Traps, Exceptions, System Calls, & Privileged Mode

Hakim Weatherspoon

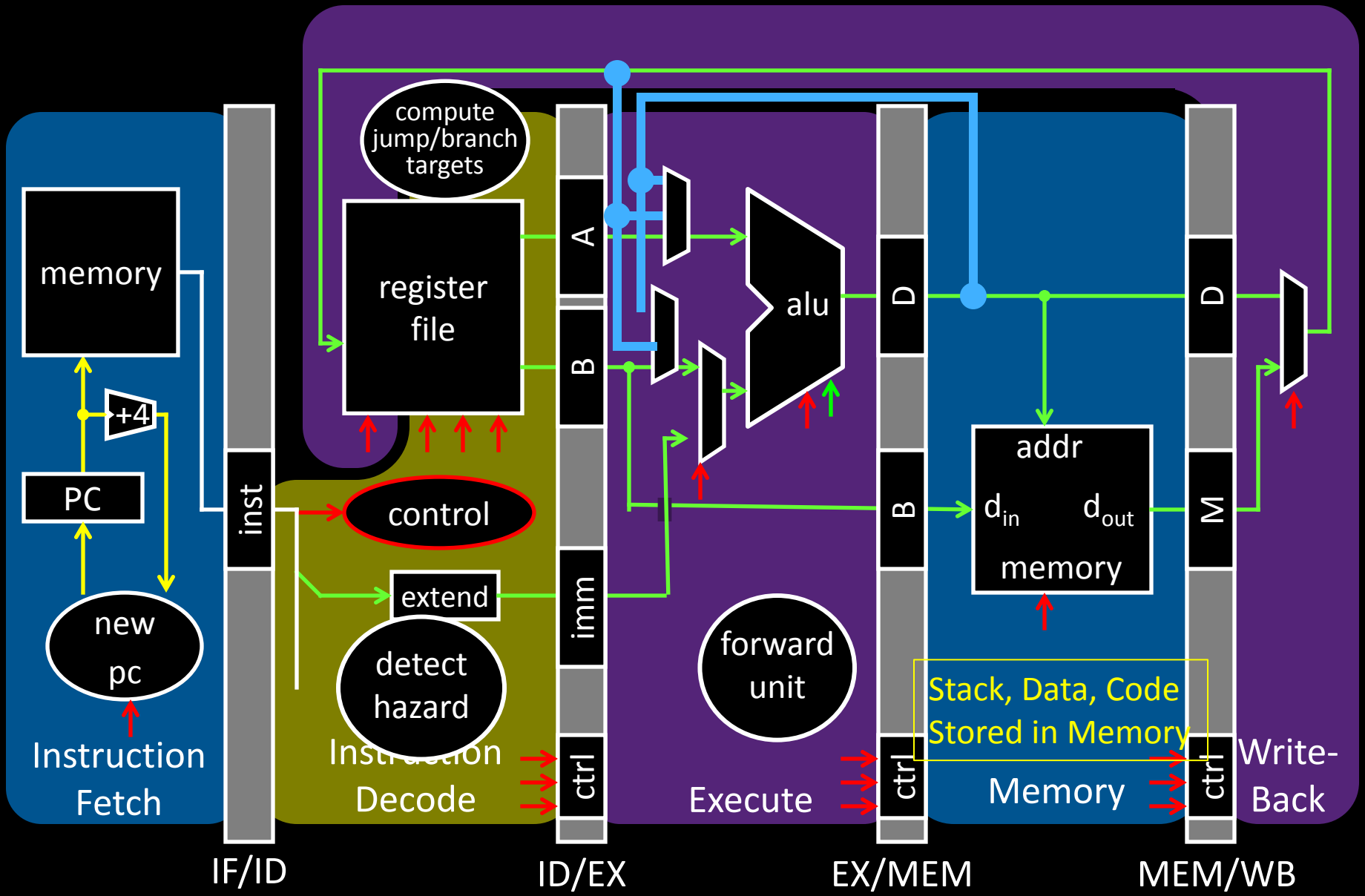
CS 3410, Spring 2013

Computer Science

Cornell University

P&H Chapter 4.9, pages 509–515, appendix B.7

Administrivia: Where are we now in the course?



Administrivia

Next four weeks

- Week 11 (Apr 8): Lab3 due and Project3/HW4 handout
- Week 12 (Apr 15): Project3 design doc due and HW4 due
- Week 13 (Apr 22): Project3 due and Prelim3
- Week 14 (Apr 29): Project4 handout

Final Project for class

- Week 15 (May 6): Project4 design doc due
- Week 16 (May 13): Project4 due

Administrivia

Lab3 is *due this week*, Thursday, April 11th

Project3 available now

- Design Doc *due next week*, Monday, April 15th
- Schedule a Design Doc review Mtg now, by this Friday, April 12th
- Whole project due Monday, April 22nd
- **Competition/Games night Friday, April 26th, 5-7pm. Location: B17 Upson**

Homework4 is available now

- Work alone
- *Due next week*, Wednesday, April 17th
- Question1 on Virtual Memory is pre-lab question for in-class Lab4
- HW Help Session Thurs (Apr 11) and Mon (Apr 15), 6-7:30pm in B17 Upso

Prelim3 is in two and a half weeks, Thursday, April 25th

- Time and Location: 7:30pm in Phillips 101 and Upson B17
- Old prelims are online in CMS

Summary of Caches/TLBs/VM

Caches, Virtual Memory, & TLBs: answer three questions

Where can block be placed?

- Direct, n-way, fully associative

What block is replaced on miss?

- LRU, Random, LFU, …

How are writes handled?

- No-write (w/ or w/o automatic invalidation)
- Write-back (fast, block at time)
- Write-through (simple, reason about consistency)

Summary of Caches/TLBs/VM

Caches, Virtual Memory, & TLBs: answer three questions

Where can block be placed?

- Caches: direct/n-way/fully associative (fa)
- VM: fa, but with a table of contents to eliminate searches
- TLB: fa

What block is replaced on miss?

- varied

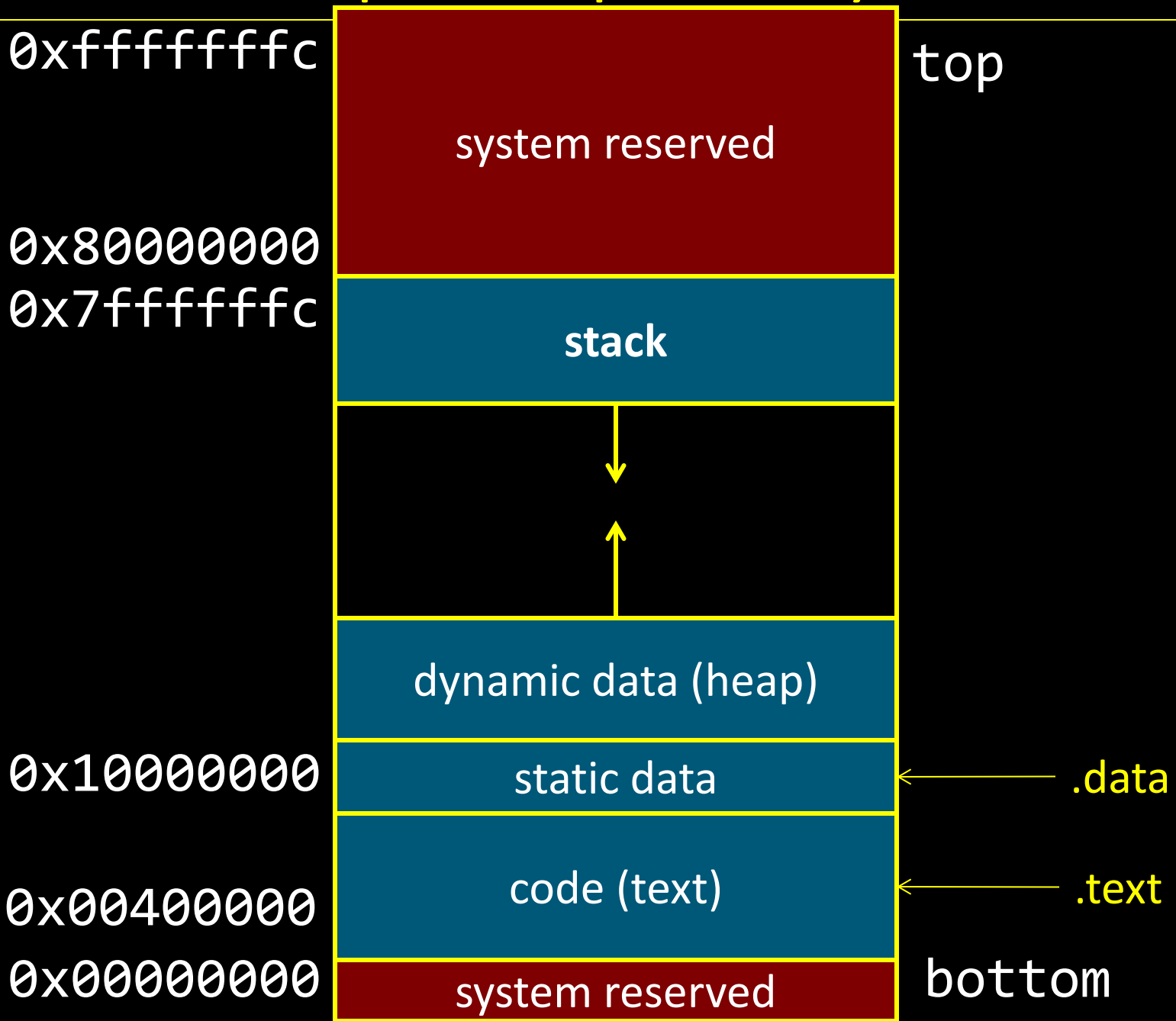
How are writes handled?

- Caches: usually write-back, or maybe write-through, or maybe no-write w/ invalidation
- VM: write-back
- TLB: usually no-write

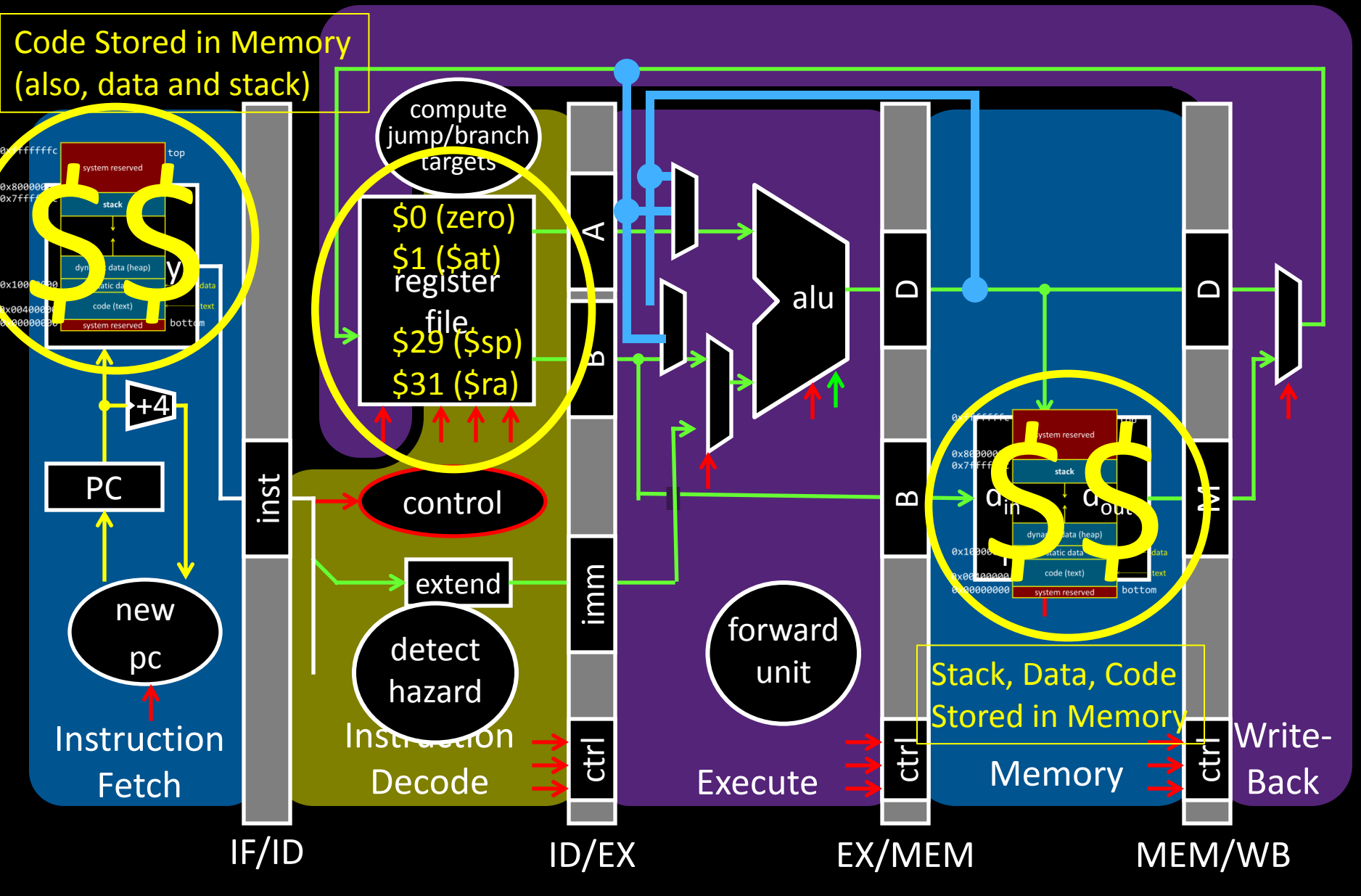
Summary of Cache Design Parameters

	L1	TLB	Paged Memory
Size (blocks)	1/4k to 4k	64 to 4k	16k to 1M
Size (kB)	16 to 64	2 to 16	1M to 4G
Block size (B)	16-64	4-32	4k to 64k
Miss rates	2%-5%	0.01% to 2%	10^{-4} to 10^{-5}%
Miss penalty	10-25	100-1000	10M-100M

Big Picture: Traps, Exceptions, System Calls (OS)



Big Picture: Traps, Exceptions, System Calls (OS)



Big Picture: Traps, Exceptions, System Calls (OS)

What happens with our pipeline if an ***exception*** occurs?

What are **exceptions?**

Any unexpected change in control flow.

Interrupt -> cause of control flow change external (async)

Exception -> cause of control flow change internal (sync)

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

Goals for Today

Exceptions

Hardware/Software Boundary

Privileged mode

Operating System

Exceptions vs Interrupts vs Traps vs Systems calls

Next Goal

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 software to help resolve exceptions

- Exceptions are at the hardware/software boundary

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

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

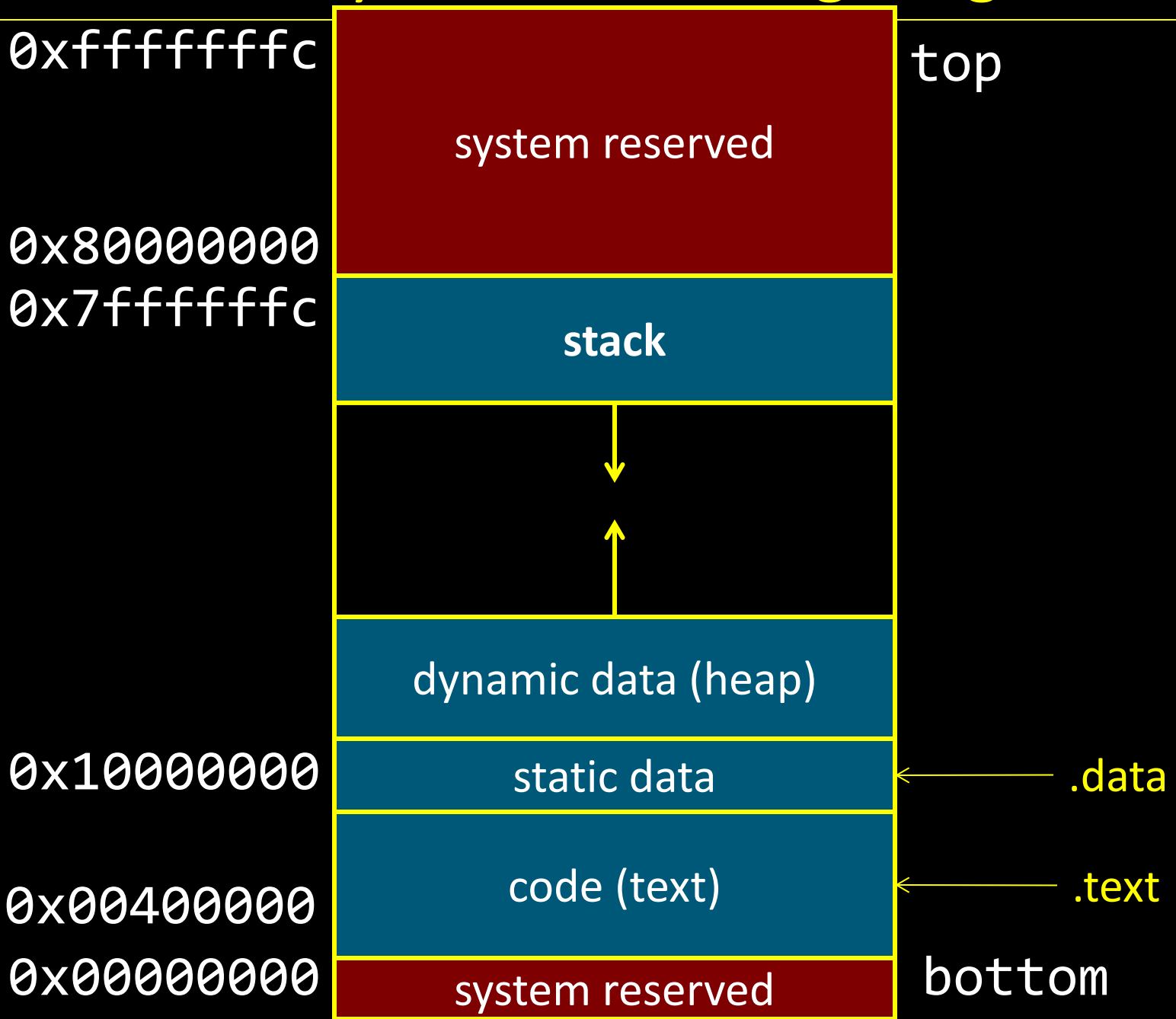
Drawbacks:

- 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, …

Anatomy of an Executing Program



Takeaway

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

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

Privileged Mode
aka Kernel Mode

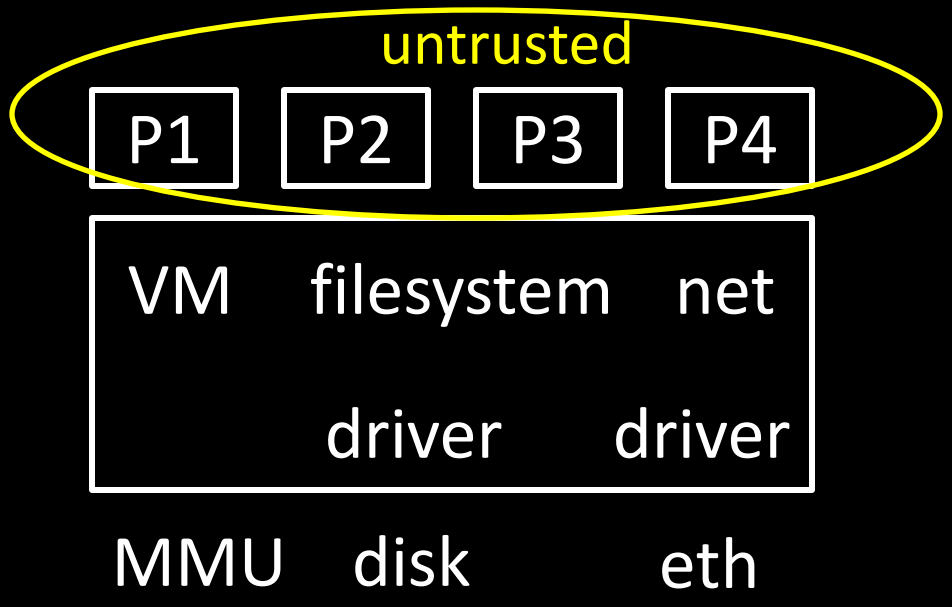
Operating System

Some things not available to untrusted programs:

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

Need trusted mediator: Operating System (OS)

- *Safe control transfer*
- *Data isolation*



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 ← 1; PC ← well known address

OS takes over…

- initialize devices, MMU, timers, etc.
- loads programs from disk, sets up pagetables, etc.
- Mode ← 0; PC ← 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

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

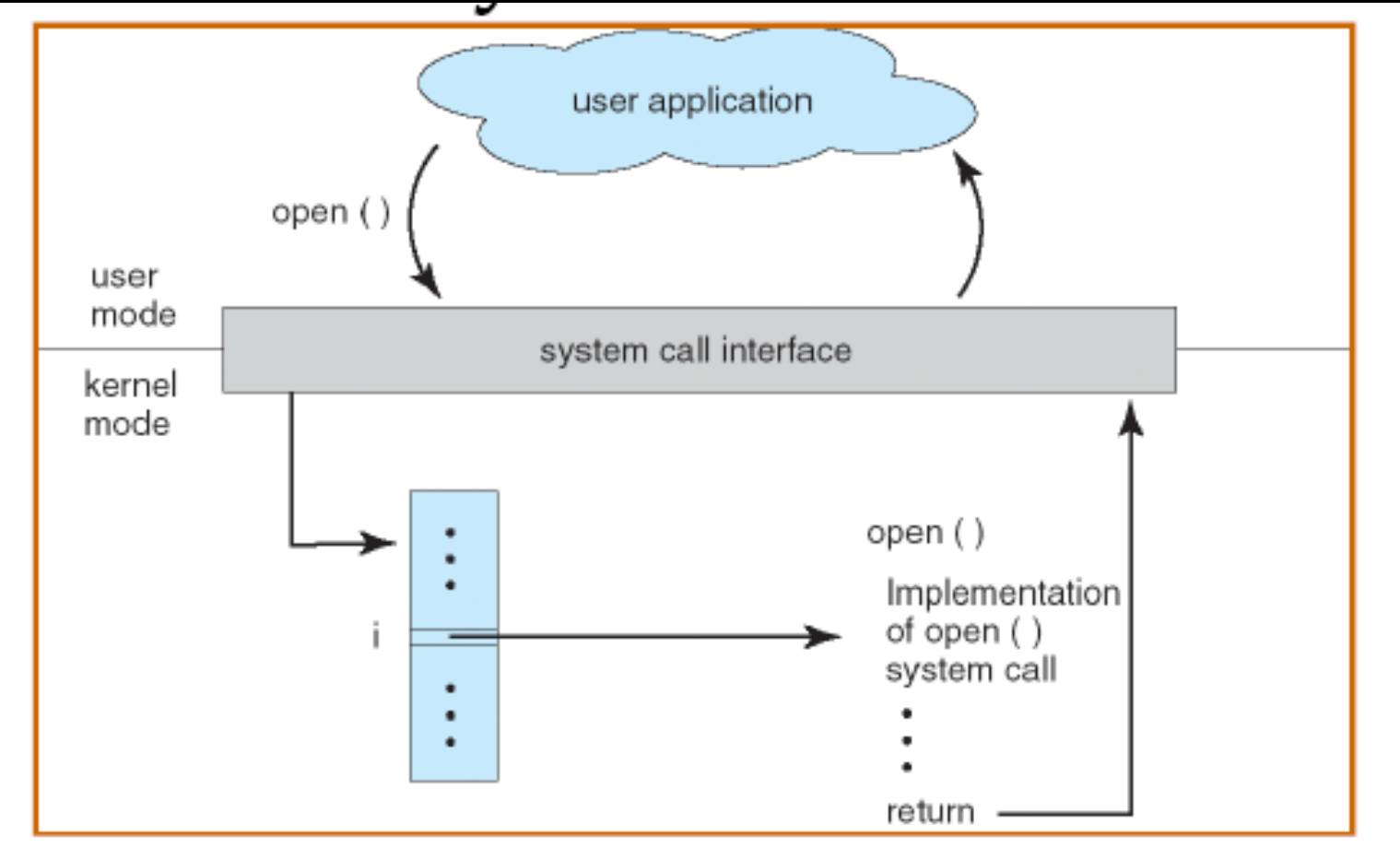
Compilers do not emit SYSCALL instructions

- 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()
- …


user application
open ( )
user mode
system call interface
kernel mode
i
open ( )
Implementation of open ( ) system call
return

Where does OS live?

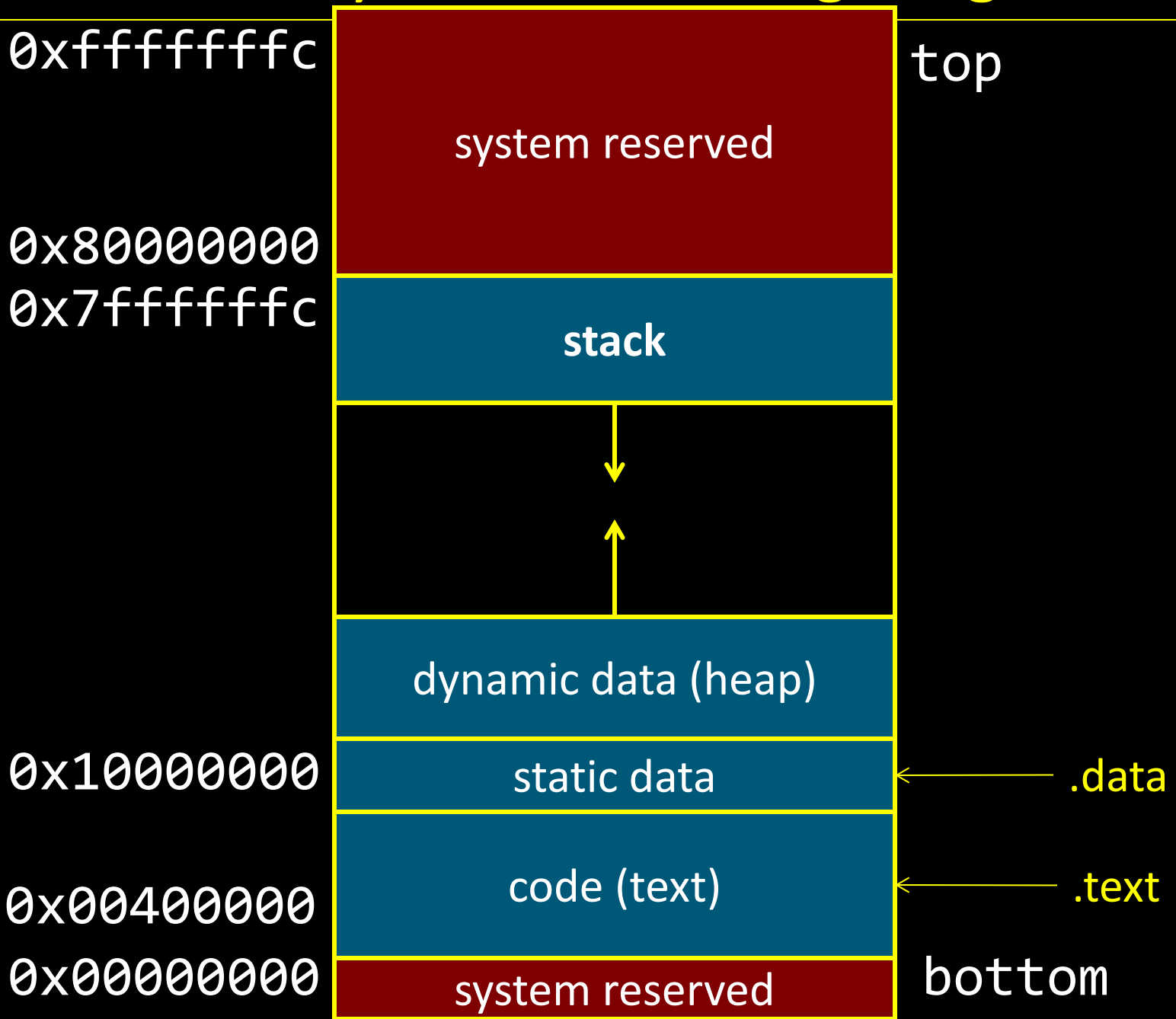
In its own address space?

- 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

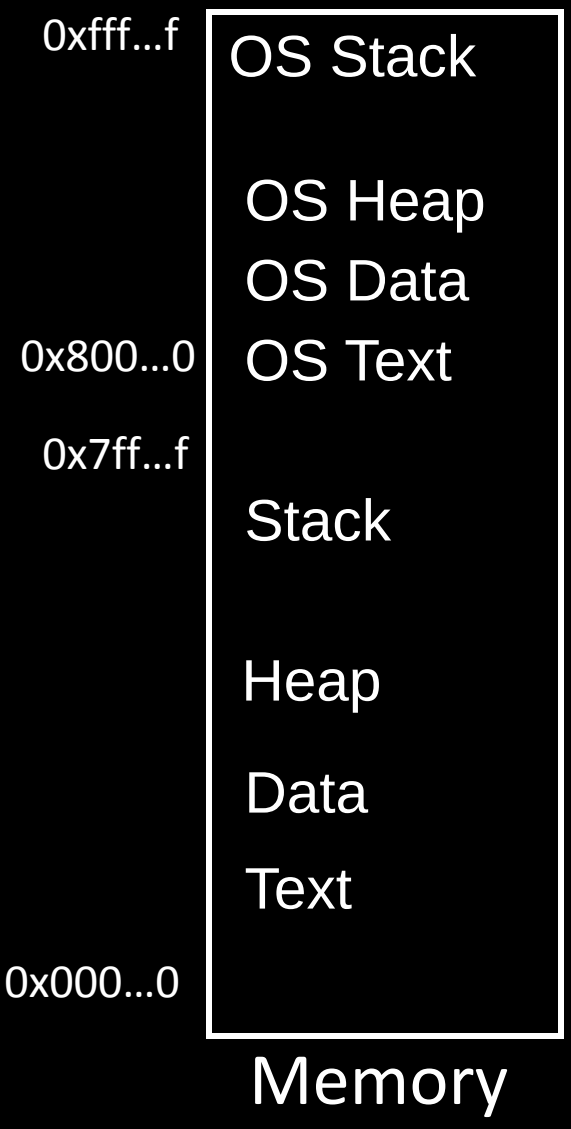
Typically all kernel text, most data

- At same VA 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



SYSCALL instruction

SYSCALL instruction does an atomic jump to a controlled location

- 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

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.

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

System call is any control transfer into the OS

Similarly, exceptions and interrupts are control transfers into the OS. How does the CPU and OS handle exceptions and interrupts?

Interrupts

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

Exceptions

Traps (System calls) are control transfers to the OS, performed under the control of the user program

Sometimes, need to transfer control to the OS at a *time when the user program least expects it*

- Division by zero,
- Alert from power supply that electricity is going out
- Alert from network device that a packet just arrived
- Clock notifying the processor that clock just ticked

Some of these causes for interruption of execution have nothing to do with the user application

Need a (slightly) different mechanism, that allows resuming the user application

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 0x80000180
# (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

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.

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.

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