Introduction

Operating Systems, Spring 2018

Department of Computer Science
Rutgers University

Logistics

Instructor: John-Austen Francisco

TAs: (Daniel Hidalgo, Yujie Ren ??)

More information (including office hours, email addresses, etc.)

- http://sakai.rutgers.edu/

If you need help outside of office hours, please email the TA

- Start subject line with [CS416]
- Emails without this tag will likely be ignored

If you would like to meet with me, please send me a tagged email

You will need an account on the iLab machines. LCSR has a Web page for you to create the account

Course Overview

Goals:

Understanding of OS mechanisms and design patterns in systems

Learn how to choose reasonable thresholds and design real systems

Prerequisites:

Comfortable with C coding, basic knowledge of architecture.

What to expect:

We will cover *core concepts* and issues in lectures

It is expected and required that you will work, think, read and code beyond what is presented in the lectures

3-4 large programming assignments in C/C++; randomly chosen demos*

1 midterm exam and 1 final exam (cumulative)

Warnings

Do NOT ignore these warnings!

- We will be working on large programming assignments. If you do not have good programming skills or cannot work hard consistently on these assignments, don't take this course.
- Cheating will be punished severely.
- For more information on academic integrity, see: http://www.cs.rutgers.edu/policies/academicintegrity/

You will learn a lot during this course, but you will have to work hard to do well in it!

Textbook and Topics

“ Operating Systems: Three Easy Pieces”, by Remzi Arpaci-Dusseau and Andrea Arpaci-Dusseau.

http://pages.cs.wisc.edu/~remzi/OSTEP

Topics

- Processes and threads
- Processor scheduling
- Synchronization
- Virtual memory
- File systems
- I/O systems
- Distributed systems

Grading

Rough guideline

416:

Midterm: 20%

Final: 20%

Assignments: 60%

Programming assignments will be done in groups. Exams and written homeworks are individual.

Cheating policy: Collaboration at the level of ideas is good. Copying code or words is not good. Giving advice is good. Pointing out exact errors or correcting them is not good.

Grading

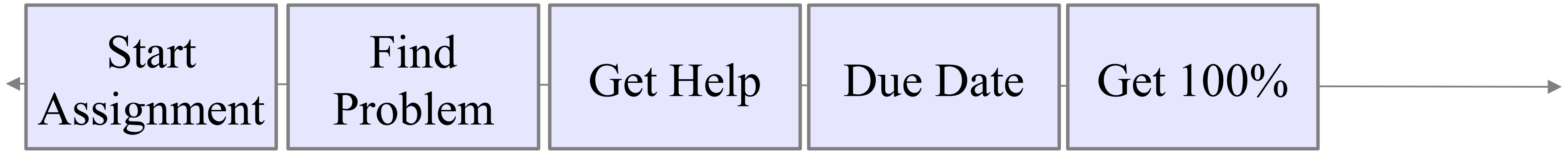
- Assignment hand-ins MUST be on time
  - Late hand-ins will not be accepted
  - Programming assignments must be turned in electronically via Sakai
    - Instructions will be posted on the web
    - We will promptly close the turn-in at the appointed time

  - Ask questions early and often

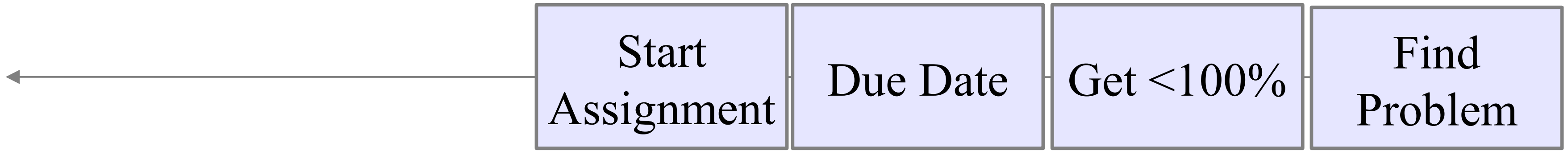
  - Use Sakai as a one-way CSV/Git/SVN repository
    - Submit early and often
    - Do not wait until the last moment and then have problems

Chronology of Learning

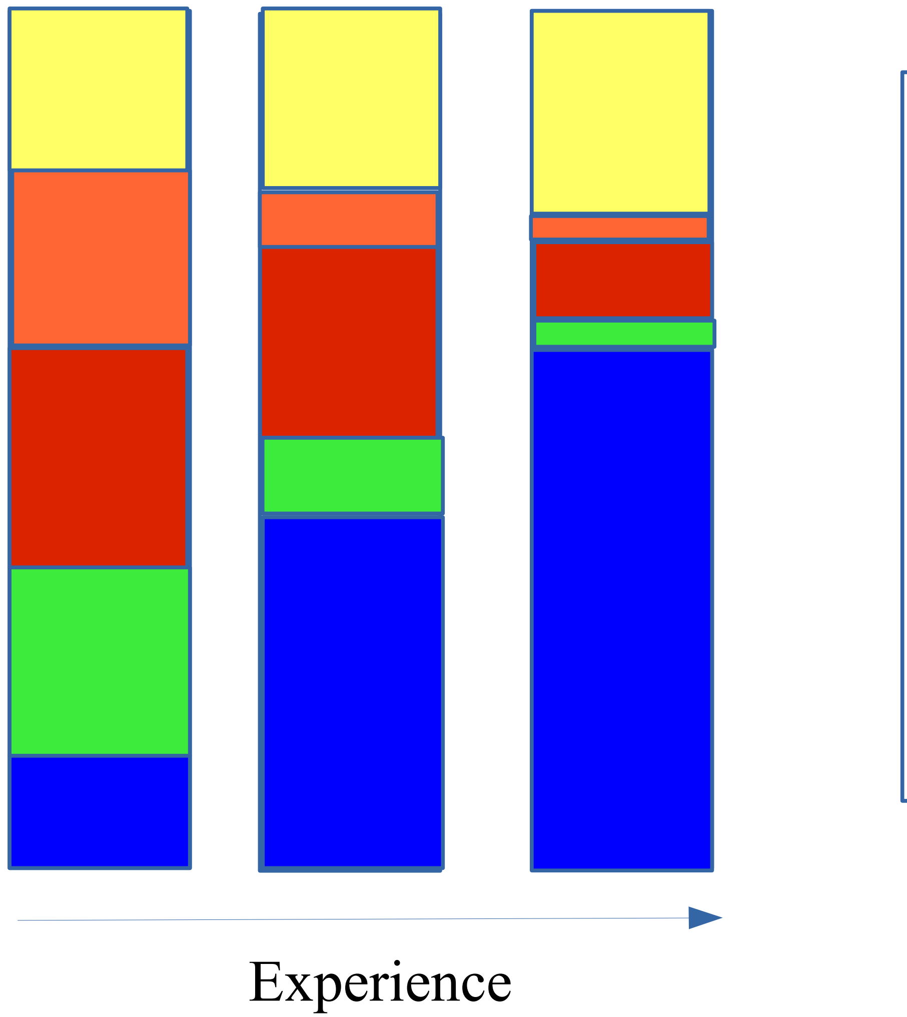
Student Strategy for Excellent Final Grade:



Student Strategy for Not-So-Excellent Final Grade:



Programming: Division of Labor



Final Note About Grading

Things that I will not do at the end of the course:

- Give you an incomplete because you think that your grade was bad
- Bump your grade up because you feel you deserve it
- Review earlier assignments or exams to try to find extra points for you
- Give you a passing grade so that you can graduate
- Give you points because you are not a CS major

Bottom line: You get exactly the grade that your exam and assignment scores determine. If you want a good grade, the best approach is to take the course seriously from day one.

Today

What is an Operating System?

Major OS components

A little bit of history

What Is An Operating System?

application (user)

---

operating system

---

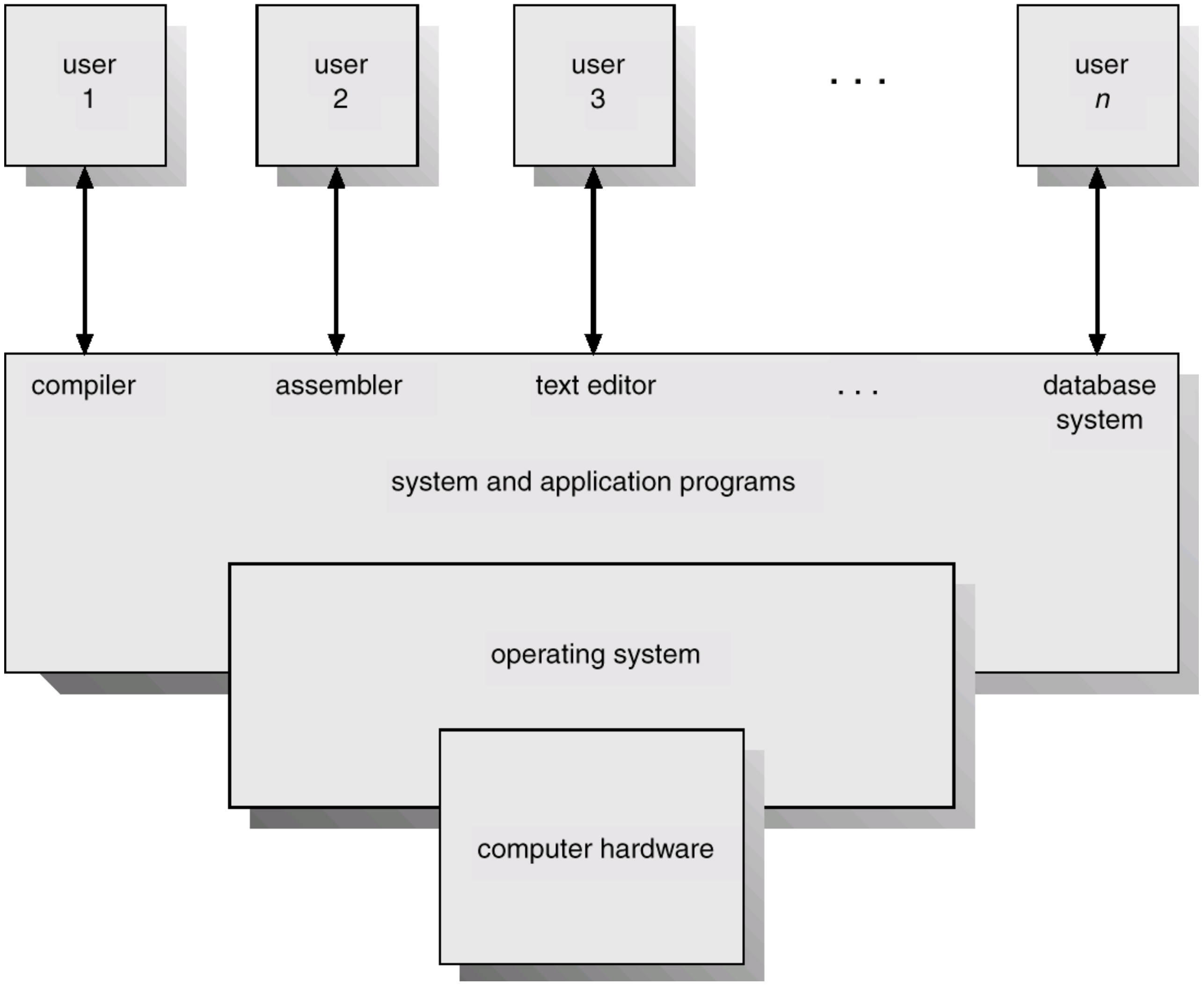
hardware

A software layer between the hardware and the programs/users which provides a *virtual machine* interface: easy and safe

A *resource manager* that allows programs/users to share the hardware resources: fair and efficient

A set of utilities to simplify application development

Abstract View of System Components



Why Do We Want An OS?

Benefits for application writers

- Easier to write programs
  - See high-level abstractions instead of low-level hw details
    - E.g. files instead of bunches of disk blocks
- Portability

Benefits for users

- Easier to use computers
  - Can you imagine trying to use a computer without the OS?
- Protection/safety
  - OS protects programs from each other
  - OS protects users from each other

Mechanism and Policy

application (user)

---

operating system: *mechanism+policy*

---

hardware

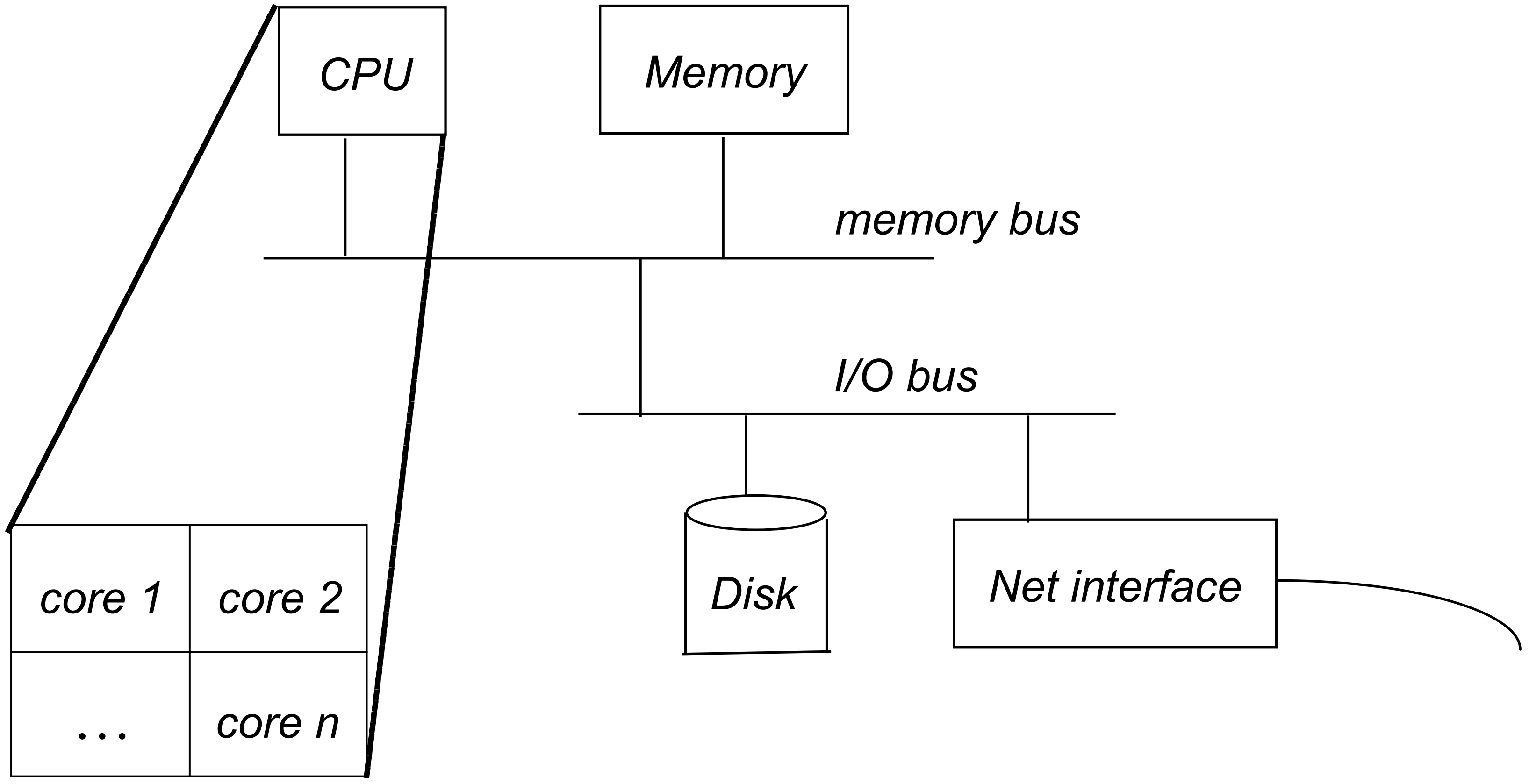
Mechanisms: data structures and operations that implement an abstraction (e.g., the file system buffer cache)

Policies: the procedures that guide the selection of a certain course of action from among alternatives (e.g., the replacement policy of the buffer cache)

Want to separate mechanisms and policies as much as possible

- Different policies may be needed for different operating environments

Basic Computer Structure



System Abstraction: Processes

A process is a system abstraction:
illusion of being the only job in the system

user: *run application*

operating system: *process*

hardware: *computer*

create, kill processes,
inter-process comm.

Multiplexing resources

Processes: Mechanism and Policy

Mechanism:

Creation, destruction, suspension, context switch, signaling, IPC, etc.

Policy:

Minor policy questions:

- Who can create/destroy/suspend processes?
- How many active processes can each user have?

Major policy question that we will concentrate on:

- How to share system resources between multiple processes?
- Typically broken into a number of orthogonal policies for individual resources, such as CPU, memory, and disk.

Processor Abstraction: Threads

A thread is a processor abstraction: illusion of having 1 processor per execution context

application: *execution context*

create, kill, synch.

operating system: *thread*

context switch

hardware: *processor*

Threads: Mechanism and Policy

Mechanism:

- Creation, destruction, suspension, context switch, signaling, synchronization, etc.

Policy:

- How to share the CPU between threads from different processes?
- How to share the CPU between threads from the same process?
- How can multiple threads synchronize with each other?
- How to control inter-thread interactions?
  - Can a thread murder other threads at will?

Memory Abstraction: Virtual memory

Virtual memory is a memory abstraction:
illusion of large contiguous memory, often more memory than physically available

application: *address space*

---

virtual addresses

operating system: *virtual memory*

---

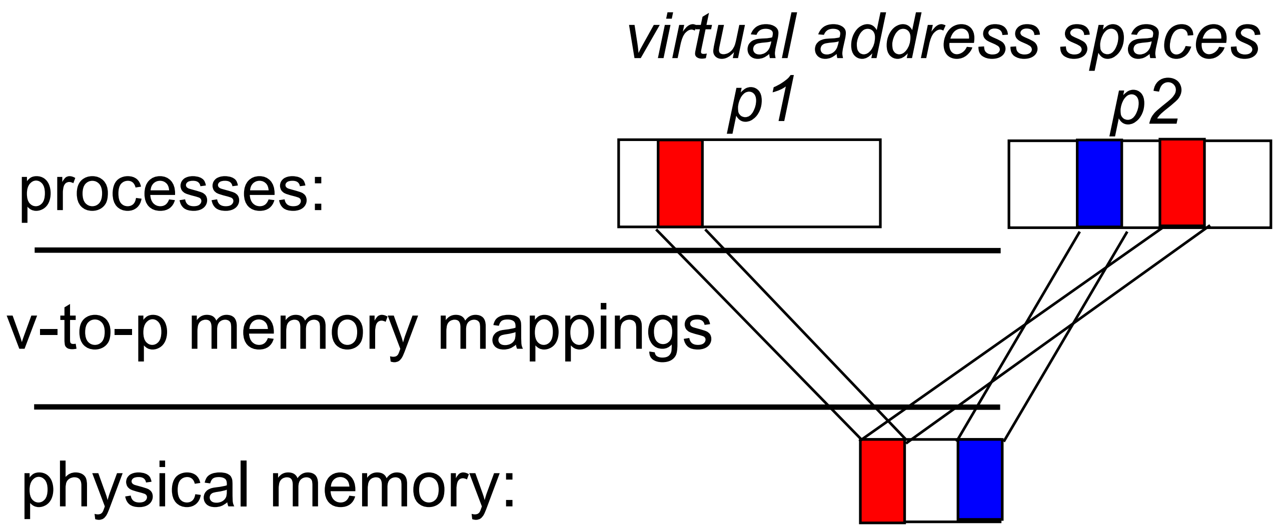
physical addresses

hardware: *physical memory*

Virtual Memory: Mechanism

Mechanism:

Virtual-to-physical memory mapping, page-fault, etc.



Virtual Memory: Policy

Policy:

How to multiplex a virtual memory that is larger than the physical memory onto what is available?

How should physical memory be allocated to competing processes?

How to control the sharing of a piece of physical memory between multiple processes?

Storage Abstraction: File System

A file system is a storage abstraction: illusion of structured storage space

application/user:	*copy file1 file2*
operating system:	*files, directories*
hardware:	*disk*

naming, protection, operations on files

operations on disk blocks

File System

Mechanism:

File creation, deletion, read, write, file-block-to-disk-block mapping, buffer cache, etc.

Policy:

Sharing vs. protection?

Which block to allocate?

File system buffer cache management?

Communication Abstraction: Messaging

Message passing is a communication abstraction:
illusion of reliable and ordered transport

application: sockets

naming, messages

operating system: *TCP/IP protocols*

network packets

hardware: *network interface*

Message Passing

Mechanism:

- Send, receive, buffering, retransmission, etc.

Policy:

- Congestion control and routing
- Multiplexing multiple connections onto a single network interface

Character & Block Devices

The device interface gives the illusion that devices support the same API – character stream and block access

application/user:	*read character from device*	naming, protection, read,write
operating system:	*character & block API*	hardware-specific PIO, interrupt handling, or DMA
hardware:	*keyboard, mouse, etc.*	

Devices

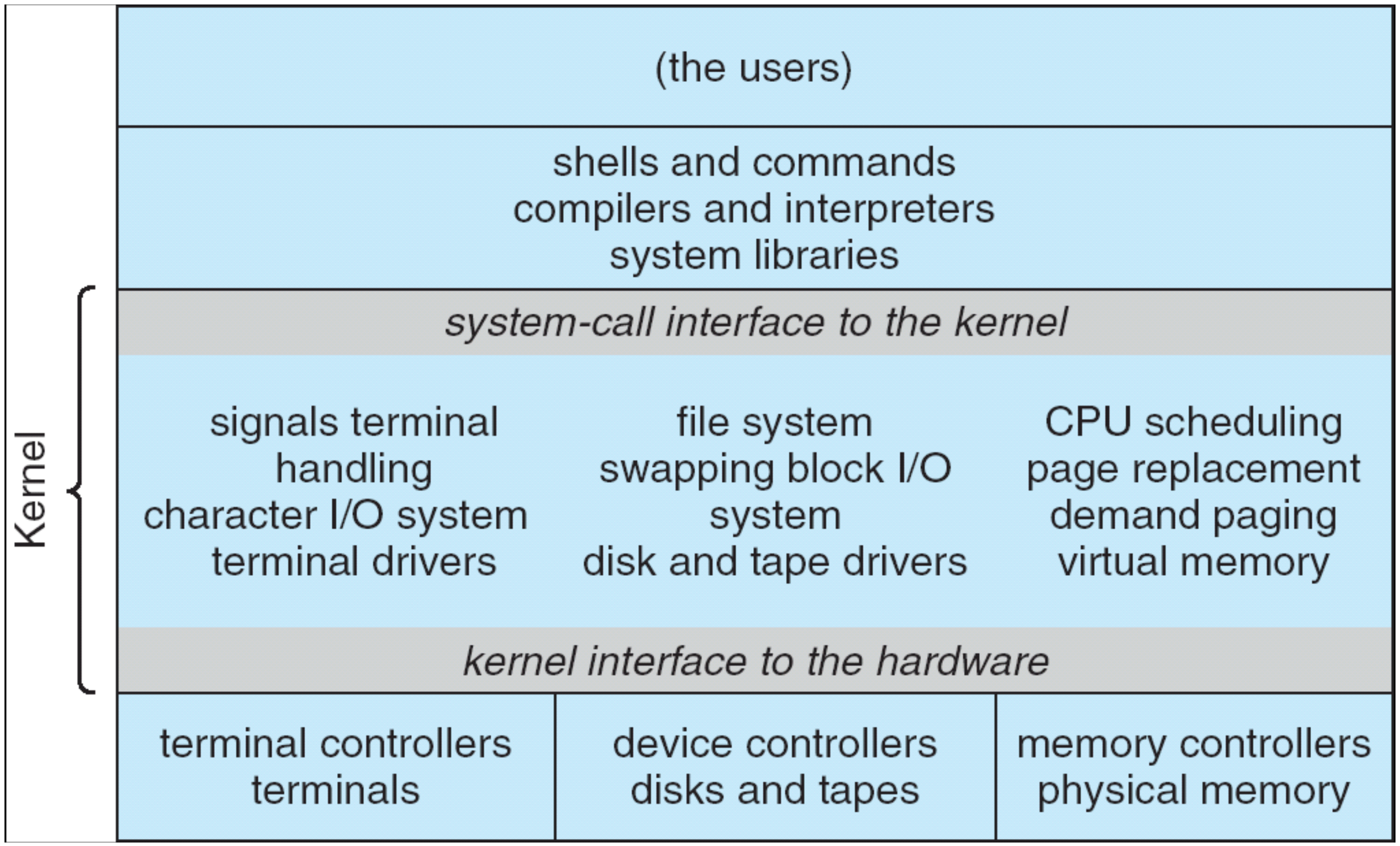
Mechanisms

- Open, close, read, write, ioctl, etc.
- Buffering

Policies

- Protection
- Sharing?
- Scheduling?

UNIX



Source: Silberschatz, Galvin, and Gagne

Major Issues in OS Design

Programming API: what should the VM look like?

Resource management: how should the hardware resources be multiplexed among multiple users?

Sharing: how should resources be shared among multiple users?

Protection: how to protect users from each other? How to protect programs from each other? How to protect the OS from applications and users?

Communication: how can applications exchange information?

Structure: how to organize the OS?

Concurrency: how do we deal with the concurrency that is inherent in OSes?

Major Issues in OS Design

Performance: how to make it all run fast?

Reliability: how do we keep the OS from crashing?

Persistence: how can we make data last beyond program execution?

Accounting: how do we keep track of resource usage?

Distribution: how do we make it easier to use multiple computers in conjunction?

Scaling: how do we keep the OS efficient and reliable as the offered load increases (more users, more processes, more processors)?

Brief OS History

- In the beginning, there really wasn't an OS
  - Program binaries were loaded using switches
  - Interface included blinking lights
- Then came batch systems
  - OS was implemented to transfer control from one job to the next
  - OS was always resident in memory
    - Resident monitor
  - Operator provided machine/OS with a stream of programs with delimiters
    - Typically, input device was a card reader, so delimiters were known as control cards

Spooling

CPUs were much faster than card readers and printers

Disks were invented – disks were much faster than card readers and printers

So, what do we do? Pipelining … what else?

- Read job 1 from cards to disk. Run job 1 while reading job 2 from cards to disk; save output of job 1 to disk. Print output of job 1 while running job 2 while reading job 3 from cards to disk. And so on …
- This is known as spooling: Simultaneous Peripheral Operation On-Line

Can use multiple card readers and printers to keep up with CPU if needed

Improves both system throughput and response time

Multiprogramming

- CPUs were still idle whenever executing program needed to interact with peripheral device
  - E.g., reading more data from tape
- Multiprogrammed batch systems were invented
  - Load multiple programs onto disk at the same time (later into memory)
  - Switch from one job to another when the first job performs an I/O operation
  - Overlap I/O of one job with computation of another job
  - Peripherals have to be asynchronous
  - Have to know when I/O operation is done: interrupt vs. polling
- Increase system throughput, possibly at the expense of response time
  - When is this better for response time? When is it worse for response time?

Time-Sharing

- As you can imagine, batching was a big pain
  - You submit a job, you twiddle your thumbs for a while, you get the output, see a bug, try to figure out what went wrong, resubmit the job, etc.
  - Even running production programs was difficult in this environment
- Technology got better: can now have terminals and support interactive interfaces
- How to share a machine (remember machines were expensive back then) between multiple people and still maintain interactive user interface?
- Time-sharing
  - Connect multiple terminals to a single machine
  - Multiplex machine between multiple users
  - Machine has to be fast enough to give illusion that each user has own machine
  - Multics was the first large time-sharing system – mid-1960’s

Parallel OS

Some applications comprise tasks that can be executed simultaneously

  - Weather prediction, scientific simulations, recalculation of a spreadsheet

Can speedup execution by running these tasks in parallel on many processors

Need OS, compiler, and/or language support for dividing programs into multiple parallel activities

Need OS support for fast communication and synchronization

Many different parallel architectures

Main goal is performance

Real-Time OS

Some applications have time deadlines by when they have to complete certain tasks

Hard real-time system

- Medical imaging systems, industrial control systems, etc.
- Catastrophic failure if system misses a deadline
  - What happens if collision avoidance software on an oil tanker does not detect another ship before the "turning or breaking" distance of the tanker?
- Challenge lies in how to meet deadlines with minimal resource waste

Soft real-time system

- Multimedia applications
- May be annoying but is not catastrophic if a few deadlines are missed
- Challenge lies in how to meet most deadlines with minimal resource waste
- Challenge also lies in how to load-shed if become overloaded

Distributed OS

- Clustering
  - Use multiple small machines to handle large service demands
    - Cheaper than using one large machine
    - Better *potential* for reliability, incremental scalability, and absolute scalability
- Wide-area distributed systems
  - Allow use of geographically distributed resources
    - E.g., use of a local PC to access Internet services like Google, Amazon
    - Don't have to carry needed information with us
- Need OS support for communication and sharing of distributed resources
  - E.g., network file systems
- Want performance (although speedup is not metric of interest here), high reliability, and use of diverse resources

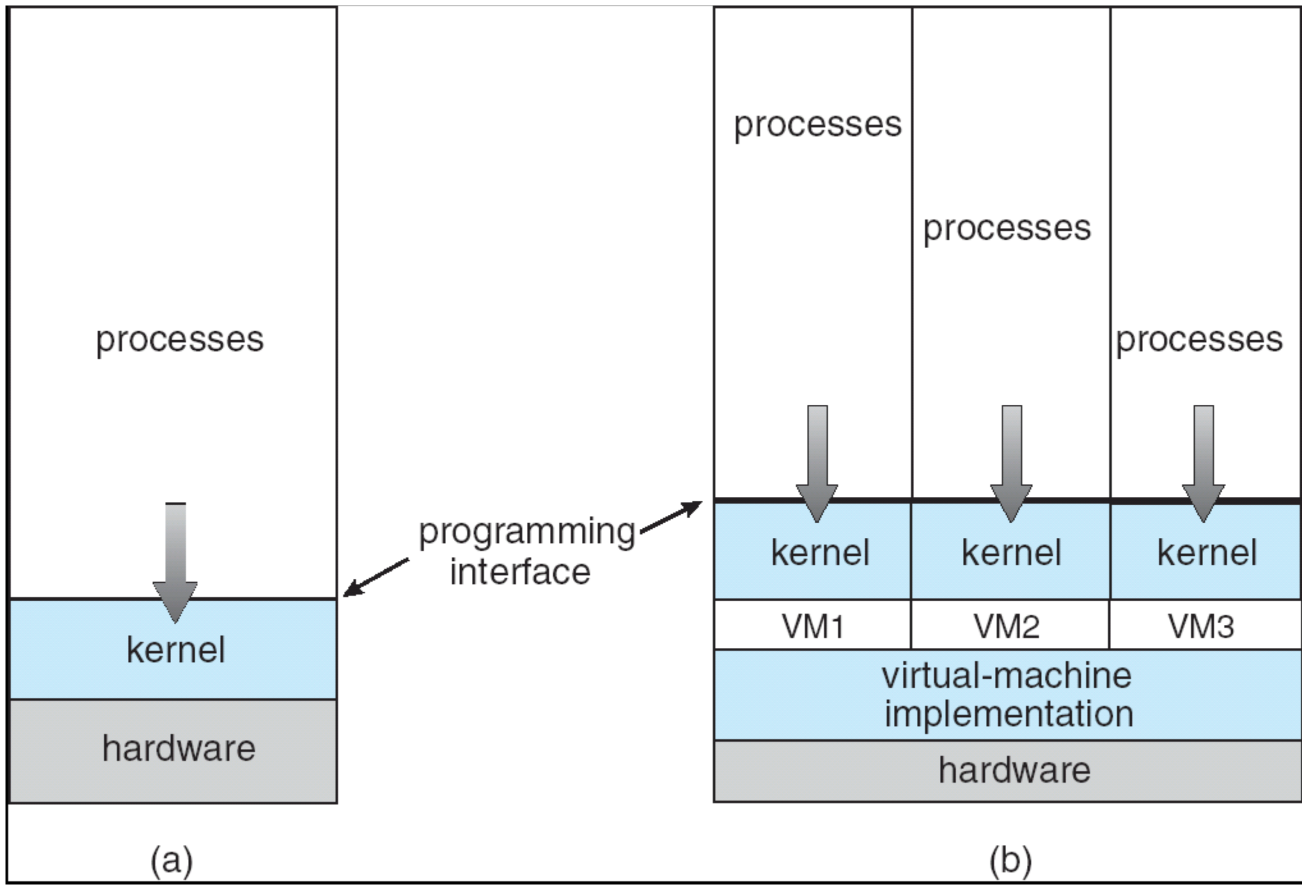
Embedded OS

- Pervasive computing
  - Right now, cell phones, automobiles, etc
  - Future, computational elements everywhere (e.g., clothes, inside your body)
- Characteristics
  - Constrained resources: slow CPU, small memories, no disk, etc.
  - What's new about this? Isn't this just like the old computers?
    - Well no, because we want to execute more powerful programs than before
  - How can we execute more powerful programs if our hardware is similar to old hardware?
    - Use many, many of them
    - Augment with services running on powerful machines
- OS support for power management, mobility, resource discovery, etc.

Virtual Machines and Hypervisors

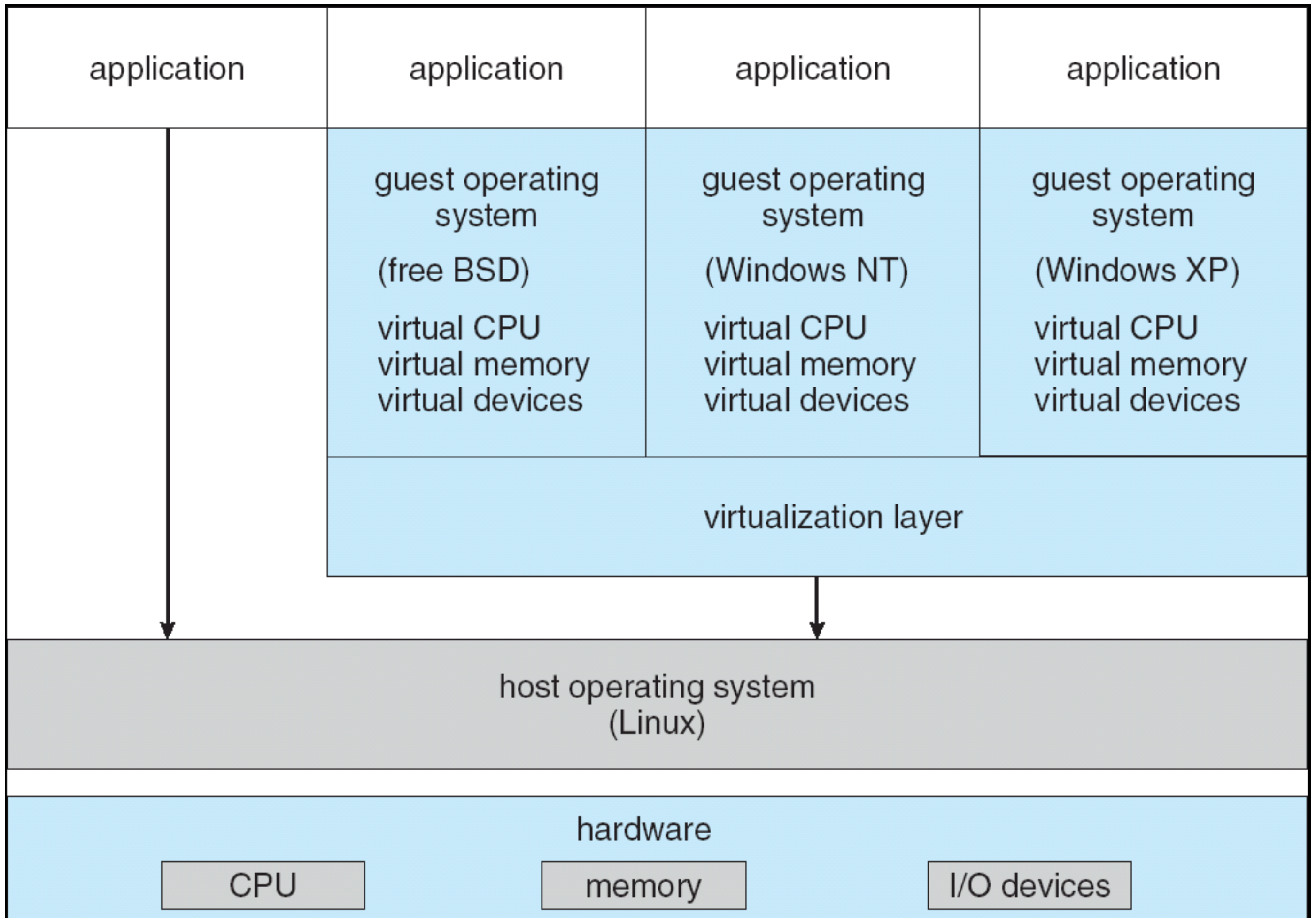
- Popular in the 60's and 70's, vanished in the 80's and 90's
- Idea: Partition a physical machine into a number of virtual machines
  - Each virtual machine behaves as a separate computer
  - Can support heterogeneous operating systems (called guest OSes)
  - Provides performance isolation and fault isolation
  - Facilitates virtual machine migration
  - Facilitates server consolidation
- Hypervisor or Virtual Machine Monitor
  - Underlying software that runs on the bare hardware
  - Manages multiple virtual machines

Virtual Machines and Hypervisors



Source: Silberschatz, Galvin, and Gagne

Virtual Machines: Another Architecture



Source: Silberschatz, Galvin, and Gagne

Next Time

Architectural refresher

Special Permissions

Requests must be accompanied by transcript and photo ID. Please write your email address, Rutgers ID, and section you want to join on your transcript

Provided that there is room in the sections, I will rank requests according to the following criteria:

Students with 105 credits or more

Students in CS major that are not repeating this course

Students in ECE major that are not repeating this course

Students in CS major that are repeating this course

Students in ECE major that are repeating this course

Other students

Cheating, turn-in, and special permission policies will be enforced strictly. There will be no if, but, … whatever