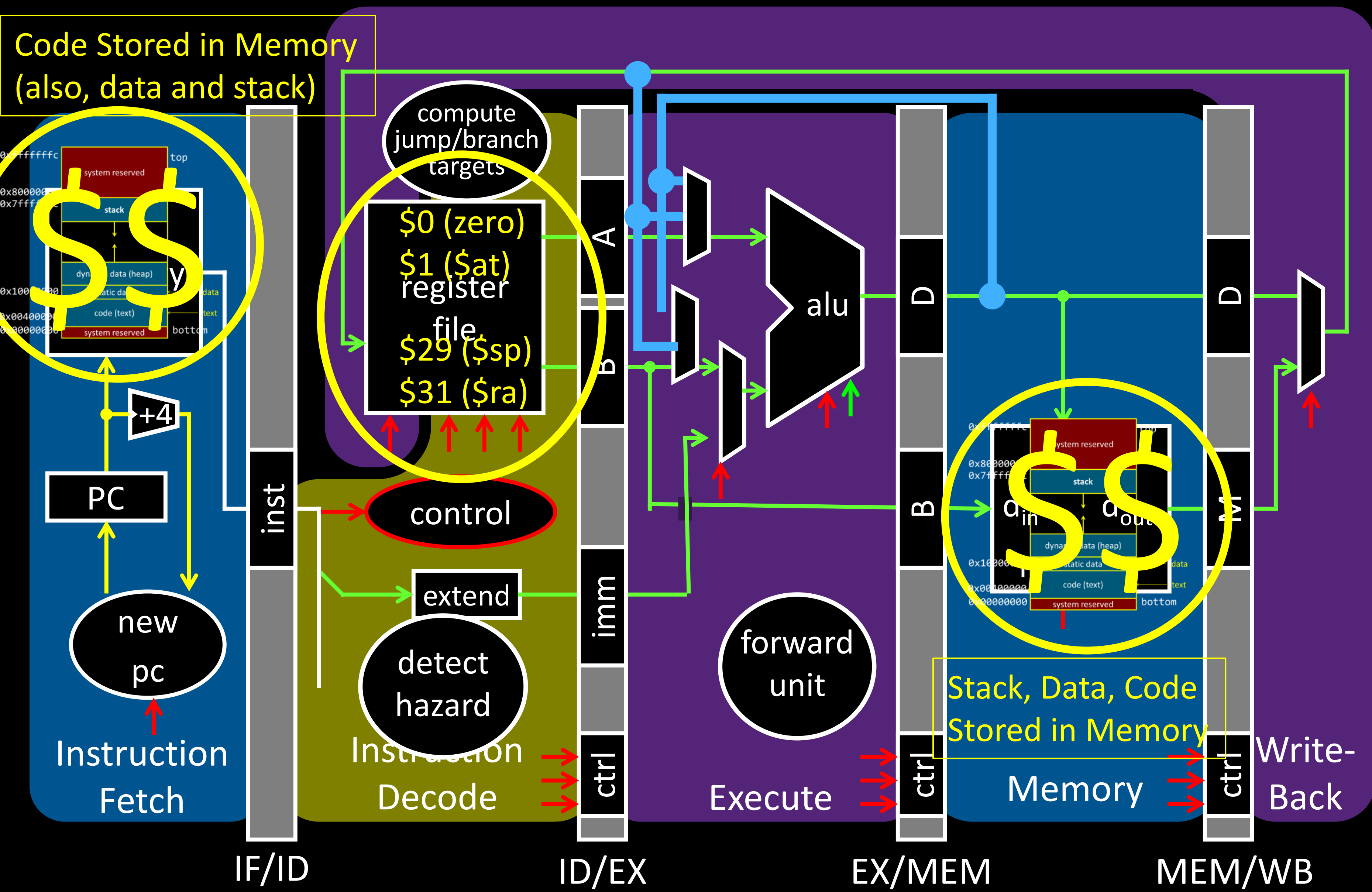
Exception -> cause of control flow change internal

- Exception: Divide by 0, overflow
- Exception: Bad memory address
- Exception: Page fault
- Interrupt: Hardware interrupt (e.g. keyboard stroke)

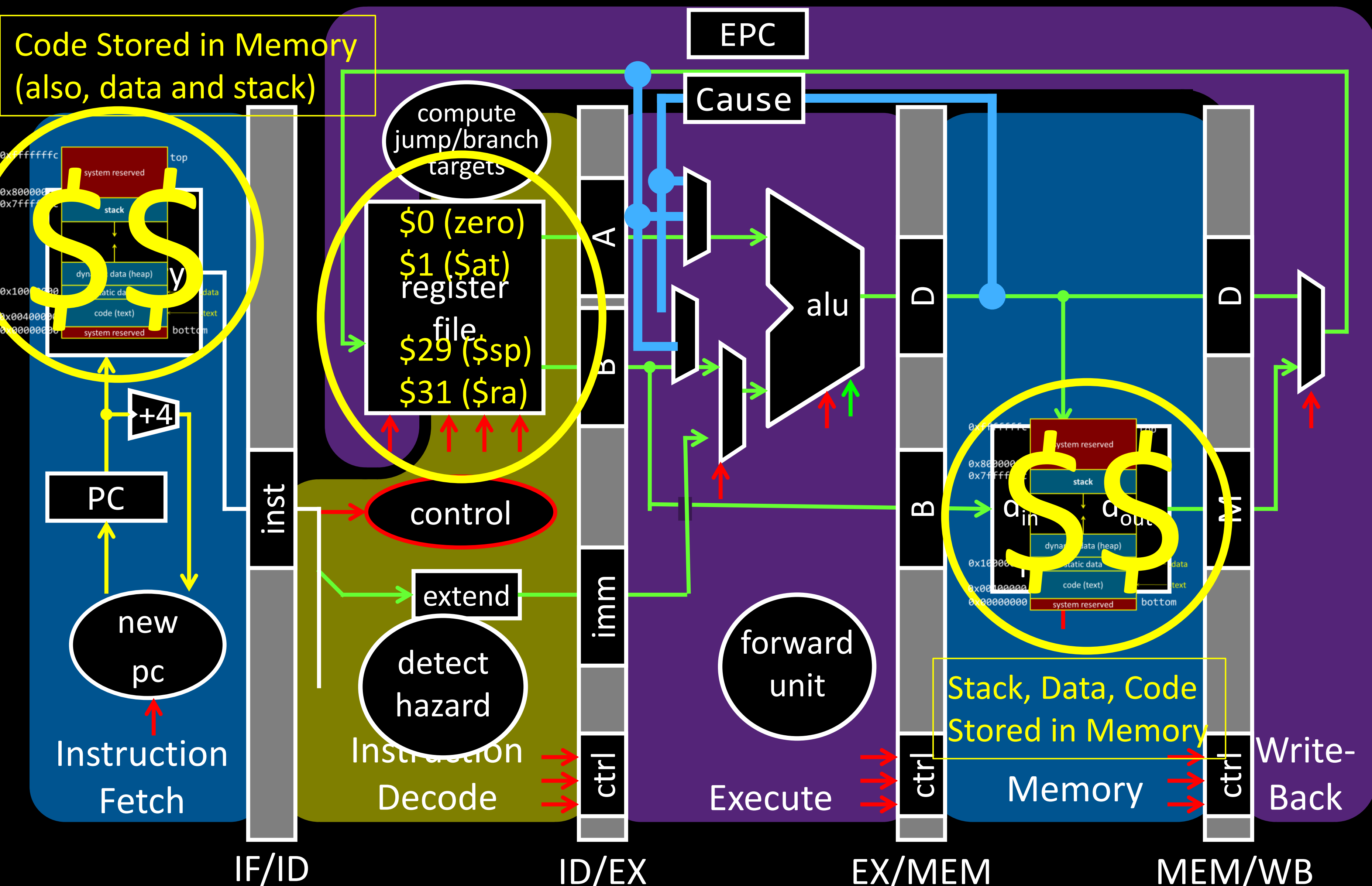
We need ***both*** HW and SW to help resolve exceptions

- Exceptions are at the hardware/software boundary

Exceptions


Code Stored in Memory
(also, data and stack)
$$
compute jump/branch targets
$0 (zero)
$1 ($at)
register file
$29 ($sp)
$31 ($ra)
A
B
imm
ctrl
alu
D
B
M
D
ctrl
ctrl
inst
PC
+4
new pc
control
extend
detect hazard
forward unit
$$
Stack, Data, Code Stored in Memory
Instruction Fetch
Instruction Decode
Execute
Memory
Write-Back
IF/ID
ID/EX
EX/MEM
MEM/WB

Exceptions



Code Stored in Memory
(also, data and stack)
EPC
Cause
compute jump/branch targets
$0 (zero)
$1 ($at)
register file
$29 ($sp)
$31 ($ra)
control
extend
detect hazard
+4
PC
new pc
inst
A
B
imm
ctrl
alu
D
B
M
forward unit
Stack, Data, Code Stored in Memory
Instruction Fetch
Instruction Decode
Execute
Memory
Write-Back
IF/ID
ID/EX
EX/MEM
MEM/WB

Hardware/Software Boundary

Hardware support for exceptions

- Exception program counter (EPC)
- Cause register
- Special instructions to load TLB
  - Only do-able by kernel

Precise and imprecise exceptions

- In pipelined architecture
  - Have to correctly identify PC of exception
  - MIPS and modern processors support this

Hardware/Software Boundary

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

Hardware/Software Boundary

What else requires both HW and SW?

Hardware/Software Boundary

Virtual to physical address translation is assisted by hardware

Need *both* hardware and software support

Software

- Page table storage, fault detection and updating
  - Page faults result in interrupts that are then handled by the OS
  - Must update appropriately Dirty and Reference bits (e.g., ~LRU) in the Page Tables

Hardware/Software Boundary

OS has to keep TLB valid

Keep TLB valid on context switch

- Flush TLB when new process runs (x86)
- Store process id (MIPs)

Also, store pids with cache to avoid flushing cache on context switches

Hardware support

- Page table register
- Process id register

Takeaway

Worst Internet security vulnerability found yet due systems practices 101 that we learn in CS3410, lack of bounds checking!

It is necessary to have a privileged mode (aka kernel mode) where a trusted mediator, the Operating System (OS), provides isolation between programs, protects shared resources, and provides safe control transfer.

Exceptions are any unexpected change in control flow. Precise exceptions are necessary to identify the exceptional instructional, cause of exception, and where to start to continue execution.

We need help of both hardware and software (e.g. OS) to resolve exceptions. Finally, we need some type of protected mode to prevent programs from modifying OS or other programs.

Next Goal

What is the difference between traps, exceptions, interrupts, and system calls?

Recap: Traps

→ Map kernel into every process using *supervisor* PTEs

→ Switch to kernel mode on trap, user mode on return

Trap: Any kind of a control transfer to the OS

Syscall: Synchronous, program-to-kernel transfer

- user does caller-saves, invokes kernel via syscall
- kernel handles request, puts result in v0, and returns

Exception: Synchronous, program-to-kernel transfer

- user div/load/store/… faults, CPU invokes kernel
- kernel saves everything, handles fault, restores, and returns

Interrupt: Aysnchronous, device-initiated transfer

- e.g. Network packet arrived, keyboard event, timer ticks
- kernel saves everything, handles event, restores, and returns

Interrupts & Exceptions

On an interrupt or exception

- CPU saves PC of exception instruction (EPC)
- CPU 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 interrupt/exception handler

Interrupts & Exceptions

Kernel interrupt/exception handler handles the event

- Saves all registers
- Examines the cause
- Performs operation required
- Restores all registers
- Performs a "return from interrupt" instruction, which restores the privilege mode, SP and PC

Example: Clock Interrupt

Example: Clock Interrupt*

- Every N cycles, CPU causes exception with Cause = CLOCK_TICK
- OS can select N to get e.g. 1000 TICKs per second

.ktext 0x8000 0180

(step 1) save *everything* but $k0, $k1 to 0xB0000000

(step 2) set up a usable OS context

(step 3) examine Cause register, take action

if (Cause == PAGE_FAULT) handle_pfault(BadVaddr)

else if (Cause == SYSCALL) dispatch_syscall($v0)

else if (Cause == CLOCK_TICK) schedule()

(step 4) restore registers and return to where program left off

* not the CPU clock, but a programmable timer clock

Scheduler

```
struct regs context[];
int ptbr[];
schedule() {
    i = current_process;
    j = pick_some_process();
    if (i != j) {
        current_process = j;
        memcpy(context[i], 0xB0000000);
        memcpy(0xB0000000, context[j]);
        asm(“mtc0 Context, ptbr[j]”);
  }
}
```

Syscall vs. Interrupt

Syscall vs. Exceptions vs. Interrupts

Same mechanisms, but…

Syscall saves and restores much less state

Others save and restore full processor state

Interrupt arrival is unrelated to user code

Takeaway

It is necessary to have a privileged mode (aka kernel mode) where a trusted mediator, the Operating System (OS), provides isolation between programs, protects shared resources, and provides safe control transfer.

Exceptions are any unexpected change in control flow. Precise exceptions are necessary to identify the exceptional instructional, cause of exception, and where to start to continue execution.

We need help of both hardware and software (e.g. OS) to resolve exceptions. Finally, we need some type of protected mode to prevent programs from modifying OS or other programs.

To handle any exception or interrupt, OS analyzes the Cause register to vector into the appropriate exception handler. The OS kernel then handles the exception, and returns control to the same process, killing the current process, or possibly scheduling another process.

Summary

Trap

- Any kind of a control transfer to the OS

Syscall

- Synchronous, program-initiated control transfer from user to the OS to obtain service from the OS
- e.g. SYSCALL

Exception

- Synchronous, program-initiated control transfer from user to the OS in response to an exceptional event
- e.g. Divide by zero, TLB miss, Page fault

Interrupt

- Asynchronous, device-initiated control transfer from user to the OS
- e.g. Network packet, I/O complete

Administrivia

Lab3 due tomorrow, Wednesday

- Take Home Lab, finish within day or two of your Lab
- Work ***alone***

HW2 Help Session on tonight, Tuesday, April 15th, at 7:30pm in Kimball B11 and Thursday, April 17th.

Administrivia

Next five weeks

- Week 11 (Apr 15): Proj3 release, Lab3 due Wed, HW2 due Sat
- Week 12 (Apr 22): Lab4 release and Proj3 due Fri
- Week 13 (Apr 29): Proj4 release, Lab4 due Tue, Prelim2
- Week 14 (May 6): Proj3 tournament Mon, Proj4 design doc due

Final Project for class

- Week 15 (May 13): Proj4 due Wed