

Introduction

CS 416: Operating Systems Design, Fall 2009

Department of Computer Science
Rutgers University

<http://www.cs.rutgers.edu/~vinodg/teaching/416/>

Logistics

Instructor: Vinod Ganapathy

Office hours: Mondays, 4:45pm-5:45pm (Core 309)

TAs: Bogdan Branzoi, Rezwana Karim, Ana Paula Centeno

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

<http://www.cs.rutgers.edu/~vinodg/teaching/416/>

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

Start subject line with [cs416]

Emails without this tag will likely be ignored

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 and the OS/architecture interface/interaction

Learn a little bit about advanced topics (security, distributed, real-time)

Prerequisites:

(CS 113 or ECE 252) and (CS 211 or ECE 331)

What to expect:

We will cover core concepts and issues in lectures

In recitations, you and your TA will mostly talk about the programming assignments

Three large programming assignments in **C**; **randomly chosen demos**

1 Midterm and 1 Final

Course project for honors section students and graduate students

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 very hard to pass it!

Textbook and Topics

“Operating System Concepts” by Silberschatz, Galvin,
and Gagne.

Topics

Processes and threads

Processor scheduling

Synchronization

Virtual memory

File systems

I/O systems

Security and protection

Distributed systems

Grading

Rough guideline

Midterm 20% (15% for honors section)

Final 35% (30% for honors section)

Assignments 45% (40% for honors section)

Course project 15% (honors section only)

Class participation and pop-quizzes will be tie breakers

Programming assignments will be done in groups of at most three; demos might change grades. Exams and written homeworks are individual.

Cheating policy: Collaboration at the level of ideas is good. Copying code or words is not good.

Grading

Occasional written homeworks: Will not be graded

This requires **discipline** on your part to actually do them

This is critical to doing well on the exams

Solutions will be posted a couple of weeks after the posting of HWs

Assignment hand-ins **MUST** be on time

Late hand-ins will not be accepted

Programming assignments must be turned in electronically

Instructions will be posted on the web

We will promptly close the turn-in at the appointed time

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

Give you extra work so that you can try to improve your grade

Bump your grade up because you feel you deserve it

Give you an F if your grade should actually be a D

Review earlier assignments or exams to try to find extra points for you

So don't come asking!

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:

3. Students with 105 credits or more
4. Students in CS major that are not repeating this course
5. Students in ECE major that are not repeating this course
6. Students in CS major that are repeating this course
7. Students in ECE major that are repeating this course
8. Other students

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

Today

What is an Operating System?

Major OS components

A little bit of history

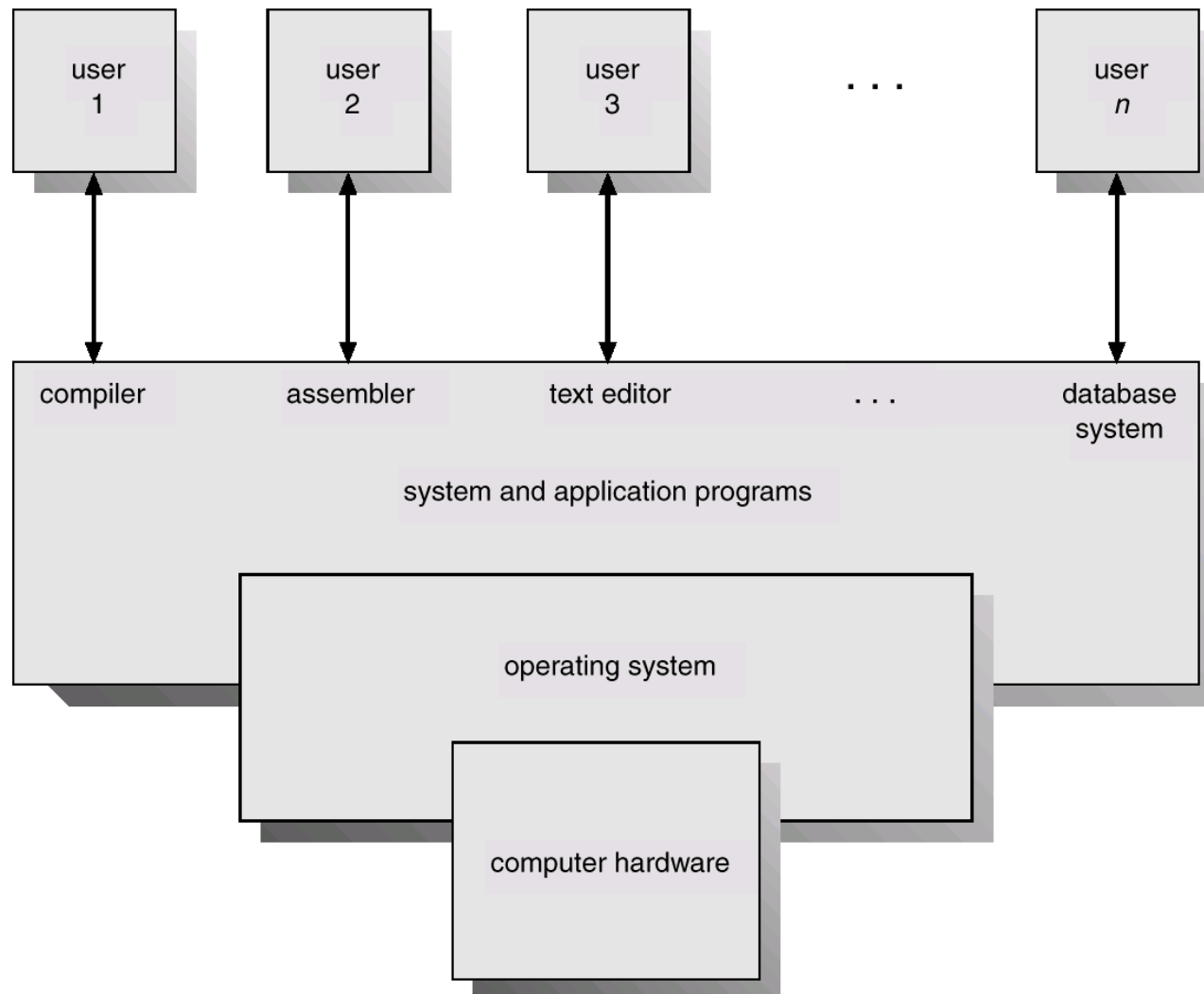
What Is An Operating System?

application (user)

operating system

hardware

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?

Safety

OS protects programs from each other

OS protects users from each other

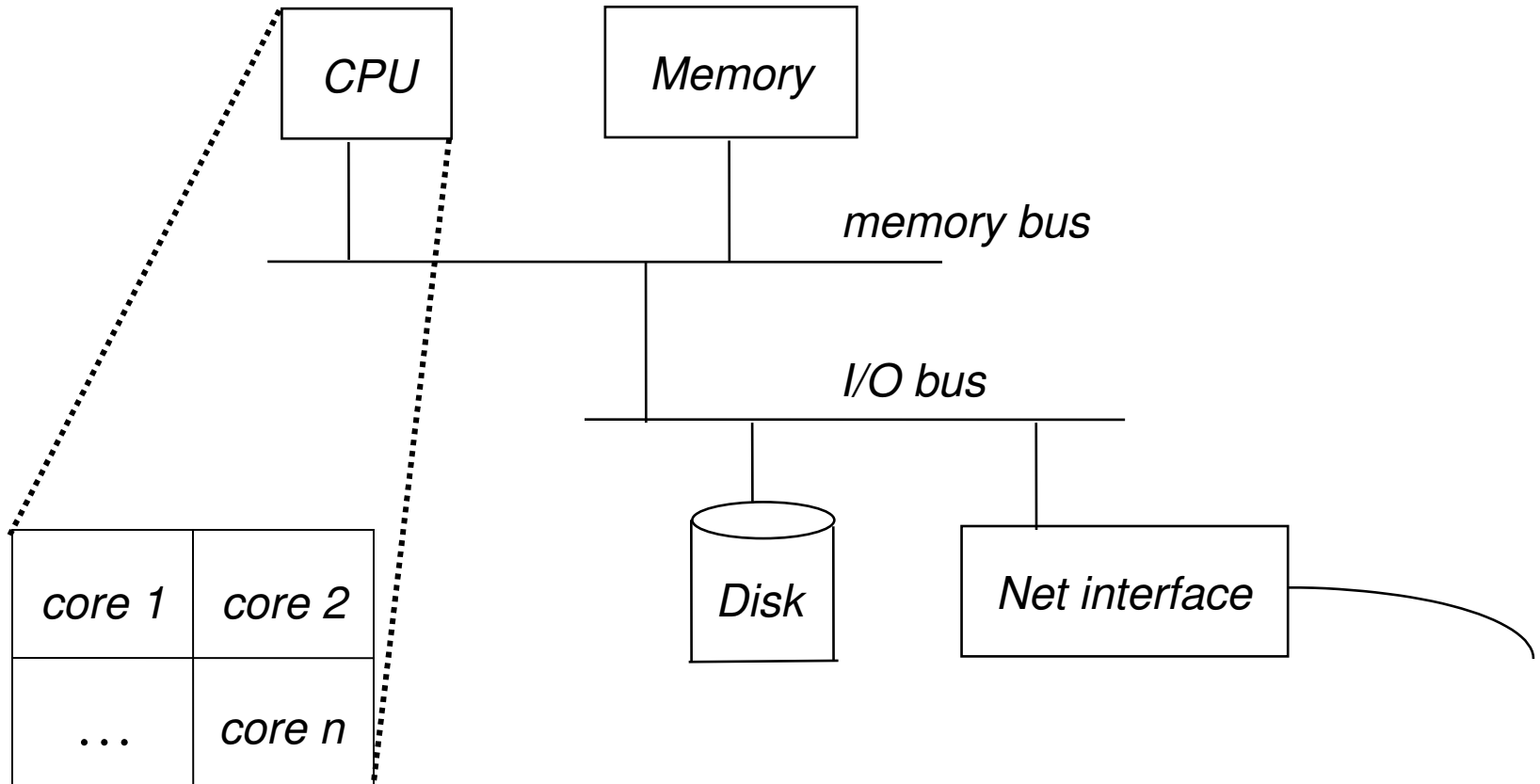
Mechanism and Policy

application (user)

operating system: *mechanism+policy*

hardware

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:	<i>execution context</i>	
<hr/>		create, kill, synch.
operating system:	<i>thread</i>	
<hr/>		context switch
hardware:	<i>processor</i>	

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*

operating system: *virtual memory*

hardware: *physical memory*

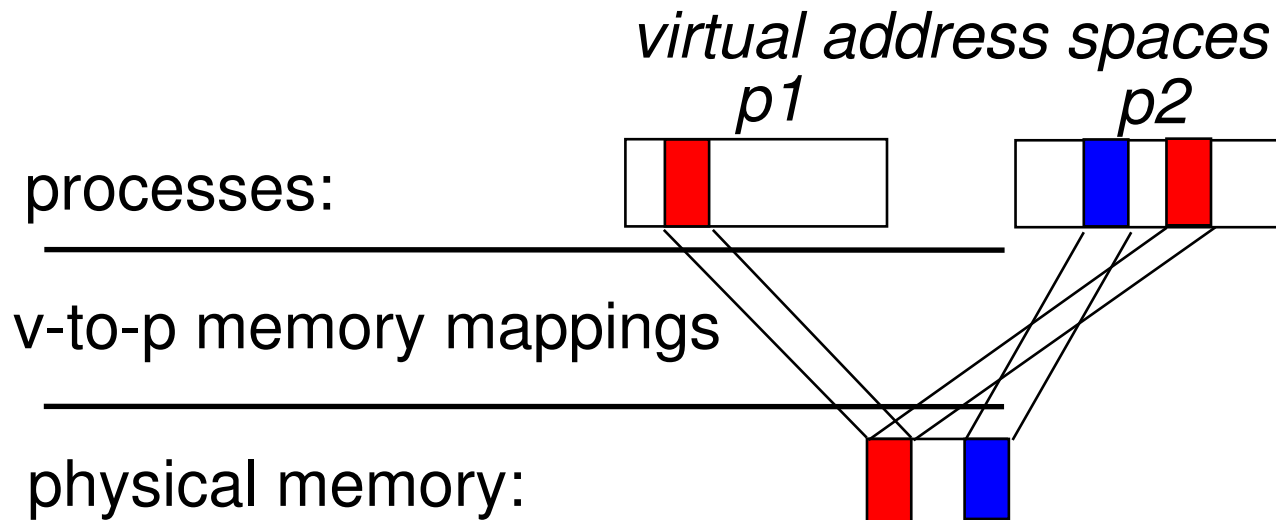
virtual addresses

physical addresses

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, file buffer cache, etc.

Policy:

Sharing vs. protection?

Which block to allocate?

File buffer cache management?

Communication Abstraction: Messaging

Message passing is a communication abstraction:
illusion of reliable (sometimes 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:	<i>read character from device</i>	naming, protection, read,write
<hr/>		
operating system:	<i>character & block API</i>	hardware-specific PIO, interrupt handling, or DMA
<hr/>		
hardware:	<i>keyboard, mouse, etc.</i>	

Devices

Mechanisms

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

Buffering

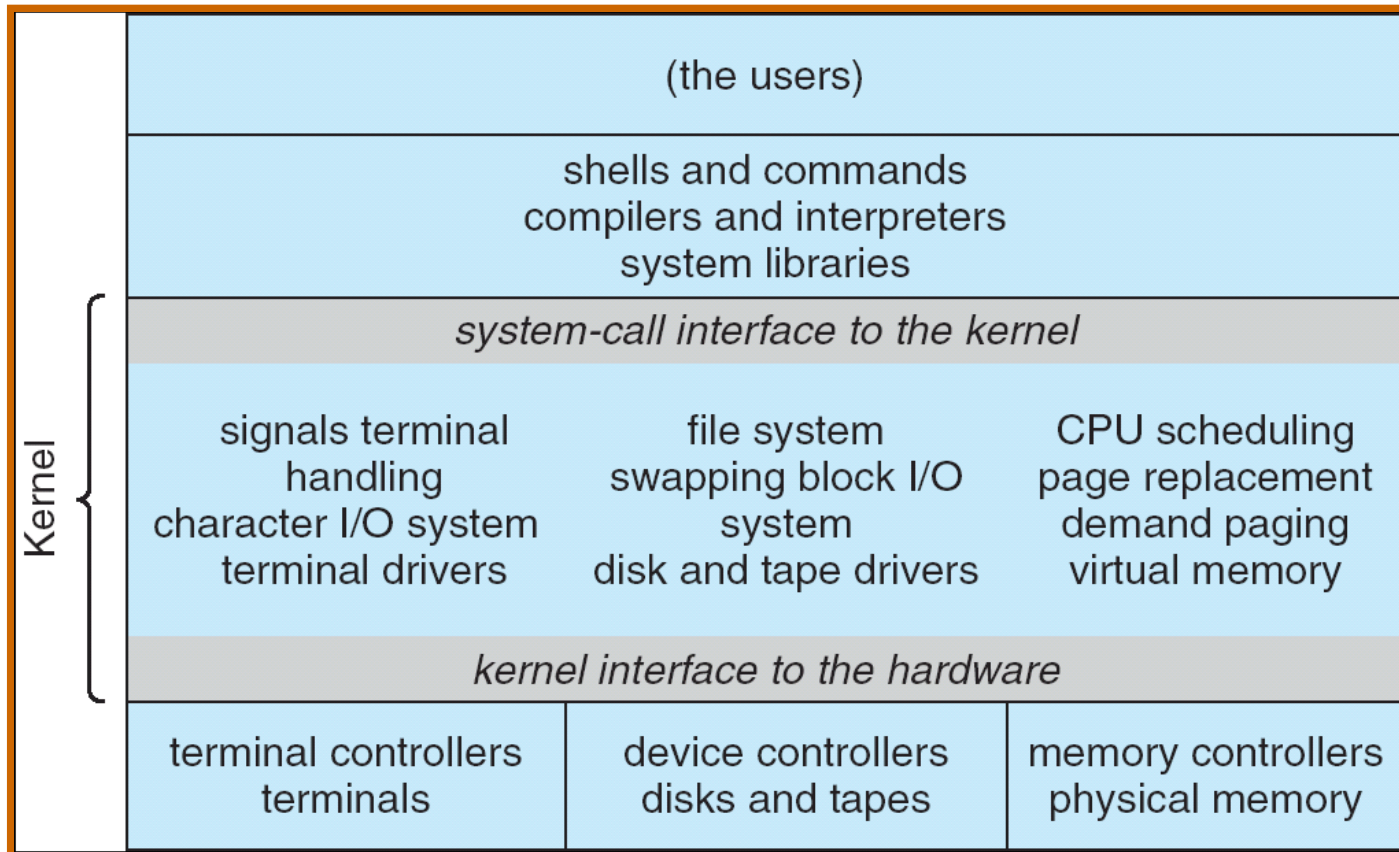
Policies

Protection

Sharing?

Scheduling?

UNIX



Source: Silberschatz, Galvin, and Gagne 2005

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 OS'es?

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 (cool!)

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: **S**imultaneous **P**eripheral **O**peration **O**n-**L**ine

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

Improves both system throughput and response time

Multiprogramming

CPU's were still idle whenever executing program needs 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 web services

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 and PDAs

Future, computational elements everywhere

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

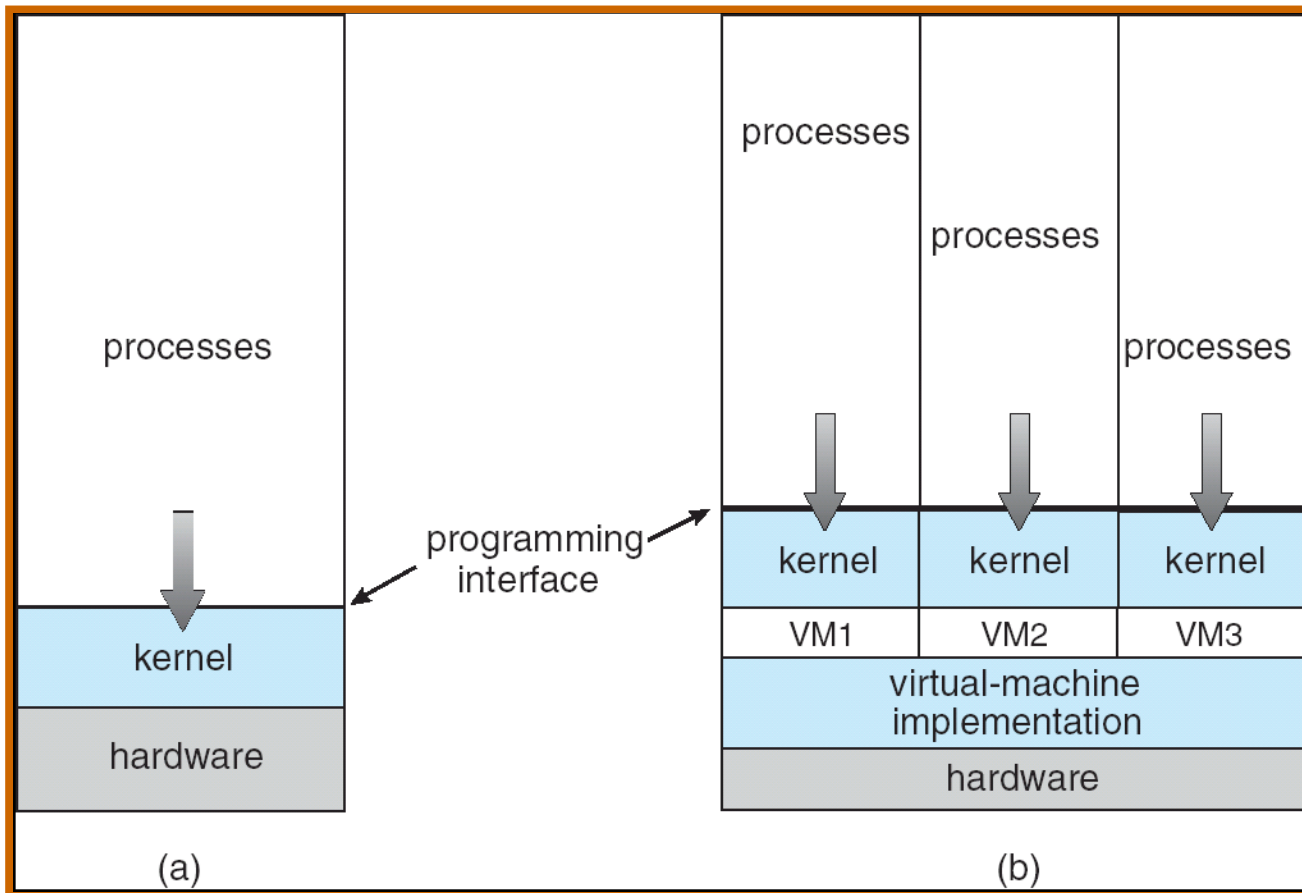
- Facilitates server consolidation

Hypervisor or Virtual Machine Monitor

- Underlying software that runs on the bare hardware

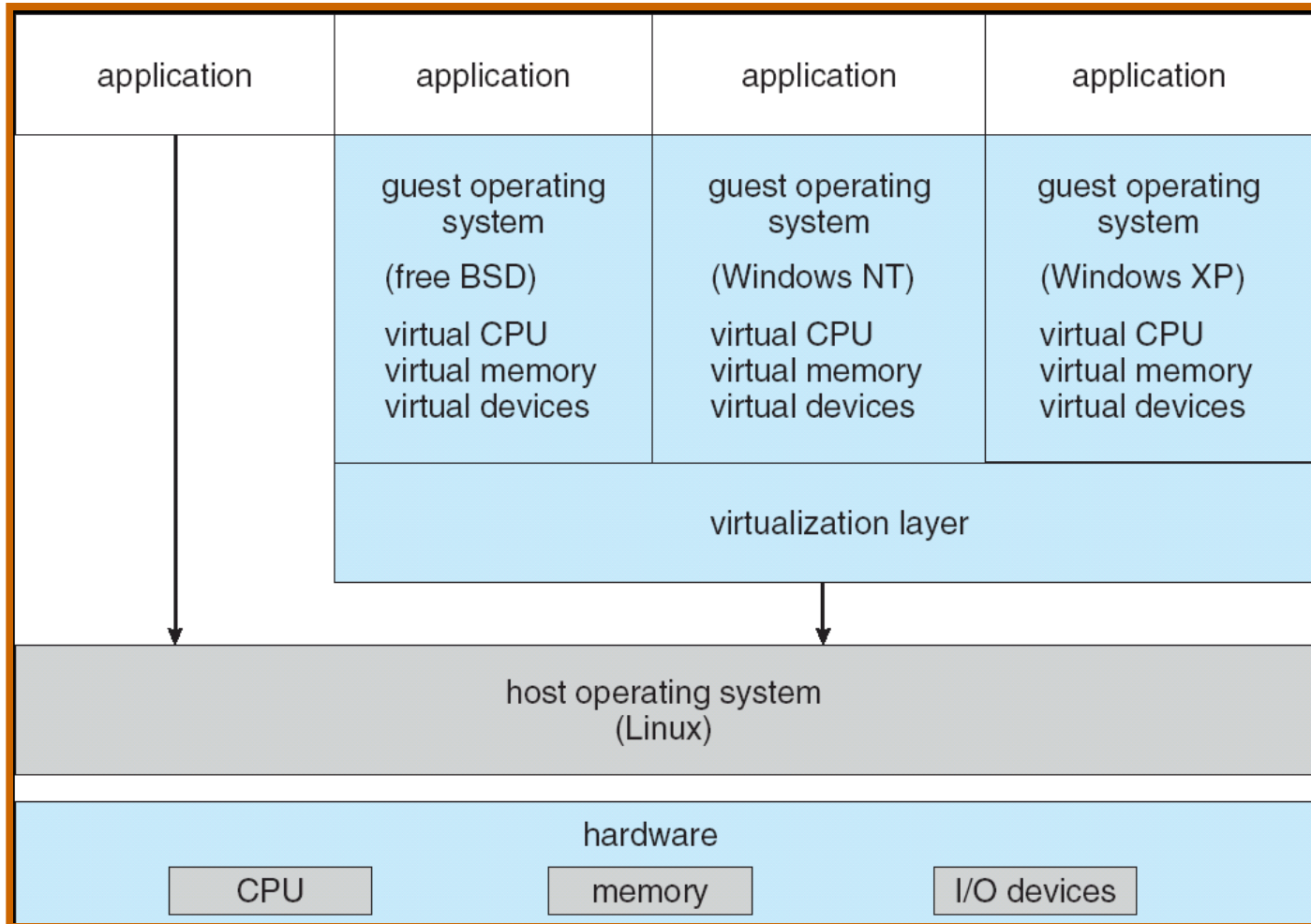
- Manages multiple virtual machines

Virtual Machines and Hypervisors



Source: Silberschatz, Galvin, and Gagne 2005

Virtual Machines: Another Architecture



Source: Silberschatz, Galvin, and Gagne 2005

Next Time

Architectural refresher