Traps, Exceptions, System Calls, & Privileged Mode

Deniz ALTINBUKEN

CS 3410, Spring 2015

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 disclosed in April 2014 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."

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

Amazon, Akamai, GitHub, Wikipedia, etc. affected!

Heartbleed Security Bug

How does it work?

-
- “Buffer over-read”

Send me this 5 letter word if you are alive: “Phone”

Phone

Send me this 1000 letter word if you are alive: “Phone”

Phone **Alice passwd 123456 Bob passwd 654321 Server passwd …**

Heartbleed Security Bug

Heartbeat Protocol:

- Client sends buffer and the length of buffer
- Server writes buffer in memory
- Returns length of characters from memory

-
- Malloc/Free did not clear memory
- Unauthenticated users can send a heartbeat

Server can return sensitive information present in memory!

Takeaway

- Worst Internet security vulnerability found

yet due systems practices 101 that we learn in CS3410, lack of bounds checking!

- Lab 3: Similar bug/vulnerability due to “Buffer overflow”
  - Browser implementation lacks bounds checking
  - Overwrite return address in memory using a smart input!

Outline for Today

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

Outline for Today

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

Operating System

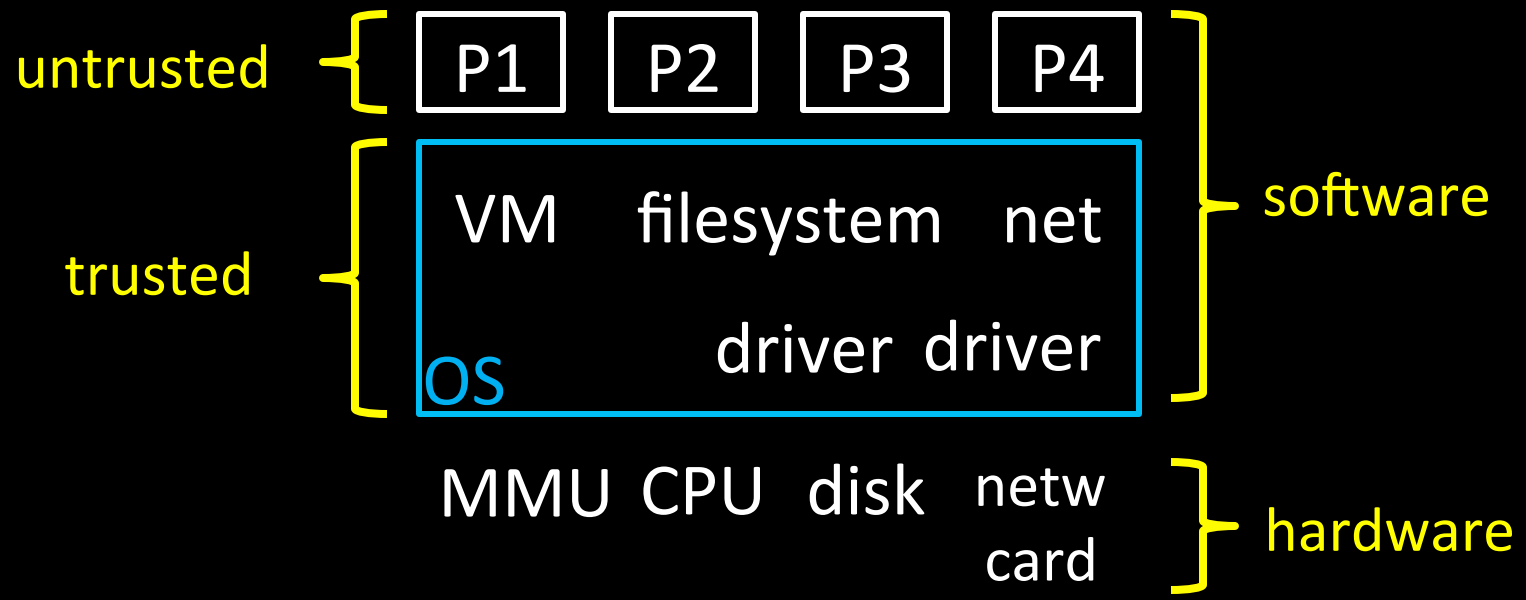
Operating System

-
  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

-
  - *Safe control transfer between processes*
  - *Isolation (memory, registers) of processes*



Which statement is FALSE?

A) OS is always in the Hard Disk.

B) OS is always in Memory.

C) All processes can access the OS code.

D) OS provides a consistent API to be used by other processes.

E) OS manages the CPU, Memory, Devices, and Storage.

Outline for Today

- 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!
- How does the CPU and OS (software) handle exceptional conditions?
  - Division by 0, Page Fault, Syscall, etc.

Privileged (Kernel) Mode

Privileged Mode

- Only privileged (and trusted!) processes can access & change important things.
  - Editing TLB, Page Tables, OS code, $sp, $fp…
- If an untrusted process could change $sp, $fp, and $gp, OS would crash!

Privileged Mode

How can we get the privileged mode to work?

Attempt #1:

- Make privileged instructions and registers available only to “OS Code”!
  - “OS Code”→ address

Privileged Mode

Would making privileged instructions and registers available only to “OS Code” work?

A) Will work great!

B) Will work, but performance will be slow!

C) Will not work because any process can jump into OS code

D) Will not work because process can access all registers

E) Whatever, I am bored!

Privileged Mode

How can we get the privileged mode to work?

Attempt #1:

- Make privileged instructions and registers available only to "OS Code"
  - "OS Code"→ resides in memory at preset virtual address

Does not work:

- Process can still JAL into middle of OS functions
- Process can still access and change memory, page tables, …

Privileged Mode

How can we get the privileged mode to work?

Attempt #2:

CPU Mode Bit in Privilege Level Status Register

Mode 0 = untrusted = user mode

- "Privileged" instructions and registers are disabled by CPU

Mode 1 = trusted = kernel mode

- All instructions and registers are enabled

Privileged Mode

How can we get the privileged mode to work?

Boot sequence:

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

OS takes over…

- initializes devices, MMU, timers, etc.
- loads programs from disk, sets up page tables, etc.
- Mode ← 0; PC ← program entry point

Privileged Mode

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 Calls

System Calls

System call: Not just a function call

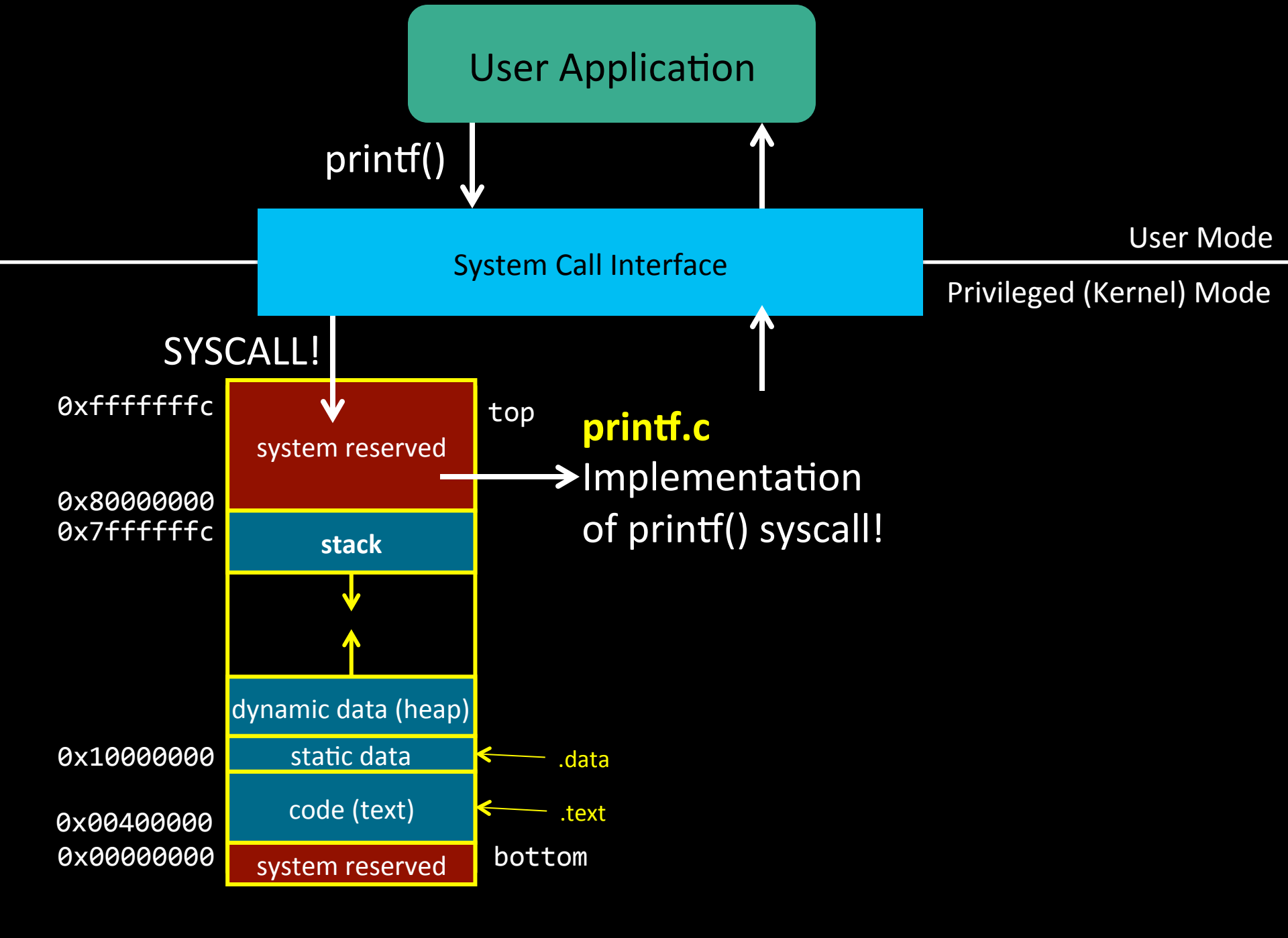
- Don't let process ram jump just anywhere in OS code
- OS can't trust process' registers (sp, fp, gp, etc.)

SYSCALL instruction: safe transfer of control to OS

- Mode ← ← syscall; PC ← *exception* vector

MIPS system call convention:

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


User Application
printf()
System Call Interface
User Mode
Privileged (Kernel) Mode
SYSCALL!
0xffffffc
top
system reserved
printf.c
Implementation
of printf() syscall!
0x80000000
0x7ffffffc
stack
dynamic data (heap)
0x10000000
static data
.data
code (text)
.text
0x00400000
0x00000000
system reserved
bottom

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

Invoking System Calls

```
int getc() {
  asm("addiu $v0, $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 the OS live?

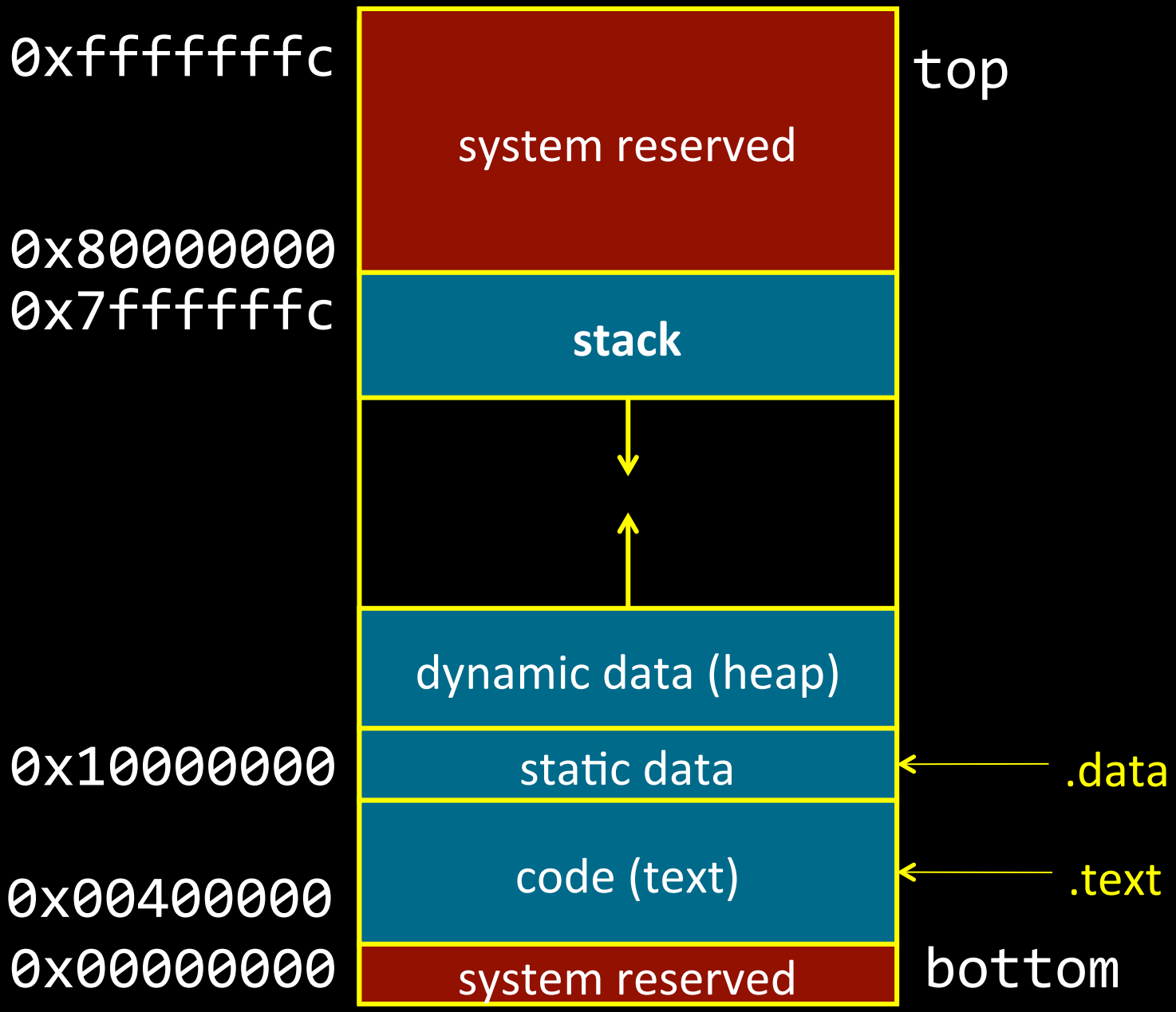
In its own address space?

- 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 virtual memory, lower part of physical memory

Anatomy of a Process



Full System Layout

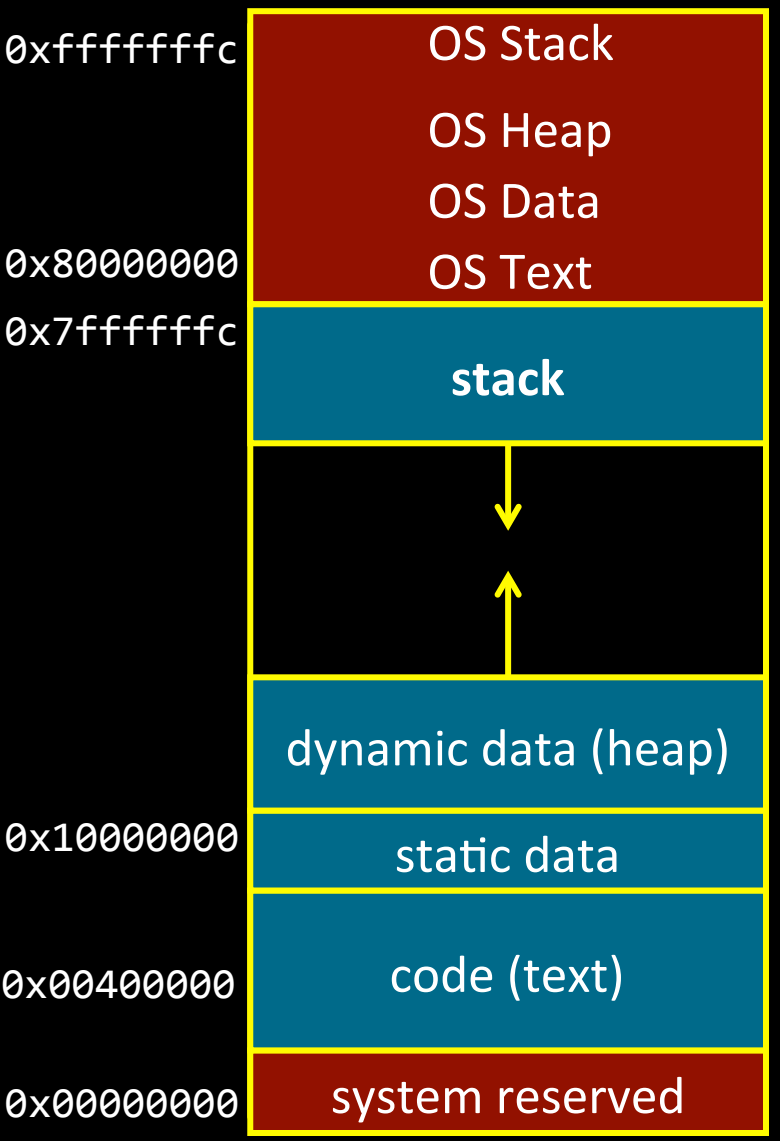
Typically all kernel text, most data

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

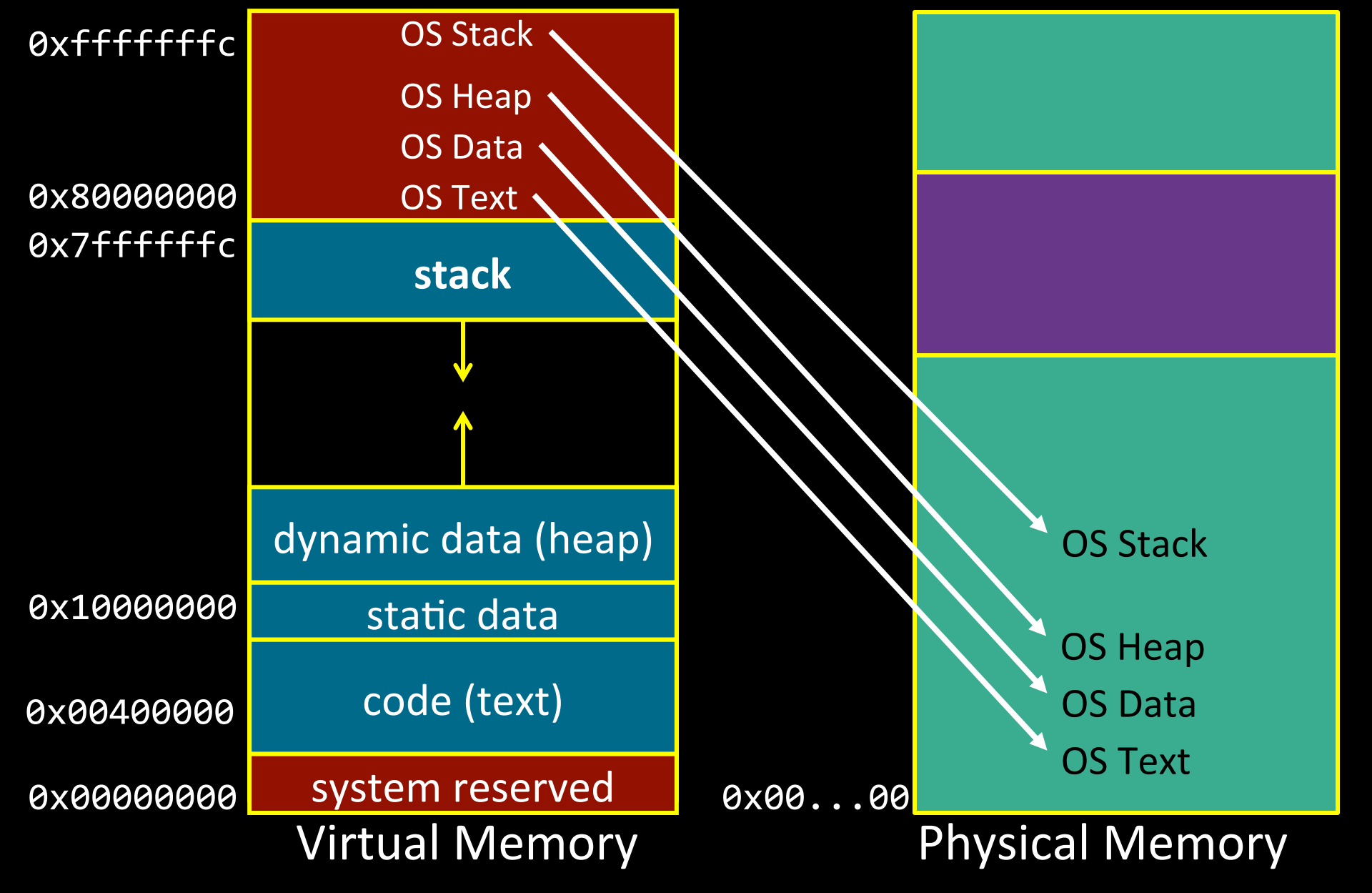
The OS is omnipresent and steps in where necessary to aid application execution

- Typically resides in high memory

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



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

- (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

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

Exceptions

Exceptions
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: I/O interrupt (e.g. keyboard stroke)

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

- Exceptions are at the hardware/software boundary

Hardware/Software Boundary

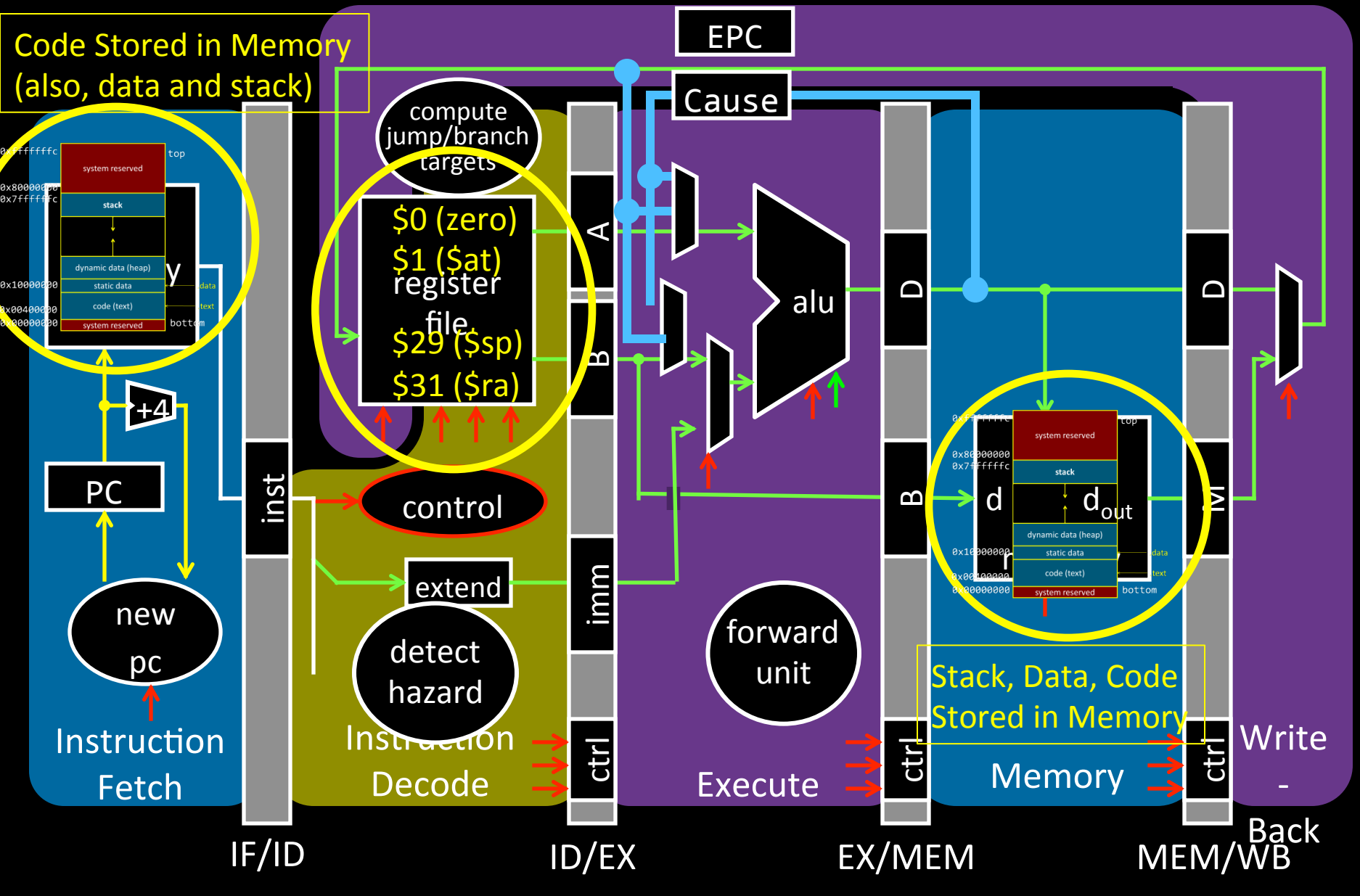
Hardware support for exceptions

- Exception program counter (EPC)
  - A 32-bit register to hold the addr of the affected instruction.
  - Syscall case: Address of SYSCALL
- Cause register
  - A register to hold the cause of the exception.
  - Syscall case: 8, Sys
- 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

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)
A
B
imm
alu
D
B
M
D
+4
PC
new
pc
inst
control
extend
detect
hazard
forward
unit
ctrl
ctrl
ctrl
d
d out
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

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 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?

Hardware/Software Boundary

Virtual to physical address translation!

Hardware

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

Software

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

Summary

Trap

- Any kind of a control transfer to the OS

Syscall

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

Exception

- Synchronous, process-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

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

Interrupts & Exceptions

Hardware

- 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

Software

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 process 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. Exception vs. Interrupt

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

- Traps are any transfer of control to the OS.
- *Exceptions are any unexpected change in control flow*.
- Precise exceptions are necessary to identify the exceptional instructions, cause of exception, and where to continue execution.
- We need help of both hardware and software (e.g. OS) to resolve exceptions.

Takeaway

To handle any exception or interrupt:

- OS analyzes the Cause register
- OS vectors into the appropriate exception handler.
- OS kernel handles the exception
- Returns control to the same process
  - Possibly kills the current process
  - Possibly schedules another process