CSC 369

Operating Systems

Bogdan Simion

bogdan@cs.toronto.edu

University of Toronto, Department of Computer Science

Why are you here?

Why did the department decide that an OS course is one of only 2 required 3rd year courses?!

a) We like to torture students. (We had to suffer through these courses, so all CS grads must suffer likewise.)

b) Interviewers always ask OS questions, so you might as well know the answers.

c) You will probably have to write OS code in the future.

d) Understanding how the OS works is a fundamental concept in CS, and will help you become a better programmer/scientist.

Administrivia

- Instructor Contact:
  - Email: bogdan@cs.toronto.edu
  - Office: BA 4268 Office Hours: Tue 3:30-4:30, Thu 3-4:30
- Webpage:
  - http://www.teach.cs.toronto.edu/~csc369h/fall
- Lecture schedule:
  - Check it carefully – some lectures may be in tutorial time!
- Piazza:
  - Linked from course webpage. Read daily. Ask questions there first
  - Please come talk to me if you have any concerns about using Piazza
- Course Syllabus (due dates, policies, etc.):
  - Linked from course webpage
  - Must read it carefully!

Course prerequisites

- Make sure you have the prerequisites!

- If you don't have the prerequisites, ask me for a waiver by email (no guarantees though!)

Course Overview

- Four assignments writing code in C (37%)
  - A1: System calls (8%)
  - A2: Synchronization (8%)
  - A3: Virtual memory (9%)
  - A4: File systems (12%)
  - In pairs, but individual contributions will be checked via midterm and final

Late policy:

10 grace tokens

Each token = 2 hours

- Weekly tutorial exercises for marks (4%)
- Weekly in class exercises for marks (4%)
- Midterm (10%)
- Final exam (45%) – must get >40% to pass the course!

Weekly Exercises

- Strong evidence that people learn better or faster by doing rather than passively listening

- Exercises make it easier to connect lecture material (information delivery) to assignments and real world

Exercises

Exercises may have many forms:

- In class group exercises
- In class exercise, but option to hand in online
- Out of class exercise to be submitted online
- Most tutorials will have an exercise
- Tutorial exercises will be largely related to assignments

Marking Scheme:

- In class: Participation / effort based
- Tutorial: Correctness (and / or effort)
- Tutorial exercises will be best 8 out of 10
- In class exercises will be best 8 out of 10

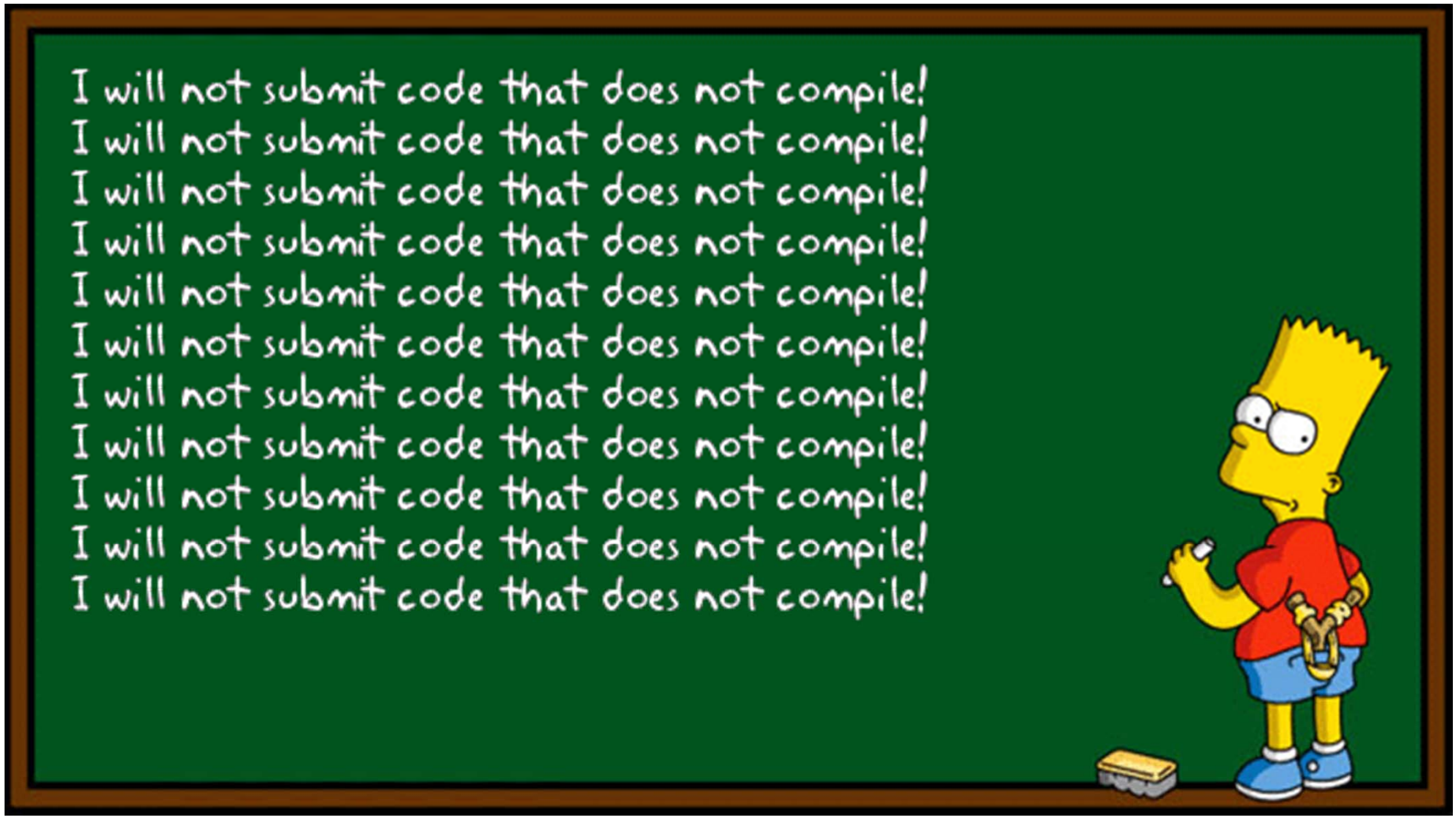
Assignments

- Write good, professional code
- Comment it properly
- Debug it properly, find corner cases
- Solve problems as they come, find workarounds if needed
- Very important experience before getting a programming job
  - Please treat them as such!

Assignments

- Assignments are due at 10:00 p.m. on the due date - check website for final due dates
- Code must work on teaching labs (formerly known as CDF)
- Make sure you commit all your source files; we cannot find files you never submitted
- Code style matters!
- Test-as-you-go
- The code you submit has to work, even if it doesn't implement everything
- Code that does not compile gets zero marks!

Did you catch that?



Assignments

- svn (make sure to revise, if necessary)

- Start early on the assignments!
  - Make sure you **can** commit in your repository; **commit often!**
  - **Do not wait** until the very last minute to submit your assignment!
  - Read the submission instructions carefully ⇒ penalties for incorrect submissions!
- Must know your code for the entire assignment!
  - For partners: **Work together**, even if you split the work!
  - May conduct **interviews** for some or all assignments!
    - Not having a good understanding of all code ⇒ 0 marks!

Start on assignments early!

- Strongly correlated with:
  - ✓ better grades
  - ✓ less stress
  - ✓ better understanding of the concepts

- VM setup (see tutorial)

Bird's eye view

	M	T	W	R	F	S	S
Intro, Processes	11	12	13	14	15	16	17
Syscalls, threads	18	19	20	21	22	23	24
Concurrency, Semaphores, Locks	25	26	27	28	29	30	1
Synchronization, CVs, monitors	2	3	4	5	6	7	8
Scheduling	9	10	11	12	13	14	15
Intro to VM, paging	16	17	18	19	20	21	22
Midterm, Paging policies	23	24	25	26	27	28	29
TLBs, Intro to File Systems	30	31	1	2	3	4	5
No classes - Fall break	6	7	8	9	10	11	12
File Systems	13	14	15	16	17	18	19
FS recovery and fault tolerance	20	21	22	23	24	25	26
Deadlocks	27	28	29	30	1	2	3
Security, Exam review	4	5	6				

Holidays

Midterm

A1 go-ahead

A1 due

A2 go-ahead

A2 due

A3 go-ahead

A3 due

A4 go-ahead

A4 due

Don't Panic!

- Help is available in many forms
  - Lectures/tutorials: Ask questions!
  - Office hours: My time dedicated specifically to helping you
  - TA lab/office hours – get help with your code
  - Piazza: Faster response!
  - Email: Longer turnaround time
  - Anonymous email: for feedback
  - Undergraduate TA Help Center:

    http://web.cs.toronto.edu/program/ugrad/ug_helpcentre.htm

Academic Integrity: Plagiarism

- Very serious academic offences, penalties are severe!
- Clear distinction between collaboration and cheating
  - Of course you can help your friend track down a bug. It is never ok to submit code that is not your own, or to give someone else your code!
  - Ask questions on Piazza, but don't add details about your solution (especially your code!)
- All potential plagiarism cases will be investigated fully
- We will run plagiarism detection software at the end of term!
  - Dropping the course does not get you out of trouble!
- Tips:
  - Don't post your code in public places (Github, etc.)
  - Don't search for solutions - taking partial work or similar ideas will be detected!
  - Make sure your partner is not plagiarizing! Discuss your work together!

Readings

Strongly Recommended!

Operating Systems: Three Easy Pieces

by Remzi H. Arpaci-Dusseau and

Andrea C. Arpaci-Dusseau

If you want more:

Modern Operating Systems

by Andrew Tannenbaum

Do the readings!

Introduction - Overview

- Introduction to this term's topics
- What is an OS, why it's important
- Recap some concepts you should know from 209 (e.g., processes)
- Why is it exciting to understand your OS, or work on designing OSes?
- Major OS components
  - Reason about approaches taken by OS designers

Introduction

- What is an OS and why do I want one?

Introduction

- What is an OS and why do I want one?
  - How does it relate to the other parts of a computer system?
  - Convenient abstraction of H/W
  - Protection, security, authentication
  - Communication
- Make sure to review some computer organization (258), systems concepts (209) and C concepts (209)!

Introduction

- What is an OS and why do I want one?
  - How does it relate to the other parts of a computer system?
  - Convenient abstraction of H/W
  - Protection, security, authentication
  - Communication
- *What are the major goals and components of an OS?*

Goals of the OS

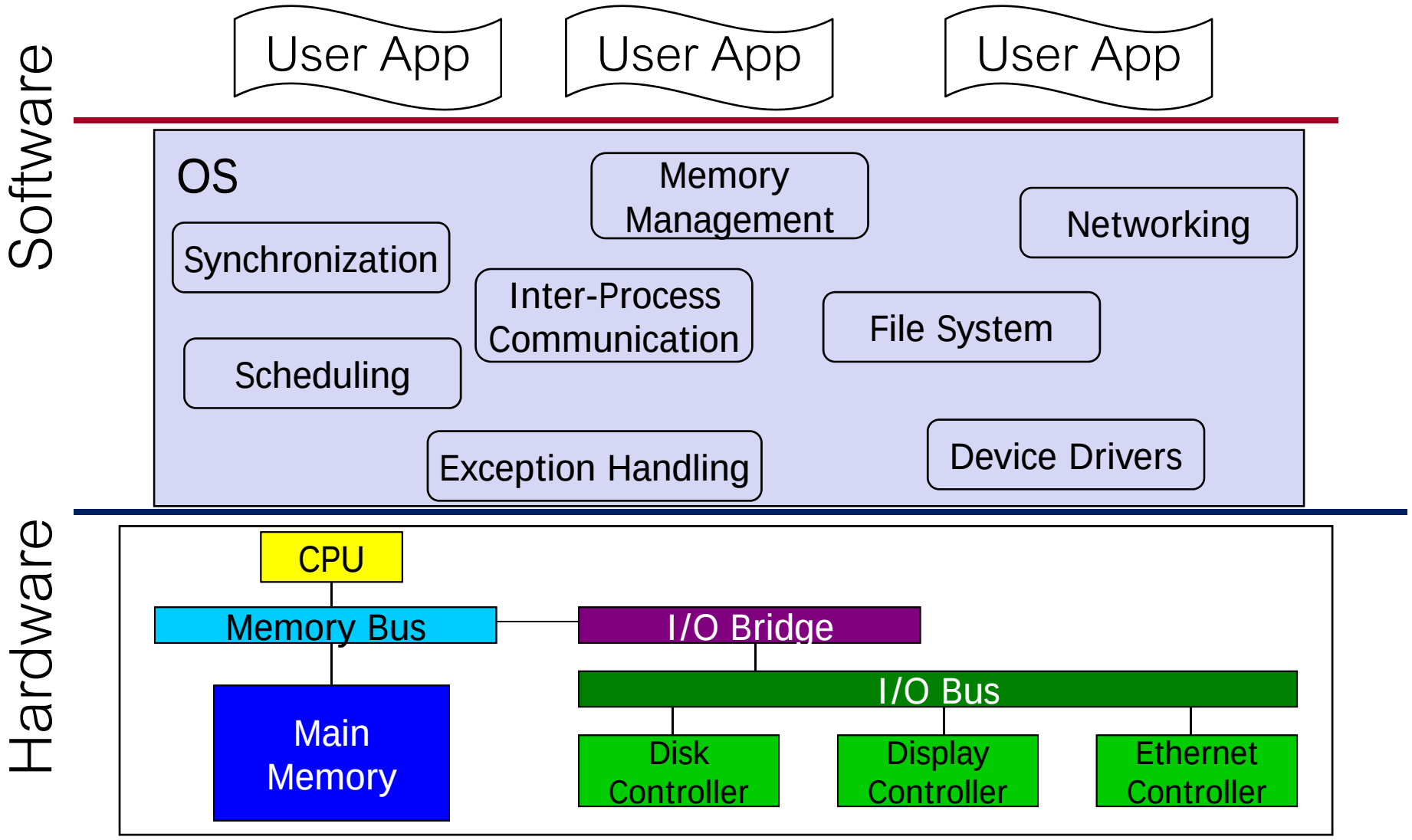
- Primary: *convenience for the user*
  - It must be easier to compute with the OS than without it
- Secondary: *efficient* operation of the computer system

- The two goals are sometimes contradictory
  - Actually, often contradictory
  - Which goal takes precedence depends on the purpose of the computer system

Roles of the OS

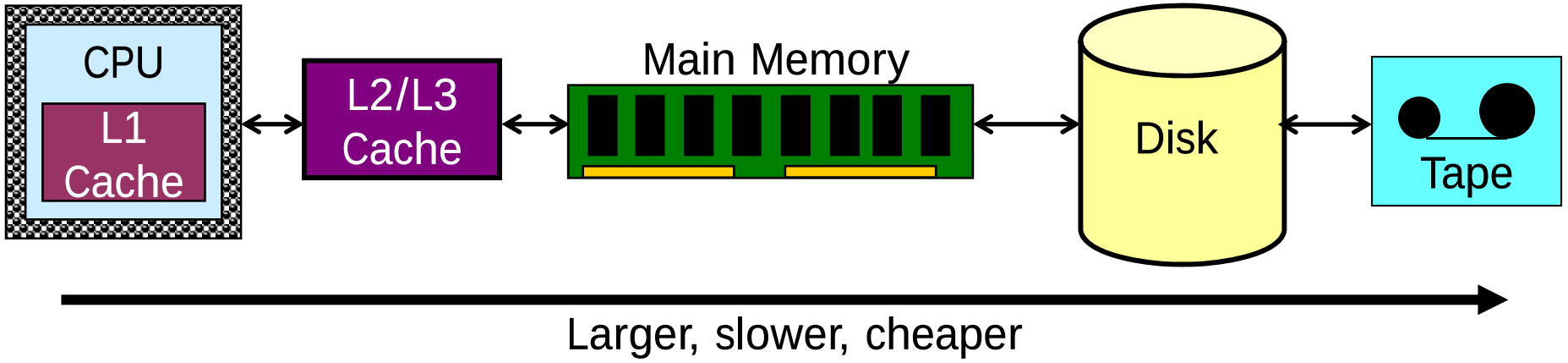
- An OS is a *virtual machine*
  - Extends and simplifies interface to physical machine
  - Provides a library of functions accessible through an API
- An OS is a *resource allocator*
  - allows the proper use of resources (hardware, software, data) in the operation of the computer system
  - provides an environment within which other programs can do useful work
- An OS is a *control program*
  - controls the execution of user programs to prevent errors and improper use of the computer
  - especially concerned with the operation and control of I/O devices

What is in the OS?



Storage Hierarchy

- processor registers, main memory, and auxiliary memory form a rudimentary memory hierarchy
- the hierarchy can be classified according to memory speed, cost, and volatility
- caches can be installed to hide performance differences when there is a large access-time gap between two levels



Major OS "Themes"

- Virtualization
  - Present physical resource as a more general, powerful, or easy-to-use form of itself
  - Present illusion of multiple (or unlimited) resources where only one (or a few) really exist
  - Examples: CPU, Memory (*demo)*
- Concurrency
  - Coordinate multiple activities to ensure correctness
- Persistence
  - Some data needs to survive crashes and power failures
- *Need abstractions, mechanisms, policies for all*

Next up...

- Hardware support for OS
- Bootstrapping
- Processes
  - What is a process?
  - Process lifecycle

Key Question: How to virtualize?

- Recall OS Goals:
  - Convenience for the user + efficient use of machine
- Virtualization helps with first goal
  - Virtual CPU + virtual memory gives each program the *illusion* that it owns the whole machine when it runs
- But how to do virtualization *efficiently*?
  - Consider Java language and Java Virtual Machine (JVM)
  - JVM provides *virtual instruction set* (Java bytecode)
  - JVM *interprets* each virtual instruction
    - *VERY* convenient, gain portability, not limited by HW
    - *VERY SLOW!*

OS Solution – Limited Direct Execution

- Key Abstraction: *the process*
  - Includes everything OS needs to know to manage running programs (more details later)
- Main Idea – Direct Execution
  - Set up CPU so that next instruction is fetched from code of process that should be executed
  - Let it go → no overhead during process execution
- Cool. Where does the "Limited" part come in?
  - Need to restrict operations process can perform
  - Need to regain control

Hardware Support for OSs

- Protection domains -> mode bit
- Interrupts
- Timers
- Memory Management unit
- Other hardware

Protection Domains

- Dual-mode operation: *user mode* and *system mode* (a.k.a *supervisor mode*, *monitor mode*, or *privileged mode*)
- Add a *mode bit* to the hardware and designate some instructions as *privileged instructions*
  - Intel actually has 4 "rings" for protection (CS register)
  - Attempting to execute privileged instruction from user mode causes *protection fault* → trap to OS
  - Protects the operating system from access by user programs, and protects user programs from each other
- So how can a process do something privileged? *System calls*!

Protection Domains

- What instructions/operations would you expect to be privileged?

☑ Setting mode bit?

☑ Disabling interrupts?

☑ Enabling interrupts?

☑ Writing to device registers?

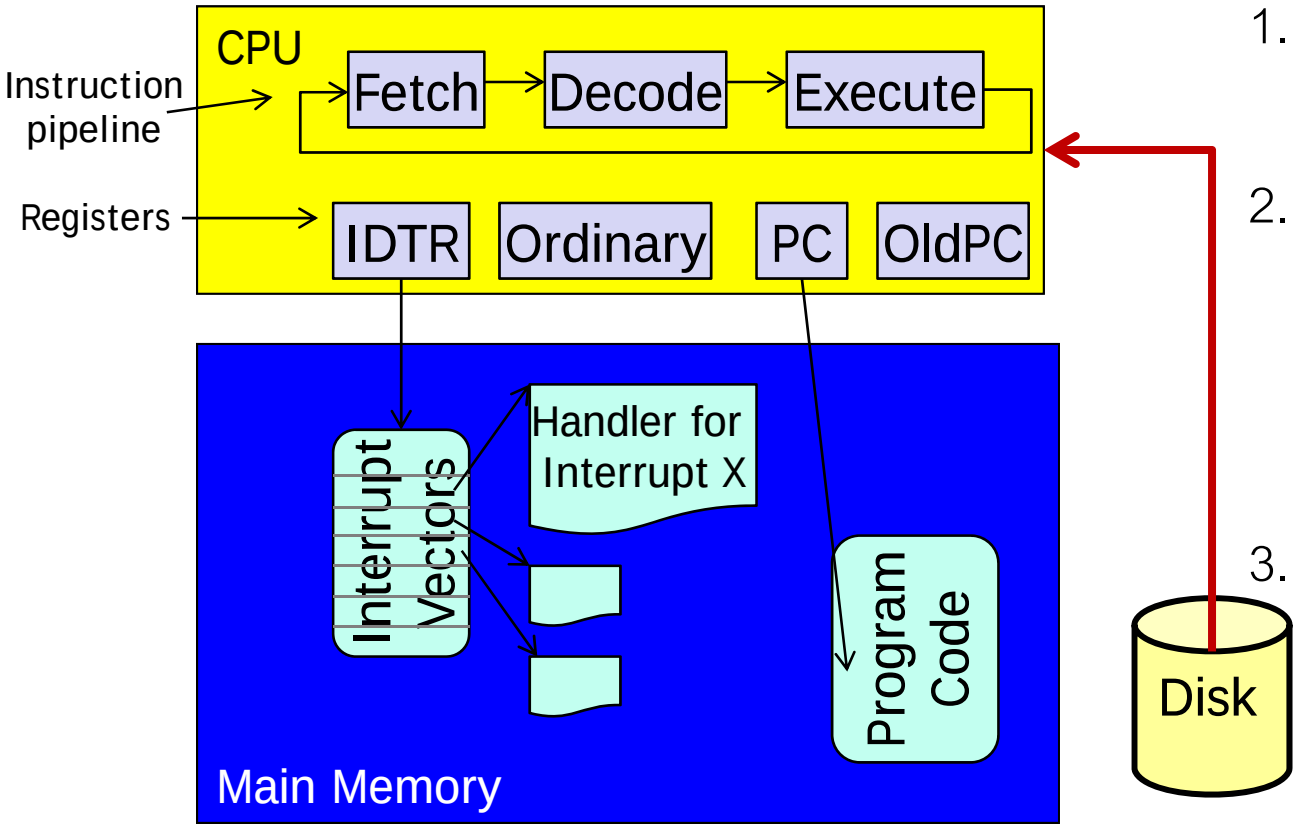
☑ Performing DMA?

☑ Halting the CPU?

- All of these are privileged!

Interrupts

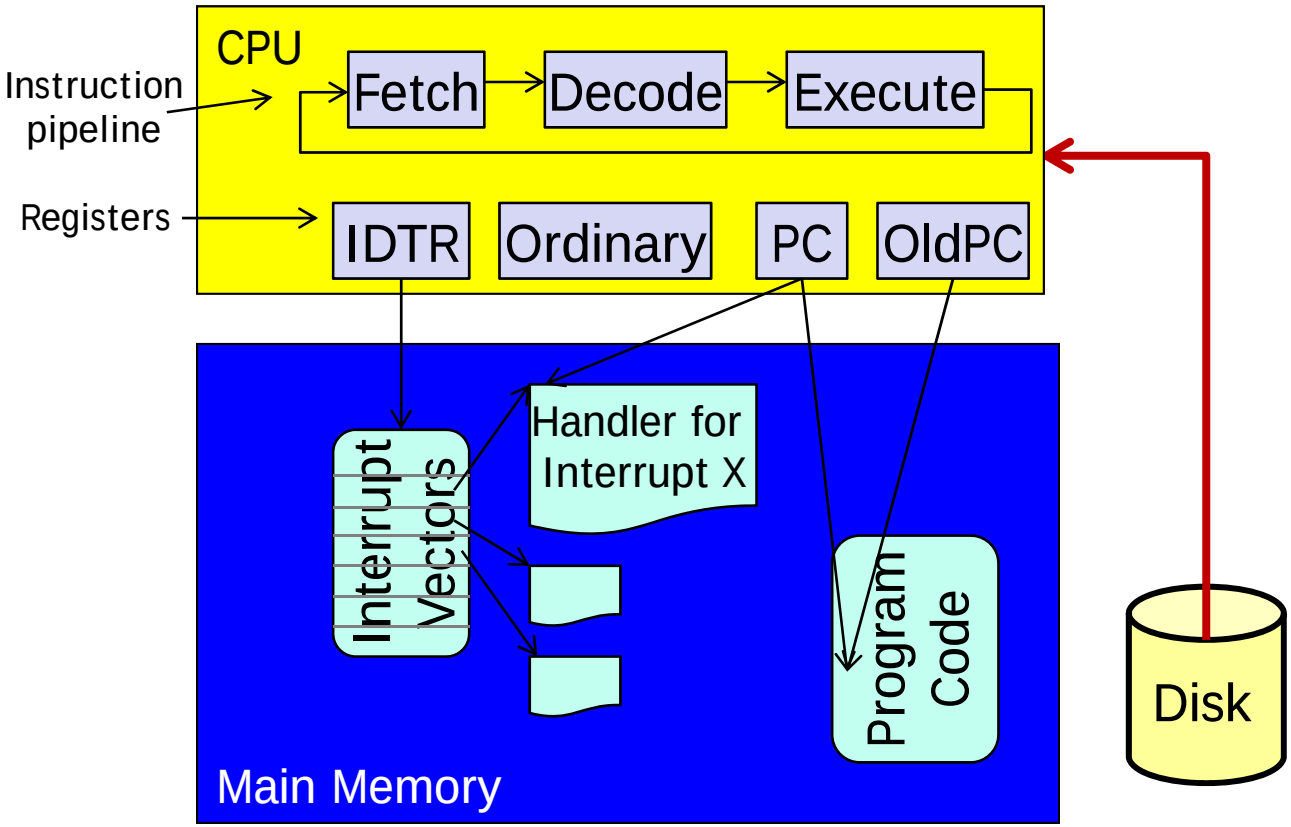
- Defn: a hardware signal that causes the CPU to jump to a pre-defined instruction called the *interrupt handler*
  - May be caused by software or hardware (Examples..)



1. OS fills in Interrupt Table (at boot time)
2. CPU execution loop: Fetch instruction at PC, Decode instr, execute instr... Forever
3. Interrupt occurs (from hardware)

Interrupts

- Defn: a hardware signal that causes the CPU to jump to a pre-defined instruction called the *interrupt handler*
  - May be caused by software or hardware (Examples..)



4. CPU changes mode, disables interrupts
5. Interrupted PC value is saved
6. IDTR + interrupt number is used to set PC to start of interrupt handler
7. Execution continues (saves additional state as first step)

Interrupts and the OS

- Interrupt *mechanism* supports OS goal of *efficient virtualization* in two ways:
  1. Any illegal instruction executed by a process causes a software-generated interrupt (aka *an exception)*
     - OS gets control whenever the process does something it shouldn't
       - Attempting to execute a privileged instruction
       - Illegal memory access
       - Illegal instructions (e.g. divide by zero)
  2. Periodic hardware-generated timer interrupt ensures OS gets control back at regular intervals
     - Can switch processes to give virtual CPU illusion

Today

- Hardware support for OS
- Bootstrapping
- Processes

Bootstrapping

- Hardware stores small program in non-volatile memory
  - BIOS – Basic Input Output System
  - Knows how to access simple hardware devices
    - Disk, keyboard, display
- When power is first supplied, this program starts executing
- What does it do?

Operating System Startup

- Hardware starts in system mode (kernel mode), so OS code can execute immediately
- OS initialization:
  - Initialize internal data structures
    - Machine dependent operations are typically done first
  - Create first process (init)
  - Switch mode to user and start running first process
  - Wait for something to happen
    - “something” always starts with an interrupt
    - OS is entirely driven by external events
      - External to the OS, that is, not the entire computer

Today

- Hardware support for OS
- Bootstrapping
- Processes
  - Definition
  - Representation
  - Lifecycle
  - API

What is a Process?

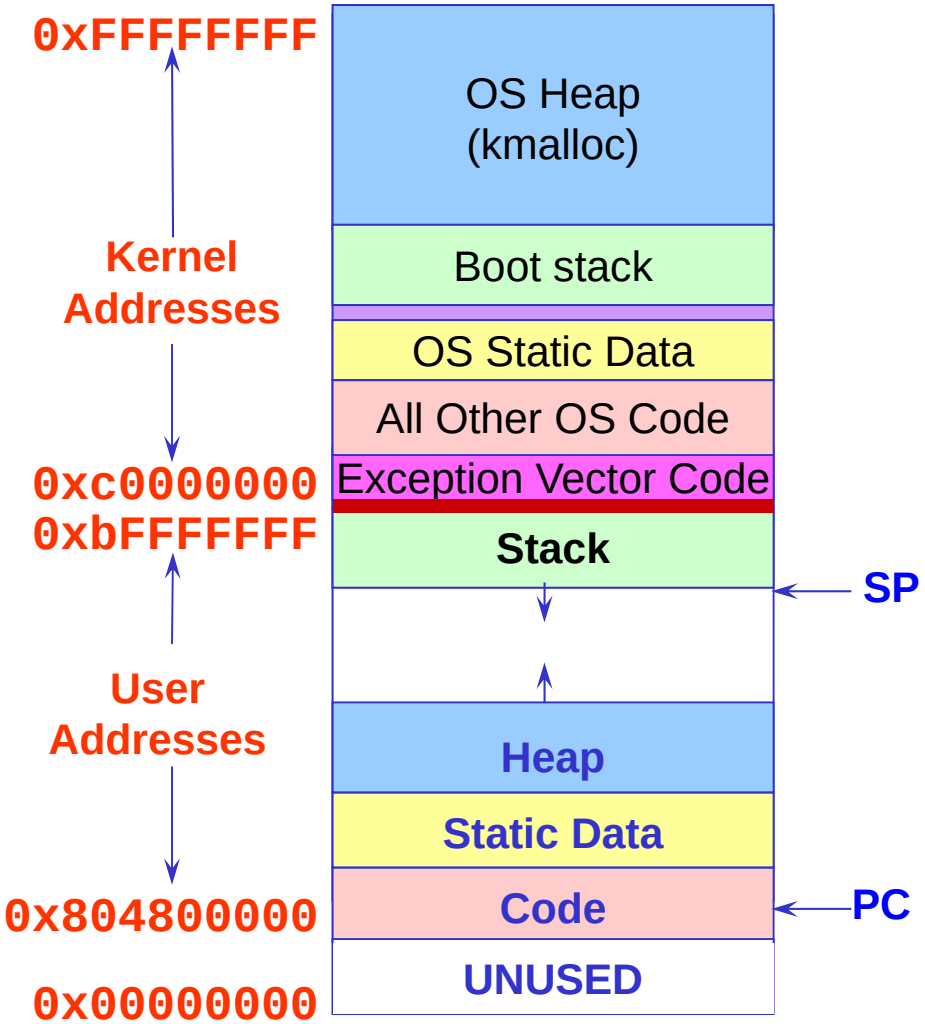
- OS abstraction for execution
  - AKA a job or a task or a sequential process
- Definition: it's a program in execution
  - An active entity
- Programs are static entities with the *potential* for execution

Program Layout in Memory

- What does a process's view of memory look like?
  - In other words, how is logical memory (logical address space) organized?

- BIG HINT: You saw this picture in CSC209 many times.

Program Layout in Memory



What is the OS view of a process?

- What data do we need to keep track of about a process?

OS View of a Process

- A *process* contains all of the state for a program in execution
  - An address space
    - The code and data for the executing user program + OS
    - An execution stack encapsulating the state of procedure calls
  - The program counter (PC) indicating the next instruction
  - A set of general-purpose registers with current values
  - A set of operating system resources
    - Open files, network connections, etc.
  - Context for kernel execution (a kernel thread & stack)
- A process is named using its process ID (PID)
- OS data about the process is stored in a *process control block (PCB)*

Process Control Block

- Generally includes:
  - process state (ready, running, blocked …)
  - program counter: address of the next instruction
  - CPU registers: must be saved at an interrupt
  - CPU scheduling information: process priority
  - memory management info: page tables
  - accounting information: resource use info
  - I/O status information: list of open files

Linux PCB

- Called the *task_struct* in Linux
  - Defined in /include/linux/sched.h

```
struct task_struct {
/* these are hardcoded - don't touch */
        volatile long state; /* -1 unrunnable, 0 runnable, >0 stopped */
        long counter; long priority; unsigned long signal;
        unsigned long blocked; /* bitmap of masked signals */
        unsigned long flags; /* per process flags, defined below */
        int errno; long debugreg[8]; /* Hardware debugging registers */
        struct exec_domain *exec_domain;
        /* various fields */
        struct linux_binfmt *binfmt;
        struct task_struct *next_task, *prev_task;
        struct task_struct *next_run, *prev_run;
        unsigned long saved_kernel_stack;
        unsigned long kernel_stack_page;
        int exit_code, exit_signal;
        …
```

Keeping track of processes

How does the OS keep track of processes?

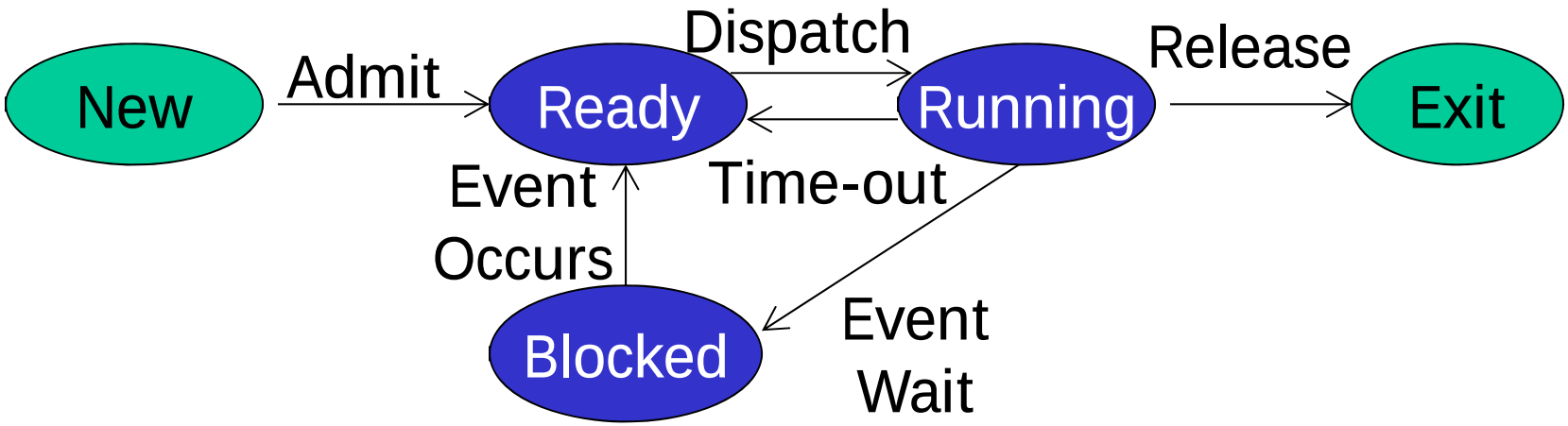
- Processes can be in various states (Ready, Running, Blocked)
- The OS maintains a collection of state queues that represent the state of all processes in the system
- Typically, the OS has one queue for each state
  - Ready, waiting for event X, etc.
- Each PCB is queued on a state queue according to its current state
- As a process changes state, its PCB is unlinked from one queue and linked into another

Today

- Processes
  - Definition
  - Representation
  - Lifecycle
  - API

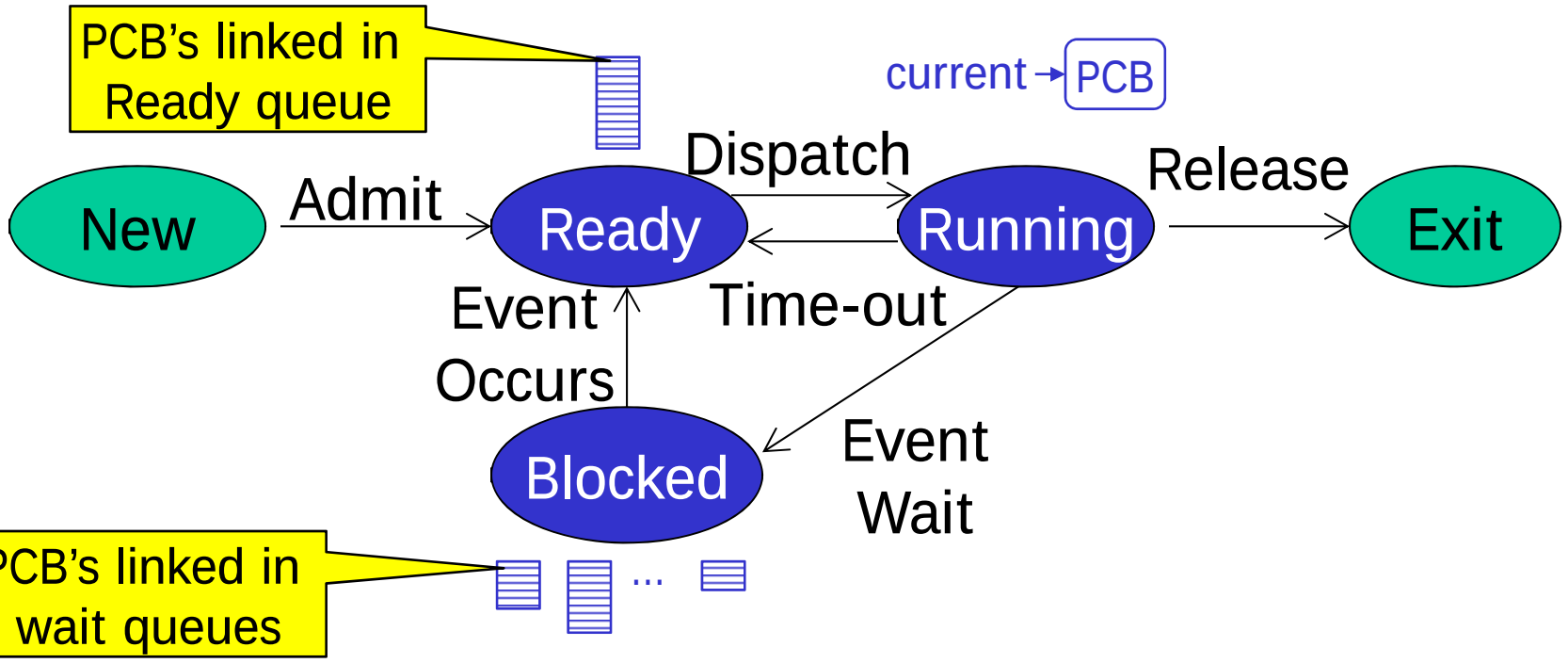
Process states & state changes

- The OS manages processes by keeping track of their *state*
  - Different *events* cause changes to a process state, which the OS must record/implement

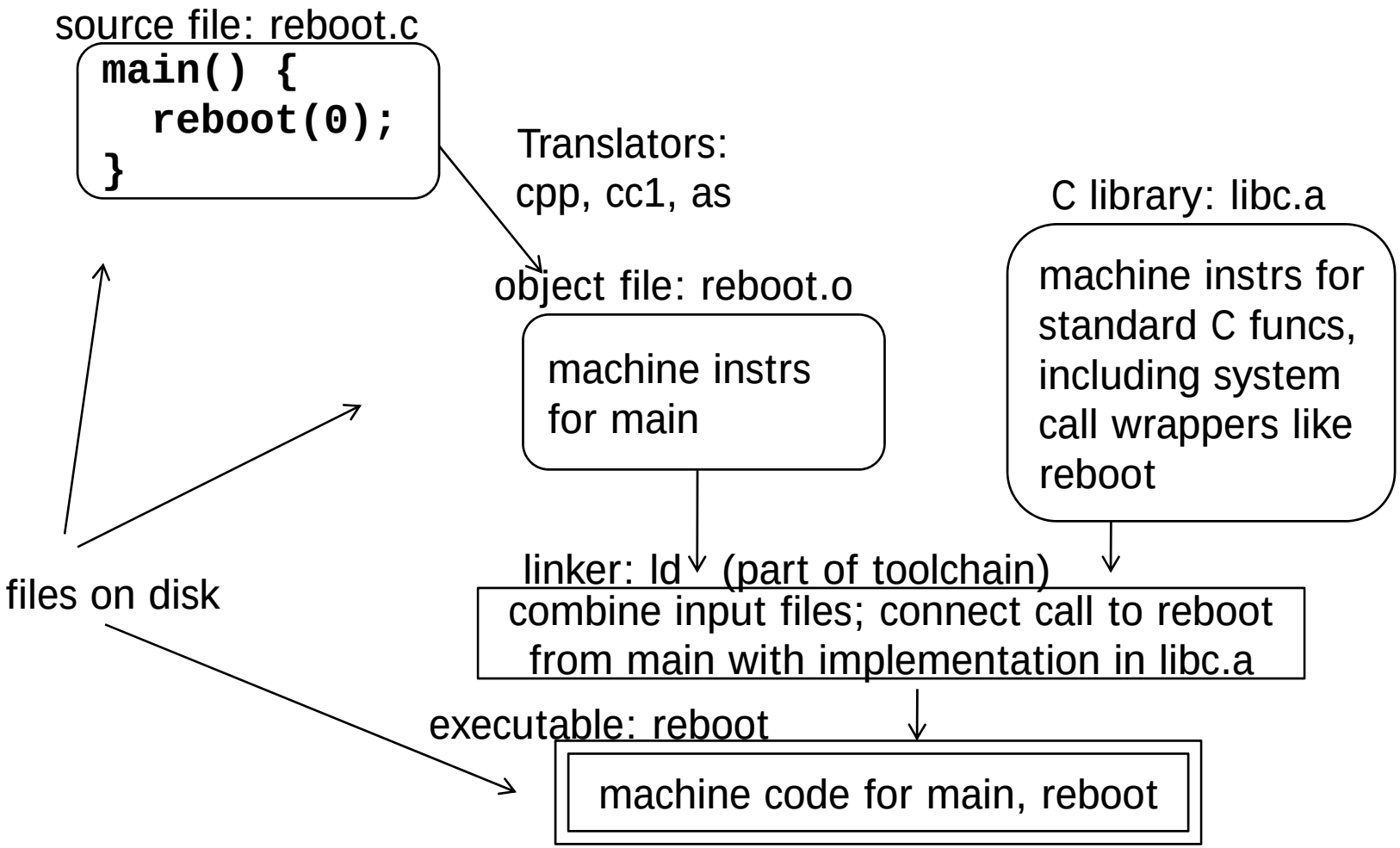


Process states & state changes

- The OS manages processes by keeping track of their *state*
  - Different *events* cause changes to a process state, which the OS must record/implement



From Program to Process… 1



From Program to Process… 2

1. Create new process
   - Create new PCB, user address space structure
   - Allocate memory
2. Load executable
   - Initialize start state for process
   - Change state to “ready”
3. Dispatch process
   - Change state to “running”

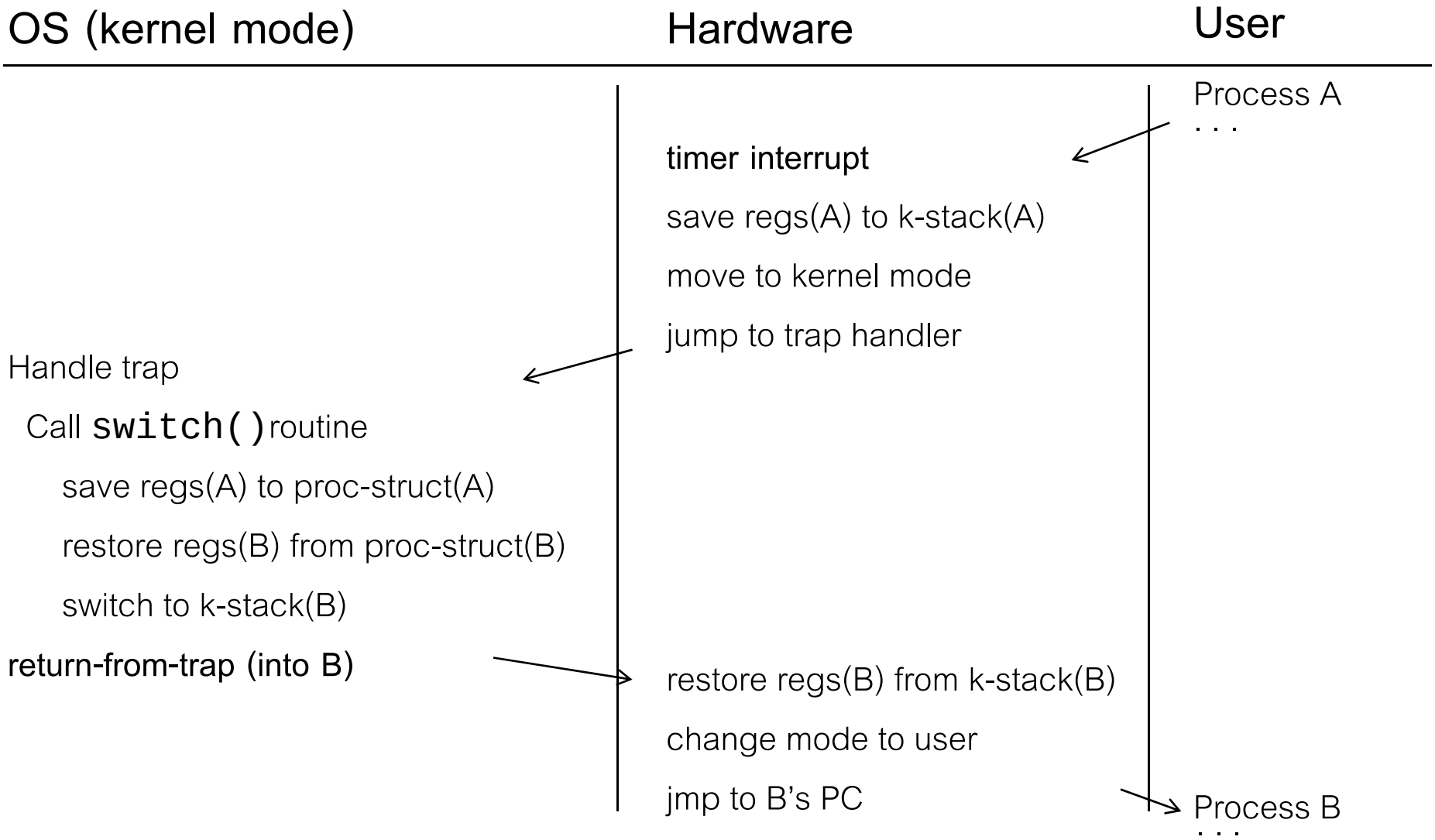
State Change: Ready to Running

- *context switch* == switch the CPU to another process

  by:

  - saving the state of the old process
  - loading the saved state for the new process
- When can this happen?
  - Process calls `yield()` system call (voluntarily)
  - Process makes other system call and is blocked
  - Timer interrupt handler decides to switch processes
    - Why would we ever need this?

Context Switch (from Process A to B)



Process Destruction

- exit()
- On exit(), a process voluntarily releases all resources
- But… OS can't discard everything immediately
- Why?
  - Must stop running the process to free everything
  - Requires *context switch* to another process
  - Parent may be waiting or asking for the return value
- exit() doesn't cause all data to be freed!
  - Zombies!

Zombies

- When a process exits, almost all of its resources are deallocated
  - Address space is freed, files are closed, etc.
- Some OS data structures retain the process's exit state
- The process retains its process ID (PID)
- It is a *zombie* until its parent cleans it up

Source: Plants vs Zombies

Today

- Processes
  - Definition
  - Representation
  - Lifecycle
  - API

Process Creation: Unix

- In Unix, processes are created using fork() system call
  `int fork()`
- fork()
  - Creates and initializes a new PCB
  - Creates a new address space
  - Initializes the address space with a **copy** of the entire contents of the address space of the parent
  - Initializes the kernel resources to point to the resources used by parent (e.g., open files)
  - Places the PCB on the ready queue
- Fork returns twice
  - Returns the child's PID to the parent, "0" to the child

Process Creation

- A process is created by another process
  - Parent is creator, child is created
    - In Linux, the parent is the "PPID" field of "ps –f"
  - Hierarchy: the first process (Unix): init (PID 1)
- In some systems, the parent defines (or donates) resources and privileges for its children
  - Unix: Process User ID is inherited – children of your shell execute with your privileges
- After creating a child, the parent may either wait for it to finish its task or continue in parallel
  - or continue for a while and then wait

Example: fork()

```
int main(int argc, char *argv[])
{
   char *name = argv[0];
   int child_pid = fork();
   if (child_pid == 0) {
       printf("Child of %s is %d\n", name, getpid());
       return 0;
   } else {
       printf("My child is %d\n", child_pid);
       return 0;
   }
}
```

What does this program print?

Process Creation: Unix (2)

- How do we actually start a new program?

```
int exec(char *prog, char *argv[])
```

- exec()
  - Stops the current process
  - Loads program “prog” into the process’ address space
  - Initializes hardware context and args for the new program
  - Places the PCB onto the ready queue
  - Note: It **does not** create a new process

- What does it mean for exec to return?

Inter-Process Communication (IPC)

- By design, processes are isolated from each other
  - But we often want them to exchange information
- OS provides various mechanisms for *inter-process communication*
  - Passing arguments to a newly exec'd program
    - Part of the execv() system call
  - Returning an integer exit status from child to parent
    - Part of the waitpid() and exit() system calls
  - Sending *signals* – a software analog of hardware interrupts
    - Provided by the kill() system call
  - Shared file system
  - Message passing, shared memory, synchronization primitives...