

ICS 143A - Principles of Operating Systems (Spring 2018)



Lecture 1 - Introduction and Overview

MWF 9:00- 9:50 p.m.

Prof. Nalini Venkatasubramanian

(nalini@uci.edu)

[lecture slides contains some content adapted from :

Silberschatz textbook authors, John Kubiatowicz (Berkeley)

**Anerson textbook, John Ousterhout(Stanford), previous slides
by Prof. Ardalan Sani, <http://www-inst.eecs.berkeley.edu/~cs162/> and others)**

]

ICS 143A Spring 2018 Staff



Instructor:

Prof. Nalini Venkatasubramanian (Venkat)
(nalini@uci.edu)

Teaching Assistants:

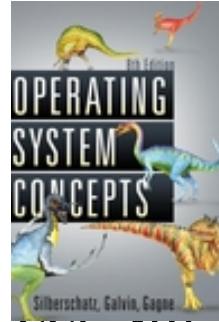
Kyle Benson(kebenson@uci.edu)
Jia Chen (jiac5@uci.edu)

Readers

Daokun Jiang
Yingtong Liu
TBD?

Course logistics and details

- Course Web page -
 - | <http://www.ics.uci.edu/~ics143>
- Lectures - MWF 9:00-9:50am, HSLH 100A
- Discussions
 - | Friday 10-10:50; 11-11:50 (ICS 174); 12:00-12:50 p.m (SSL 48)
- ICS 143 Textbook:
 - | Operating System Concepts – 9th Edition, Silberschatz, Galvin and Gagne, Addison-Wesley Inc.
 - | (Eighth, Seventh, Sixth and Fifth editions, and Java Versions are fine).
- Other Suggested Books
 - | Modern Operating Systems, by Tanenbaum
 - | Principles of Operating Systems, L.F. Bic and A.C. Shaw
 - | Operating Systems: Principles and Practice, by T. Anderson and M. Dahlin (second edition)



Course logistics and details



Homeworks and Assignments

- 4 written homeworks in the quarter
- 1 programming assignment (knowledge of C++ or Java required).
 - Handed out at midterm; submit/demo during Finals Week
 - Multistep assignment – don't start in last week of classes!!!
- Late homeworks will not be accepted.
- All submissions will be made using the EEE Dropbox for the course

Tests

- Midterm - tentatively Tuesday, Week 6 in class
- Final Exam - as per UCI course catalog

ICS 143 Grading Policy



- Homeworks - 30%
- Midterm - 30% of the final grade,
 - Tentatively Wed, Week 6 in class
- Final exam - 40% of the final grade
 - Per UCI course catalog
- Final assignment of grades will be based on a curve.

Lecture Schedule



Week 1:

- Introduction to Operating Systems, Computer System Structures, Operating System Structures

Week 2 : Process Management

- Processes and Threads, CPU Scheduling

Week 3: Process Management

- CPU Scheduling, Process Synchronization

Week 4: Process Management

- Process Synchronization

Week 5: Process Management

- Process Synchronization, Deadlocks

Course Schedule



Week 6 - Storage Management

- Midterm revision, exam, Memory Management

Week 7 - Storage Management

- Memory Mangement, Virtual Memory

Week 8 – Storage Management

- Virtual Memory, FileSystems

Week 9 - FileSystems

- FileSystems Implementation, I/O subsystems

Week 10 – I/O Subsystems

- Case study – UNIX, WindowsNT, course revision and summary.

Introduction



- What is an operating system?
- Operating Systems History
 - Simple Batch Systems
 - Multiprogrammed Batch Systems
 - Time-sharing Systems
 - Personal Computer Systems
- Parallel and Distributed Systems
- Real-time Systems

What is an Operating System?



- An OS is a program that acts an intermediary between the user of a computer and computer hardware.
- Major cost of general purpose computing is software.
 - OS simplifies and manages the complexity of running application programs efficiently.

Computer System Components



Hardware

- Provides basic computing resources (CPU, memory, I/O devices).

Operating System

- Controls and coordinates the use of hardware among application programs.

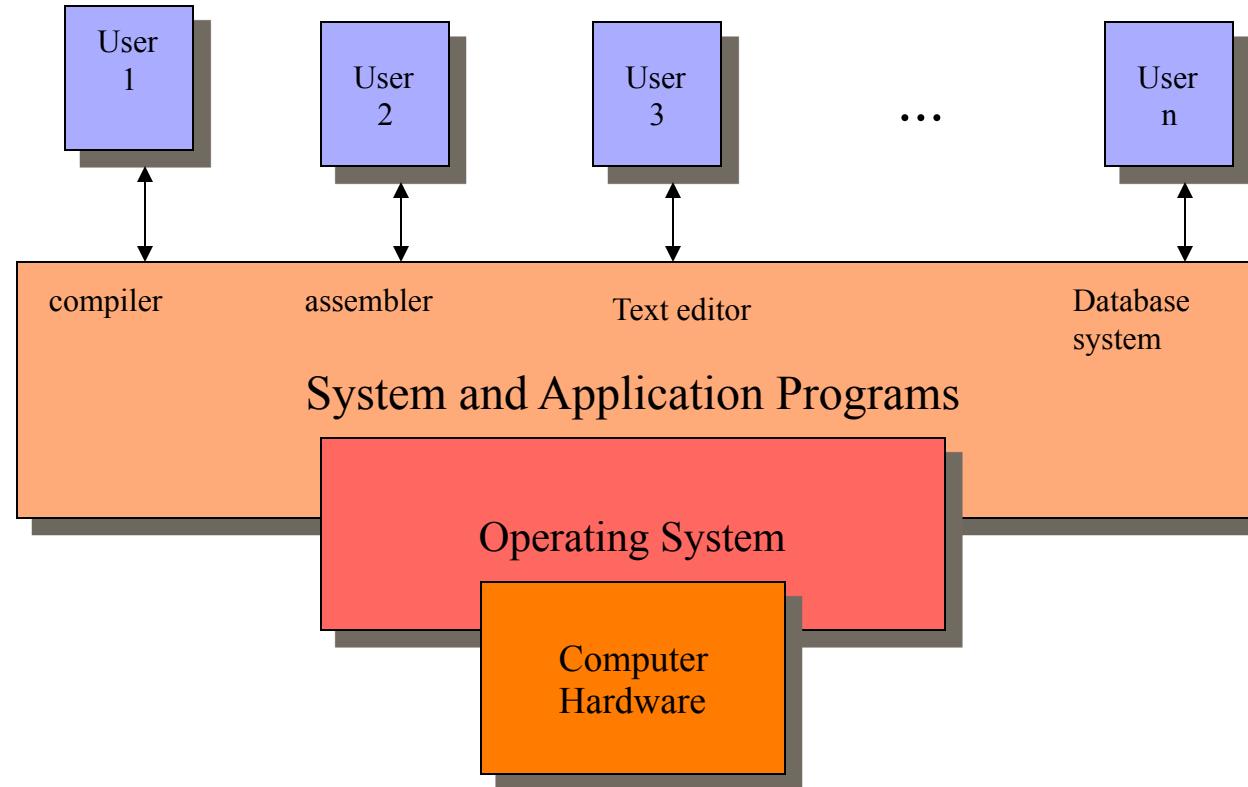
Application Programs

- Solve computing problems of users (compilers, database systems, video games, business programs such as banking software).

Users

- People, machines, other computers

Abstract View of System



Operating System Views



Resource allocator

- | to allocate resources (software and hardware) of the computer system and manage them efficiently.

Control program

- | Controls execution of user programs and operation of I/O devices.

Kernel

- | The program that executes forever (everything else is an application with respect to the kernel).

Operating system roles



- **Referee**

- Resource allocation among users, applications
- Isolation of different users, applications from each other
- Communication between users, applications

- **Illusionist**

- Each application appears to have the entire machine to itself
- Infinite number of processors, (near) infinite amount of memory, reliable storage, reliable network transport

- **Glue**

- Libraries, user interface widgets, ...
- Reduces cost of developing software

Example: file systems



- Referee
 - Prevent users from accessing each other's files without permission
- Illusionist
 - Files can grow (nearly) arbitrarily large
 - Files persist even when the machine crashes in the middle of a save
- Glue
 - Named directories, printf, ...

Goals of an Operating System



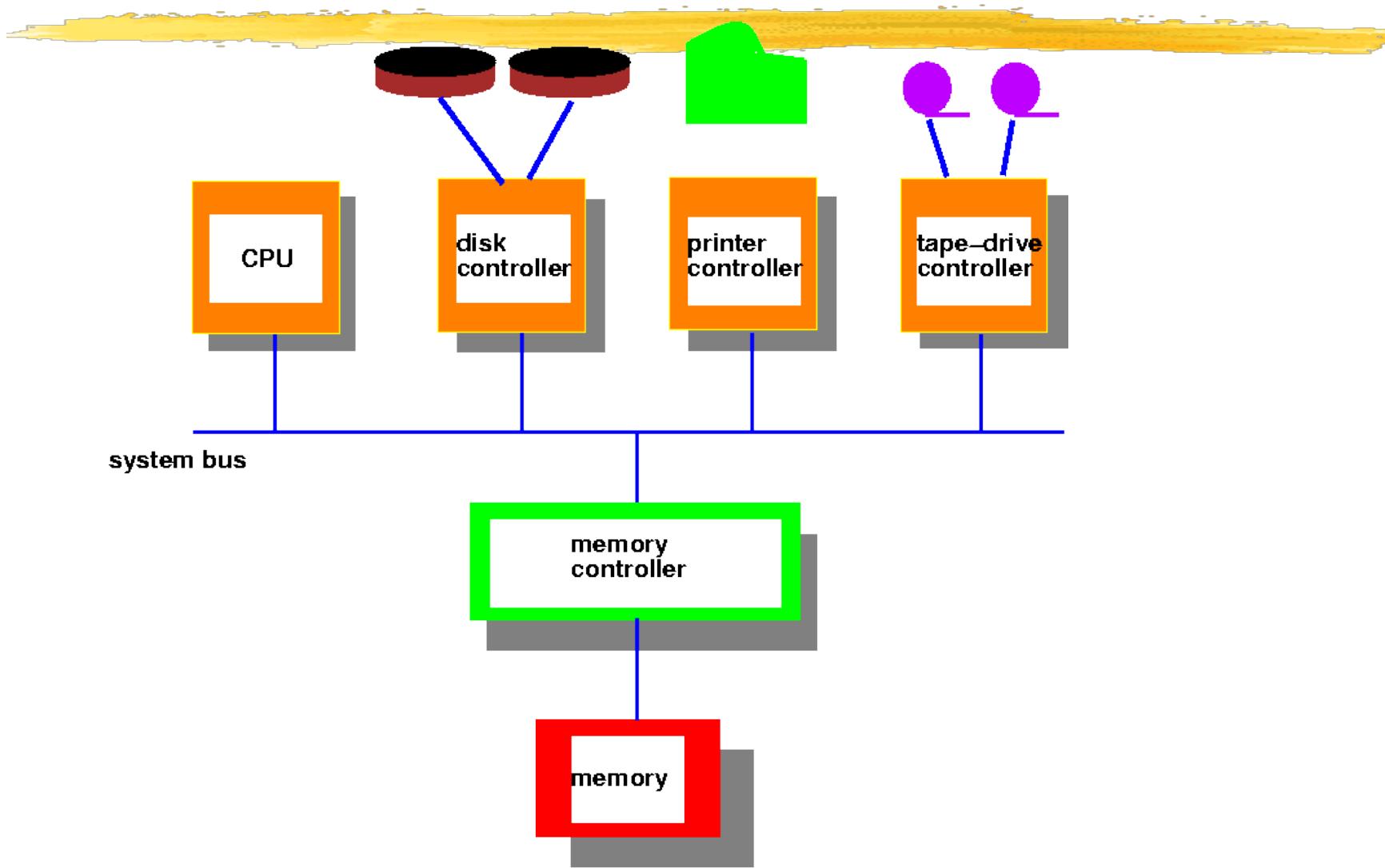
- Simplify the execution of user programs and make solving user problems easier.
- Use computer hardware efficiently.
 - Allow sharing of hardware and software resources.
- Make application software portable and versatile.
- Provide isolation, security and protection among user programs.
- Improve overall system reliability
 - error confinement, fault tolerance, reconfiguration.

Why should I study Operating Systems?

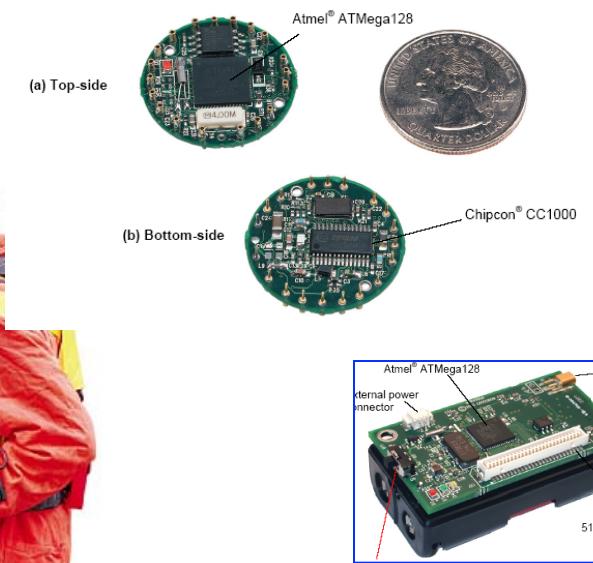
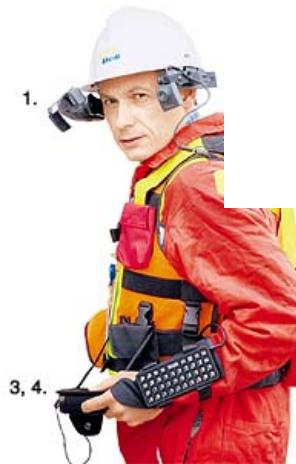
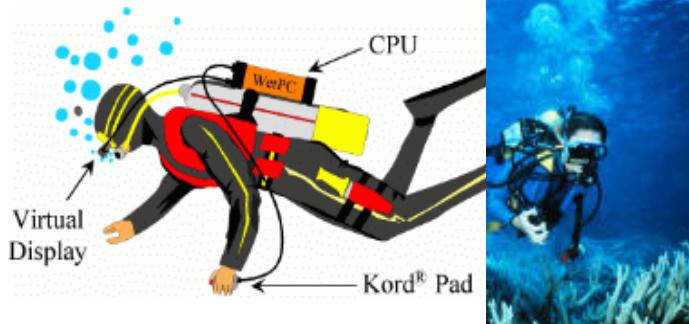


- | Need to understand interaction between the hardware and applications
 - | New applications, new hardware..
 - | Inherent aspect of society today
- | Need to understand basic principles in the design of computer systems
 - | efficient resource management, security, flexibility
- | Increasing need for specialized operating systems
 - | e.g. embedded operating systems for devices - cell phones, sensors and controllers
 - | real-time operating systems - aircraft control, multimedia services

Computer System Architecture (traditional)



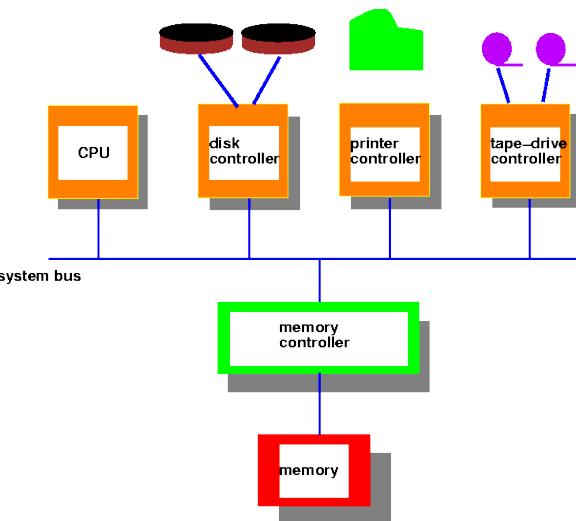
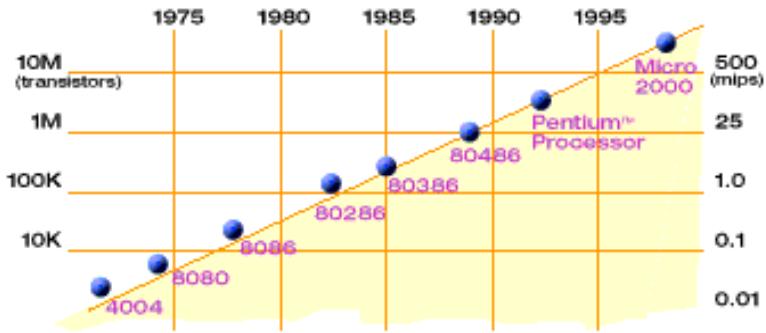
Systems Today



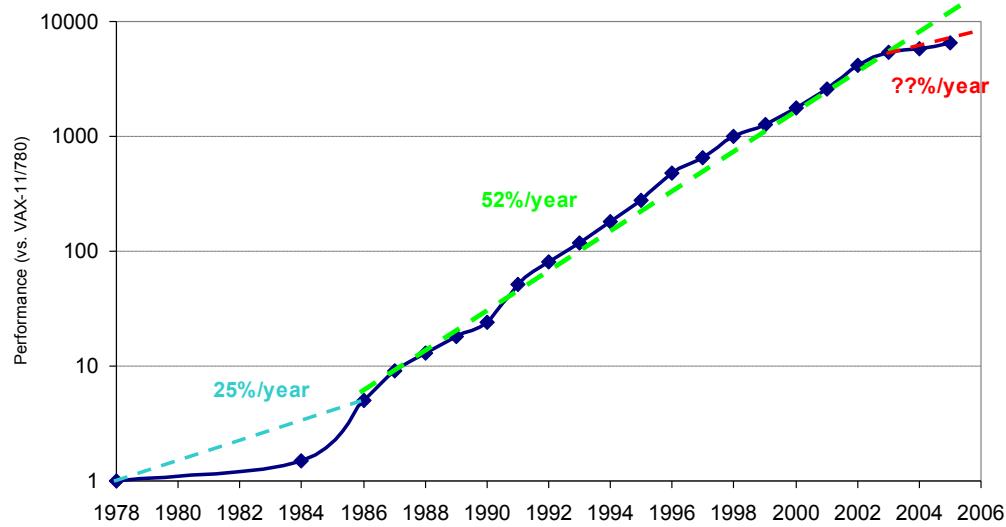
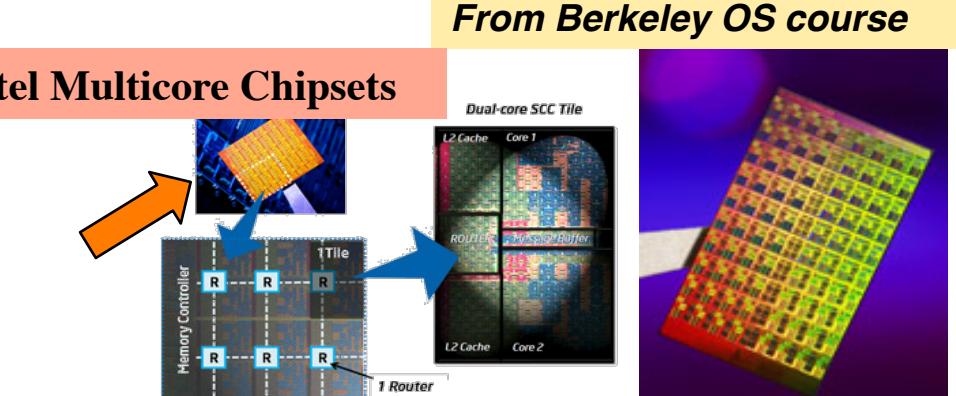


Hardware Complexity Increases

Moore's Law: 2X transistors/
Chip Every 1.5 years



Intel Multicore Chipsets



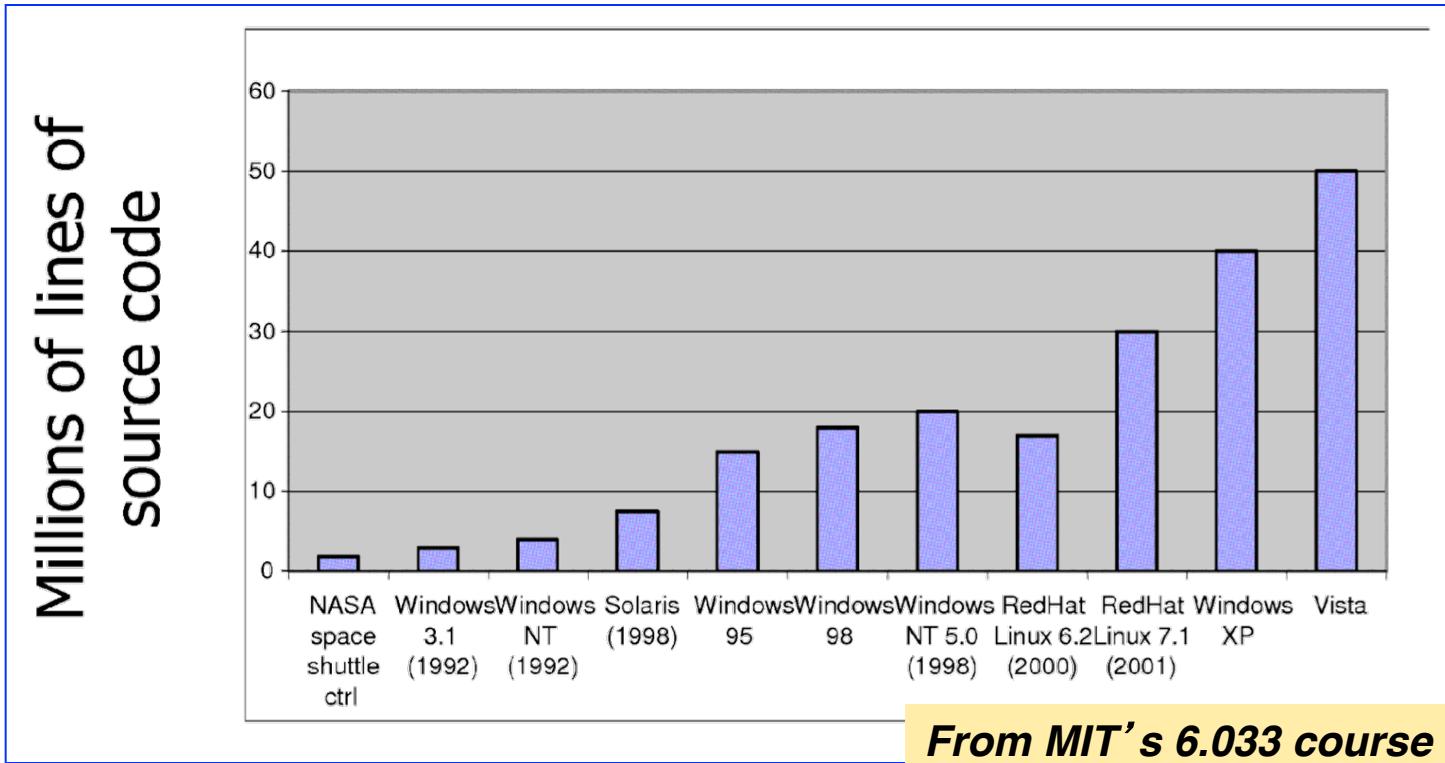
Principles of Operating Systems -

From Hennessy and Patterson, *Computer Architecture: A Quantitative Approach*, 4th edition, Sept. 15, 2006

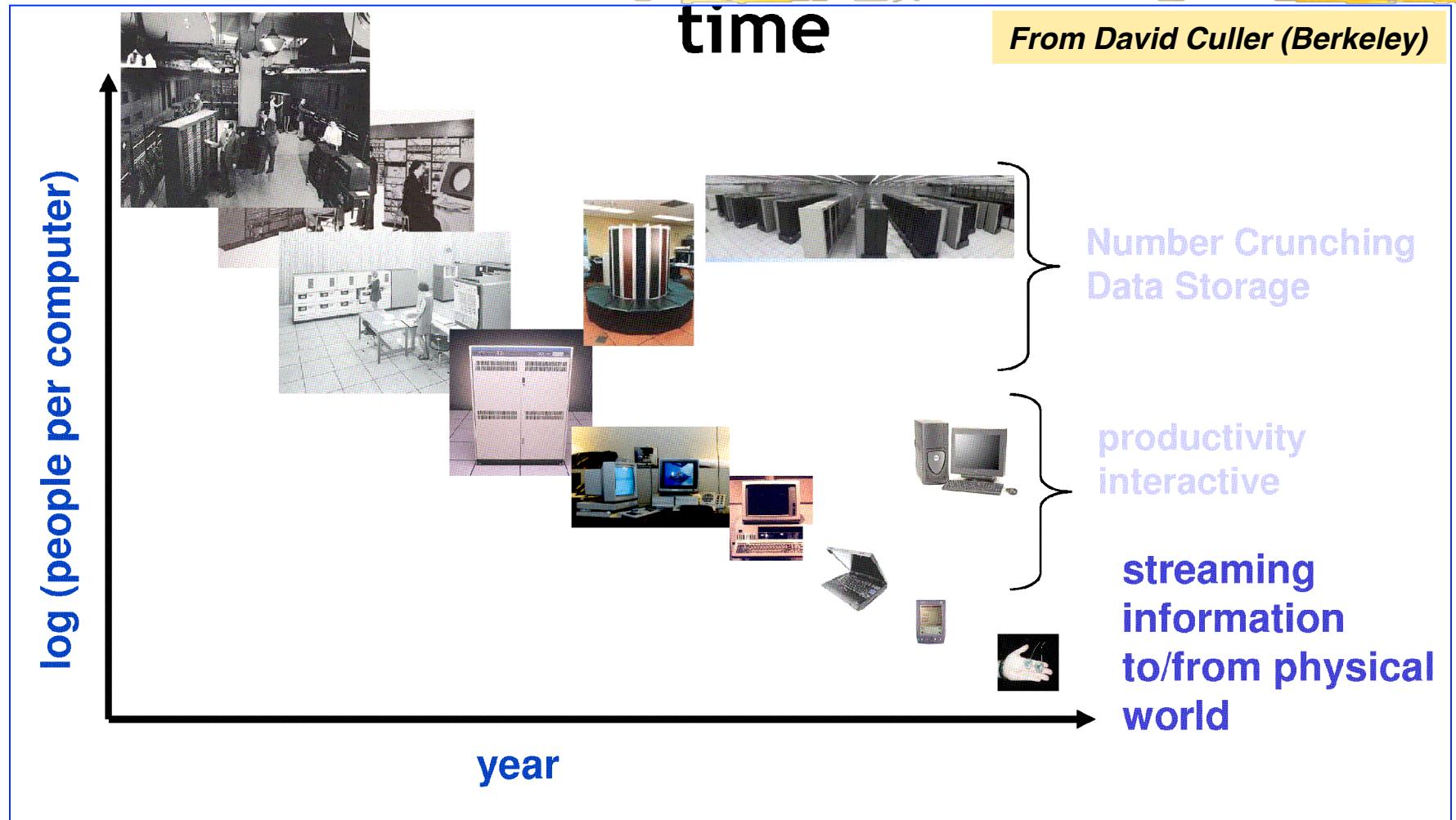
OS needs to keep pace with hardware improvements

| | 1981 | 1997 | 2014 | Factor (2014/1981) |
|--------------------------------|---------------------|------------------------|-----------------------|-----------------------|
| Uniprocessor speed (MIPS) | 1 | 200 | 2500 | 2.5K |
| CPUs per computer | 1 | 1 | 10+ | 10+ |
| \$/Processor MIPS | \$100K | \$25 | \$0.20 | 500K |
| DRAM Capacity (MiB)/\$ | 0.002 | 2 | 1K | 500K |
| Disk Capacity (GiB)/\$ | 0.003 | 7 | 25K | 10M |
| Home Internet | 300 bps | 256 Kbps | 20 Mbps | 100K |
| Machine room network | 10 Mbps (shared) | 100 Mbps (switched) | 10 Gbps (switched) | 1000 |
| Ratio of users to computers | 100:1 | 1:1 | 1:several | 100+ |

Software Complexity Increases



People-to-Computer Ratio Over Time



Operating System Spectrum



Monitors and Small Kernels

- | special purpose and embedded systems, real-time systems

Batch and multiprogramming

Timesharing

- | workstations, servers, minicomputers

Personal Computing Systems

Desktops and Laptops

Mobile Platforms/devices (of all sizes)

Parallel and Distributed Systems

Servers in the cloud

Early Systems - Bare Machine (1950s)

Hardware – *expensive* ; Human – *cheap*

Structure

- | Large machines run from console
- | Single user system
 - Programmer/User as operator
- | Paper tape or punched cards

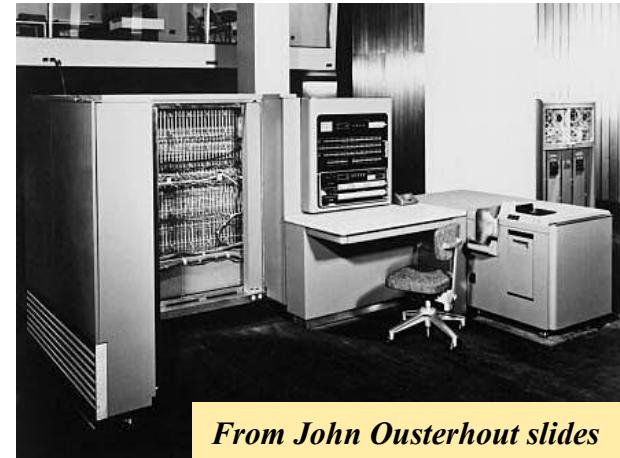
Early software

- | Assemblers, compilers, linkers, loaders, device drivers, libraries of common subroutines.

Secure execution

Inefficient use of expensive resources

- | Low CPU utilization, high setup time.



From John Ousterhout slides

Simple Batch Systems (1960's)

- Reduce setup time by batching jobs with similar requirements.
- Add a card reader, Hire an operator
 - User is NOT the operator
 - Automatic job sequencing
 - Forms a rudimentary OS.
- Resident Monitor
 - Holds initial control, control transfers to job and then back to monitor.
- Problem
 - Need to distinguish job from job and data from program.



From John Ousterhout slides

Supervisor/Operator Control

| Secure monitor that controls job processing

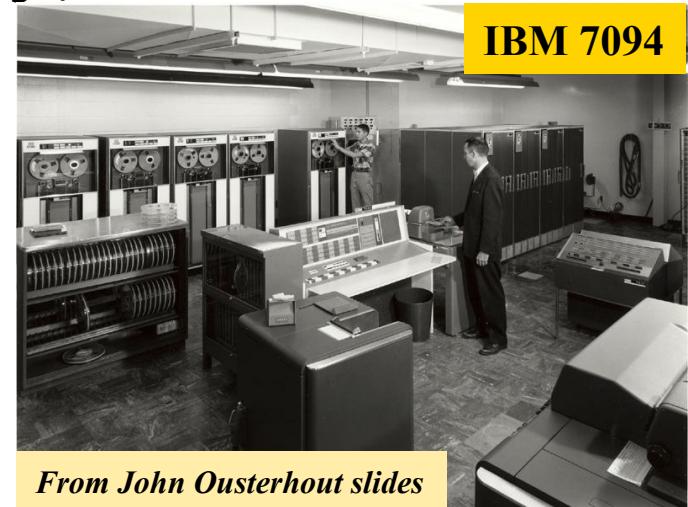
- | Special cards indicate what to do.
- | User program prevented from performing I/O

| Separate user from computer

- | User submits card deck
- | cards put on tape
- | tape processed by operator
- | output written to tape
- | tape printed on printer

| Problems

- | Long turnaround time - up to 2 DAYS!!!
- | Low CPU utilization
 - I/O and CPU could not overlap; slow mechanical devices.



Batch Systems - Issues



I Solutions to speed up I/O:

I Offline Processing

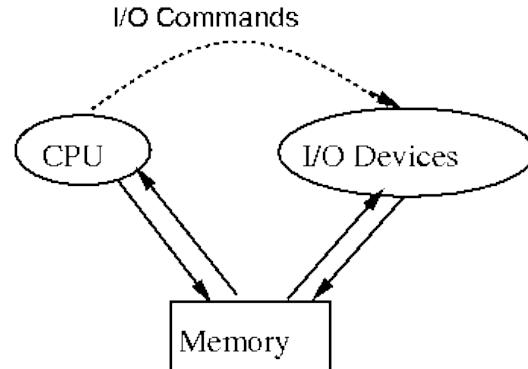
- | load jobs into memory from tapes, card reading and line printing are done offline.

I Spooling

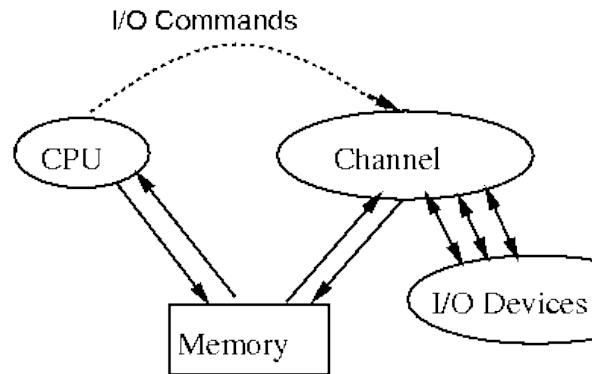
- | Use disk (random access device) as large storage for reading as many input files as possible and storing output files until output devices are ready to accept them.
- | Allows overlap - I/O of one job with computation of another.
- | Introduces notion of a job pool that allows OS choose next job to run so as to increase CPU utilization.

Speeding up I/O

- Direct Memory Access (DMA)



- Channels



Batch Systems - I/O completion



How do we know that I/O is complete?

Polling:

- | Device sets a flag when it is busy.
- | Program tests the flag in a loop waiting for completion of I/O.

Interrupts:

- | On completion of I/O, device forces CPU to jump to a specific instruction address that contains the interrupt service routine.
- | After the interrupt has been processed, CPU returns to code it was executing prior to servicing the interrupt.

Multiprogramming



- Use interrupts to run multiple programs simultaneously
 - ▀ When a program performs I/O, instead of polling, execute another program till interrupt is received.
- Requires secure memory, I/O for each program.
- Requires intervention if program loops indefinitely.
- Requires CPU scheduling to choose the next job to run.

Timesharing



Hardware – *getting cheaper*; Human – *getting expensive*

- Programs queued for execution in FIFO order.
- Like multiprogramming, but timer device interrupts after a quantum (timeslice).
 - Interrupted program is returned to end of FIFO
 - Next program is taken from head of FIFO
- Control card interpreter replaced by command language interpreter.

Timesharing (cont.)



- Interactive (action/response)
 - when OS finishes execution of one command, it seeks the next control statement from user.
- File systems
 - online filesystem is required for users to access data and code.
- Virtual memory
 - Job is swapped in and out of memory to disk.

Personal Computing Systems



Hardware – *cheap* ; Human – *expensive*

- Single user systems, portable.
- I/O devices - keyboards, mice, display screens, small printers.
- Laptops and palmtops, Smart cards, Wireless devices.
- Single user systems may not need advanced CPU utilization or protection features.
- Advantages:
 - user convenience, responsiveness, ubiquitous

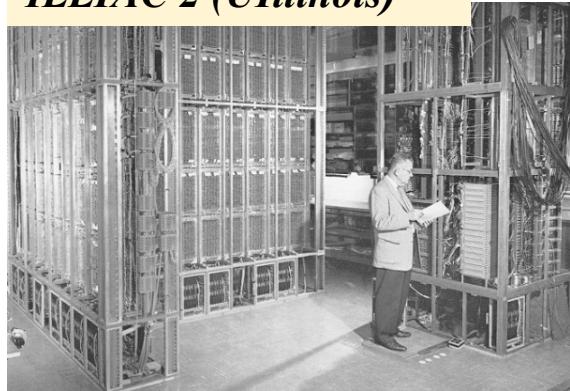
Parallel Systems



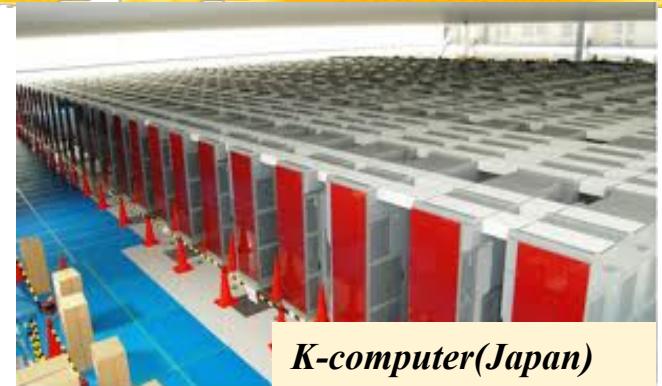
- Multiprocessor systems with more than one CPU in close communication.
- Improved Throughput, economical, increased reliability.
- Kinds:
 - Vector and pipelined
 - Symmetric and asymmetric multiprocessing
 - Distributed memory vs. shared memory
- Programming models:
 - Tightly coupled vs. loosely coupled ,message-based vs. shared variable

Parallel Computing Systems

ILLIAC 2 (UIllinois)



*Climate modeling,
earthquake
simulations, genome
analysis, protein
folding, nuclear fusion
research,*



K-computer(Japan)



Connection Machine (MIT)

Tianhe-1(China)



IBM Blue Gene

Distributed Systems



Hardware – *very cheap* ; Human – *very expensive*

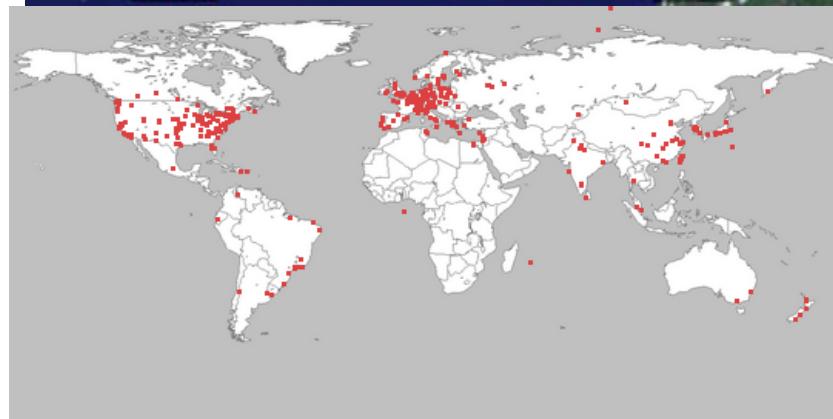
- Distribute computation among many processors.
- Loosely coupled -
 - no shared memory, various communication lines
- client/server architectures
- Advantages:
 - resource sharing
 - computation speed-up
 - reliability
 - communication - e.g. email
- Applications - digital libraries, digital multimedia

Distributed Computing Systems

Globus Grid Computing Toolkit



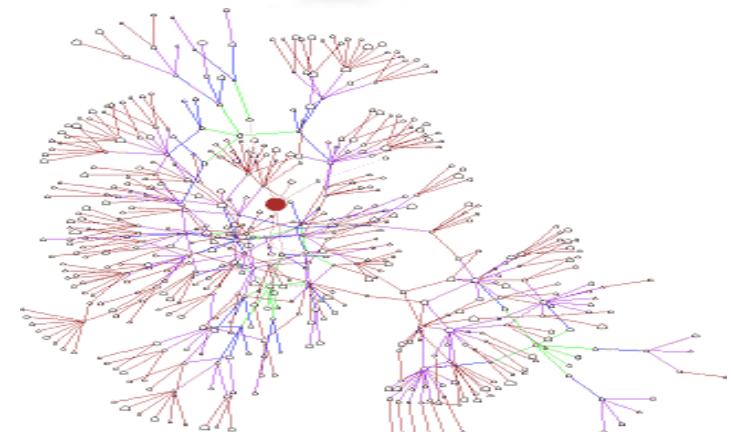
Cloud Computing Offerings



PlanetLab

Topics of Operating Systems
Lecture 1

Gnutella P2P Network



Real-time systems

- Correct system function depends on timeliness
- Feedback/control loops
- Sensors and actuators
- Hard real-time systems -
 - Failure if response time too long.
 - Secondary storage is limited
- Soft real-time systems -
 - Less accurate if response time is too long.
 - Useful in applications such as multimedia, virtual reality.



A personal computer today



interaction

- Super AMOLED display
- Capacitive touchscreen (multitouch)
- Audio (speaker, microphone)
- Vibration
- S pen

- 4G LTE
- NFC
- WiFi
- Bluetooth
- Infrared
- 64 GB internal storage (extended by microSD)
- Adreno 330 GPU
- Hexagon DSP
- Multimedia processor

- 13 MP front camera
- 2 MP back camera
- Accelerometer
- Gyroscope
- Proximity sensor
- Compass
- Barometer
- Temperature sensor
- Humidity sensor
- Gesture Sensor
- GPS

A personal computer today



- Super AMOLED display
- Capacitive touchscreen (multitouch)
- Audio (speaker, microphone)
- Vibration
- S pen
- 4G LTE
- NFC
- WiFi
- Bluetooth
- Infrared
- 64 GB internal storage (extended by microSD)
- Adreno 330 GPU
- Hexagon DSP
- Multimedia processor

- 13 MP front camera
- 2 MP back camera
- Accelerometer
- Gyroscope
- Proximity sensor
- Compass
- Barometer
- Temperature sensor
- Humidity sensor
- Gesture Sensor
- GPS

sensing

A personal computer today



- Super AMOLED display
- Capacitive touchscreen (multitouch)
- Audio (speaker, microphone)
- Vibration
- S pen

- 4G LTE
- NFC
- WiFi
- Bluetooth
- Infrared

connectivity

- 64 GB internal storage (extended by microSD)
- Adreno 330 GPU
- Hexagon DSP
- Multimedia processor

- 13 MP front camera
- 2 MP back camera
- Accelerometer
- Gyroscope
- Proximity sensor
- Compass
- Barometer
- Temperature sensor
- Humidity sensor
- Gesture Sensor
- GPS

A personal computer today



- Super AMOLED display
- Capacitive touchscreen (multitouch)
- Audio (speaker, microphone)
- Vibration
- S pen
- 4G LTE
- NFC
- WiFi
- Bluetooth
- Infrared

- 64 GB internal storage (extended by microSD)
- Adreno 330 GPU
- Hexagon DSP *acceleration*
- Multimedia processor

- 13 MP front camera
- 2 MP back camera
- Accelerometer
- Gyroscope
- Proximity sensor
- Compass
- Barometer
- Temperature sensor
- Humidity sensor
- Gesture Sensor
- GPS

Operating systems are everywhere



Operating systems are everywhere



Summary of lecture



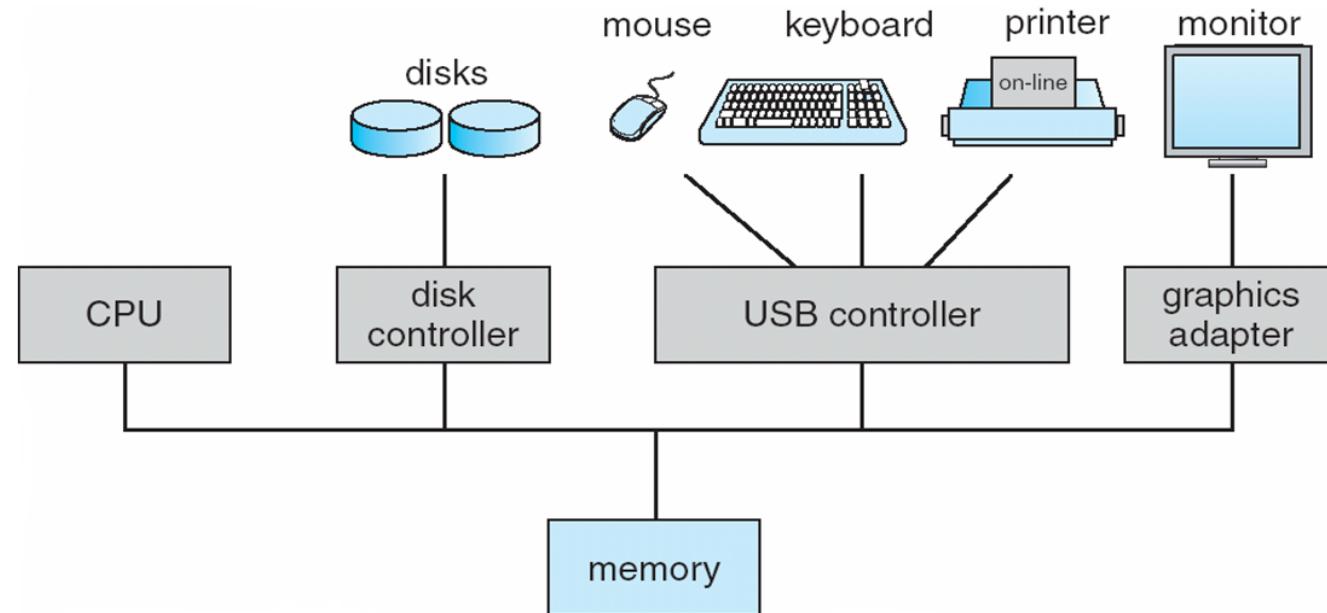
- What is an operating system?
- Early Operating Systems
- Simple Batch Systems
- Multiprogrammed Batch Systems
- Time-sharing Systems
- Personal Computer Systems
- Parallel and Distributed Systems
- Real-time Systems

Computer System & OS Structures

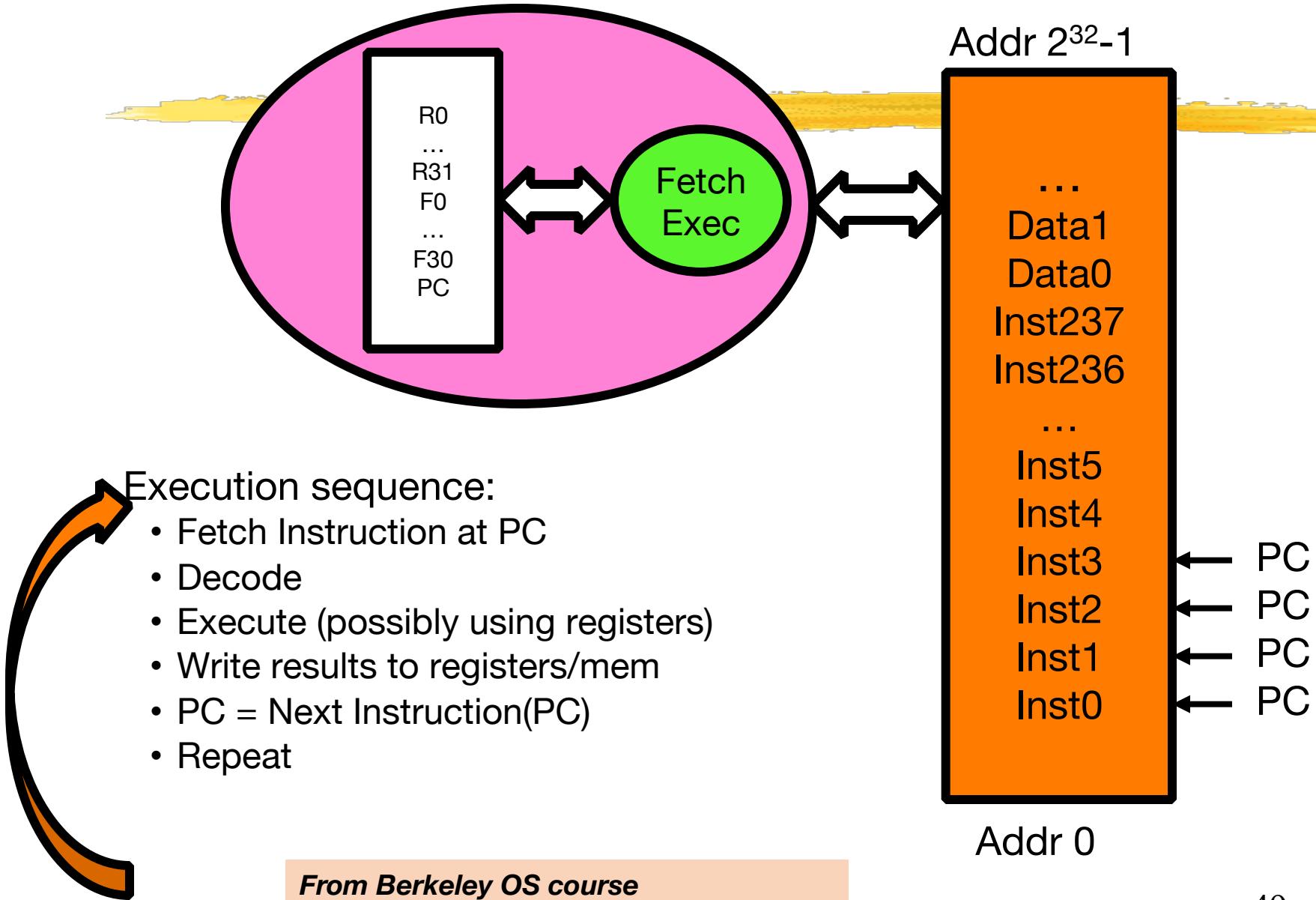


- Computer System Organization
- Operational Flow and hardware protection
- System call and OS services
- Storage architecture
- OS organization
- OS tasks
- Virtual Machines

Computer System **Organization**

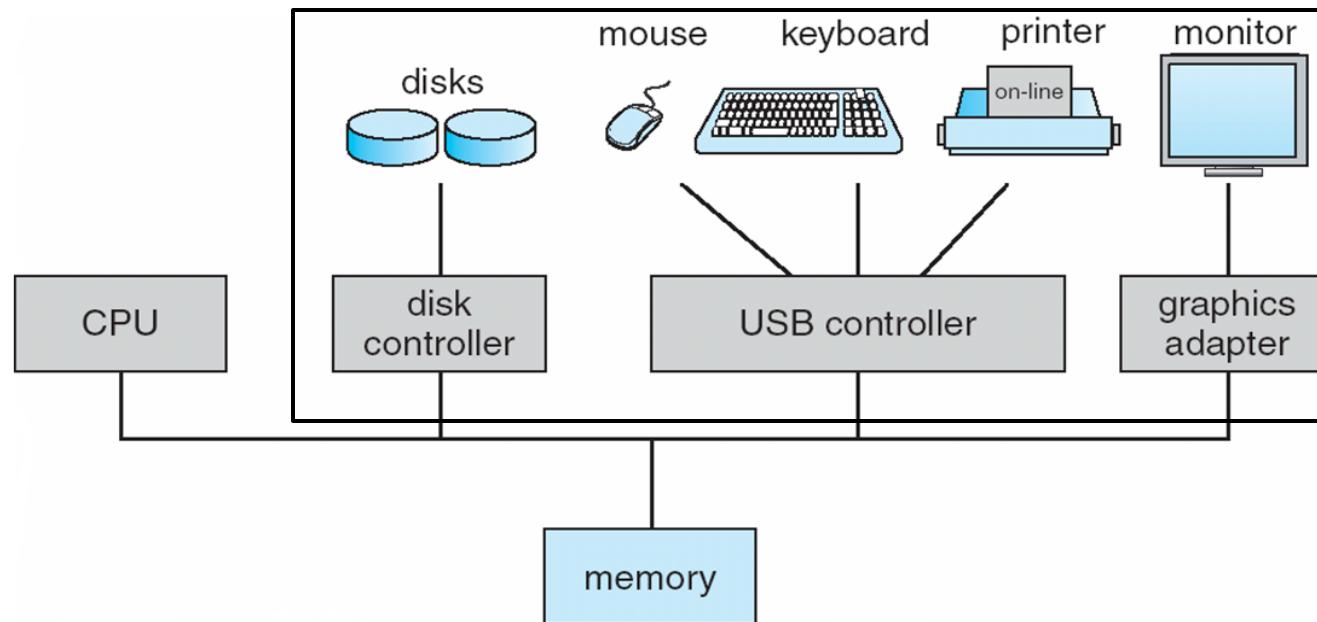


CPU execution



Computer System **Organization**

I/O devices



I/O devices



- I/O devices and the CPU execute concurrently.
- Each device controller is in charge of a particular device type
 - Each device controller has a local buffer. I/O is from the device to local buffer of controller
- CPU moves data from/to main memory to/from the local buffers

Interrupts

- **Interrupt** transfers control to the **interrupt service routine**
 - Interrupt Service Routine: Segments of code that determine action to be taken for interrupt.
- Determining the type of interrupt
 - Polling: same interrupt handler called for all interrupts, which then polls all devices to figure out the reason for the interrupt
 - Interrupt Vector Table: different interrupt handlers will be executed for different interrupts

| Interrupt Number | Address |
|------------------|---------|
| 0 | 0003h |
| 1 | 000Bh |
| 2 | 0013h |
| 3 | 001Bh |
| 4 | 0023h |
| 5 | 002Bh |
| 6 | 0033h |
| 7 | 003Bh |
| 8 | 0043h |
| 9 | 004Bh |
| 10 | 0053h |
| 11 | 005Bh |
| 12 | 0063h |
| 13 | 006Bh |
| 14 | 0073h |
| 15 | 007Bh |
| 16 | 0083h |
| 17 | 008Bh |
| 18 | 0093h |
| 19 | 009Bh |
| 20 | 00A3h |
| 21 | 00ABh |
| 22 | 00B3h |
| 23 | 00BBh |
| 24 | 00C3h |
| 25 | 00CBh |
| 26 | 00D3h |
| 27 | 00DBh |
| 28 | 00E3h |
| 29 | 00EBh |
| 30 | 00F3h |
| 31 | 00FBh |

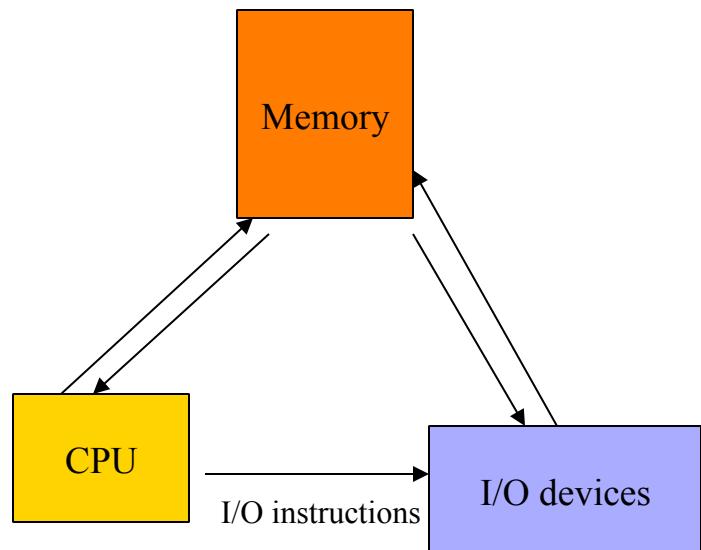
Interrupt handling



- OS preserves the state of the CPU
 - stores registers and the program counter (address of interrupted instruction).
- What happens to a new interrupt when the CPU is handling one interrupt?
 - Incoming interrupts can be disabled while another interrupt is being processed. In this case, incoming interrupts may be lost or may be buffered until they can be delivered.
 - Incoming interrupts can be masked (i.e., ignored) by software.
 - Incoming interrupts are delivered, i.e., nested interrupts.

Direct Memory Access (DMA)

- Typically used for I/O devices with a lot of data to transfer (in order to reduce load on CPU).
- Device controller transfers blocks of data from buffer storage directly to main memory without CPU intervention.
- Device controller interrupts CPU on completion of I/O



Process Abstraction



Process Abstraction



- Process: an *instance* of a program, running with limited rights

Process Abstraction and rights



- Process: an *instance* of a program, running with limited rights
- Address space: set of rights of a process
 - Memory that the process can access
- Other permissions the process has (e.g., which system calls it can make, what files it can access)

Hardware Protection



- CPU Protection:
 - Dual Mode Operation
 - Timer interrupts
- Memory Protection
- I/O Protection

How to limit process rights?



Should a process be able to execute any instructions?



Should a process be able to execute any instructions?

- No
 - Can alter system configuration
 - Can access unauthorized memory
 - Can access unauthorized I/O
 - etc.
- How to prevent?

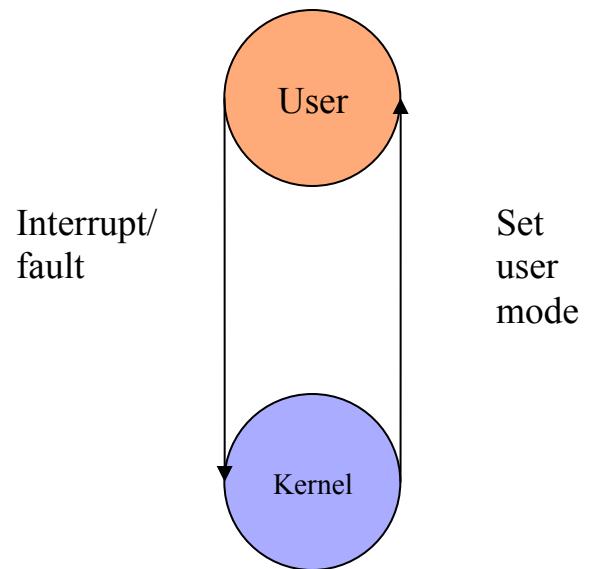
Dual-mode operation



- Provide hardware support to differentiate between at least two modes of operation:
 1. User mode -- execution done on behalf of a user.
 2. Kernel mode (monitor/supervisor/system mode) -- execution done on behalf of operating system.
- “Privileged” instructions are only executable in the kernel mode
- Executing privileged instructions in the user mode “traps” into the kernel mode

Dual-mode operation(cont.)

- Mode bit added to computer hardware to indicate the current mode: kernel(0) or user(1).
- When an interrupt or trap occurs, hardware switches to kernel mode.



CPU Protection



- How to prevent a process from executing indefinitely?

CPU Protection



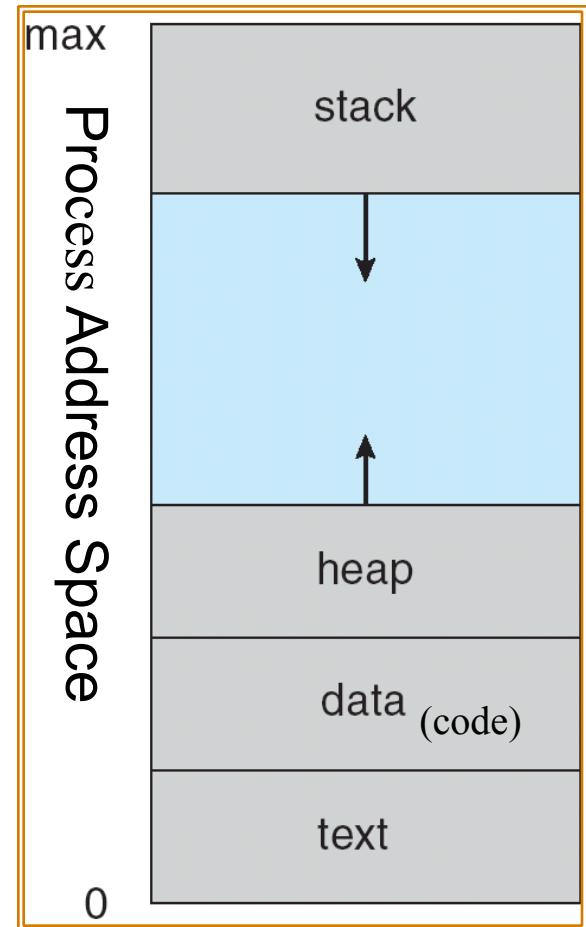
- Timer - interrupts computer after specified period to ensure that OS maintains control.
 - Timer is decremented every clock tick.
 - When timer reaches a value of 0, an interrupt occurs.
- Timer is commonly used to implement time sharing.
- Timer is also used to compute the current time.
- Programming the timer can only be done in the kernel since it requires privileged instructions.

How to isolate memory access?



Process address space

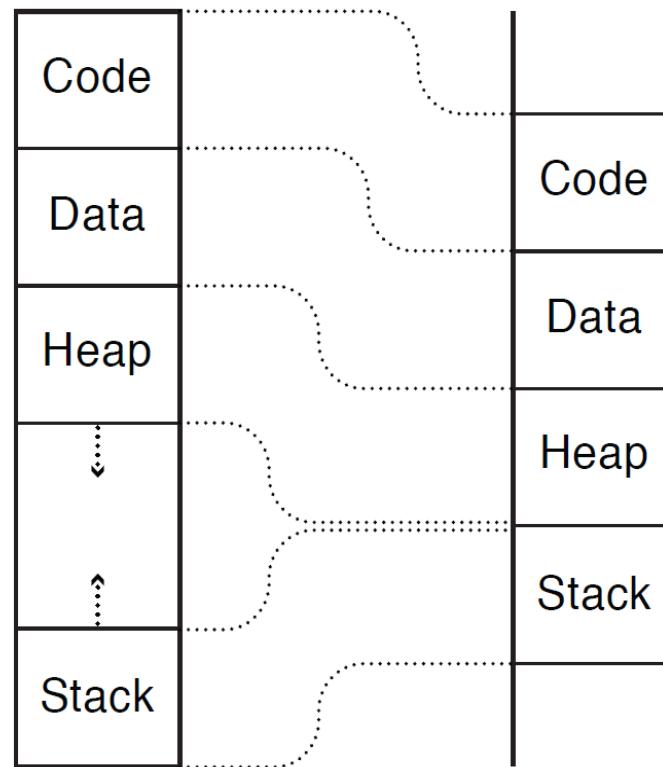
- Address space \Rightarrow the set of accessible addresses
 - For a 32-bit processor there are $2^{32} = 4$ billion addresses



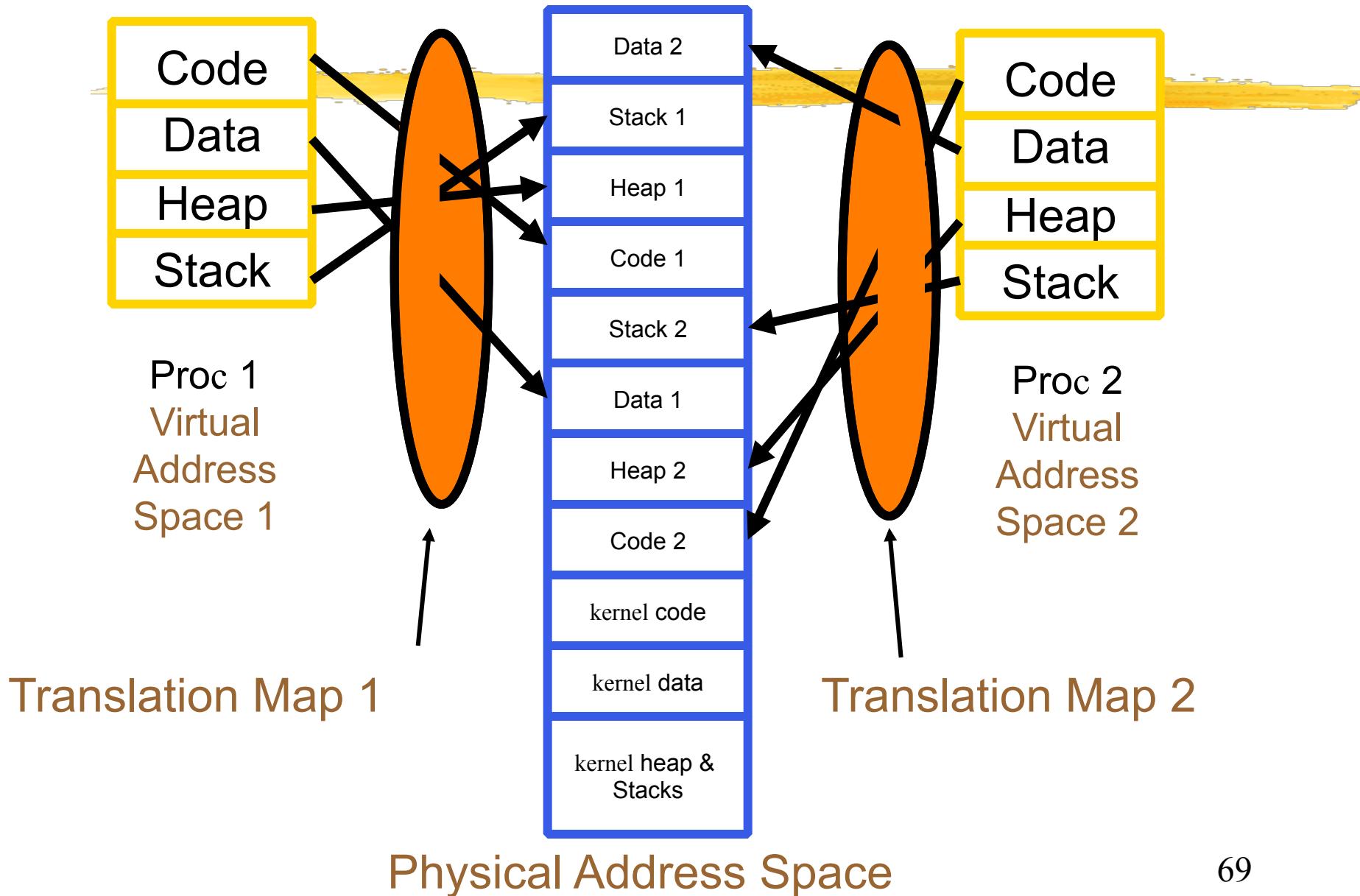
Virtual Address

Virtual Addresses
(Process Layout)

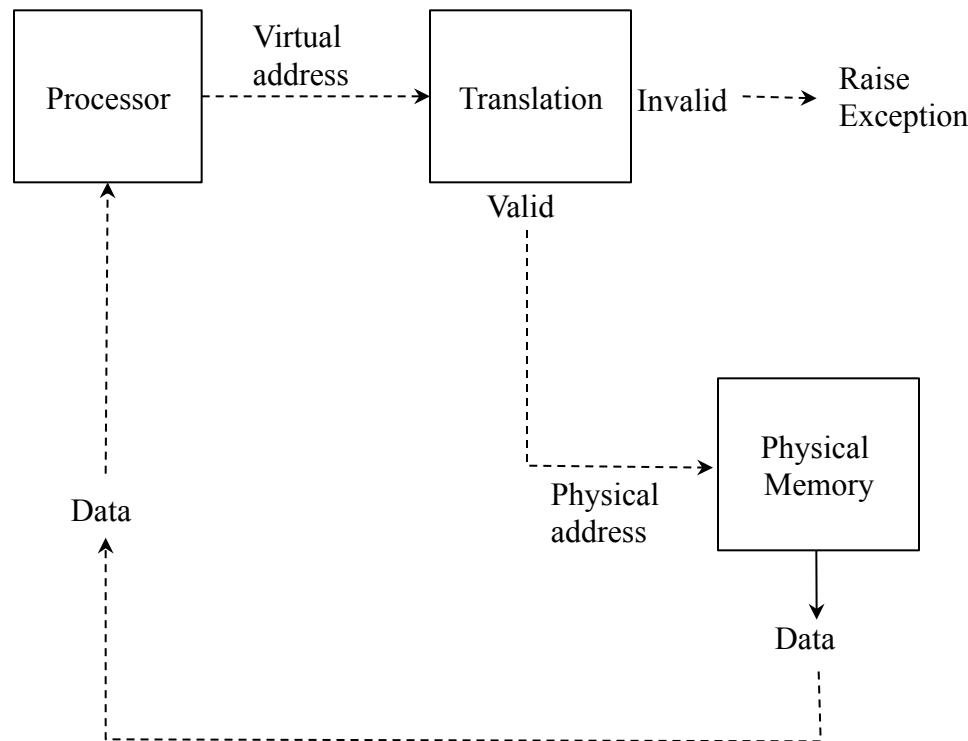
Physical
Memory



Providing the Illusion of Separate Address Spaces



Address translation and memory protection



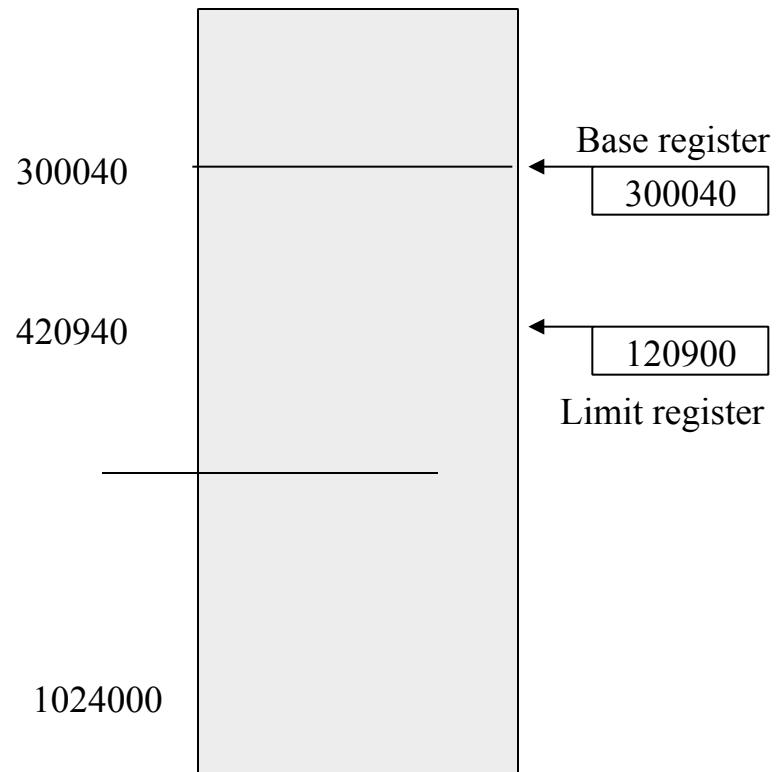
Memory Protection



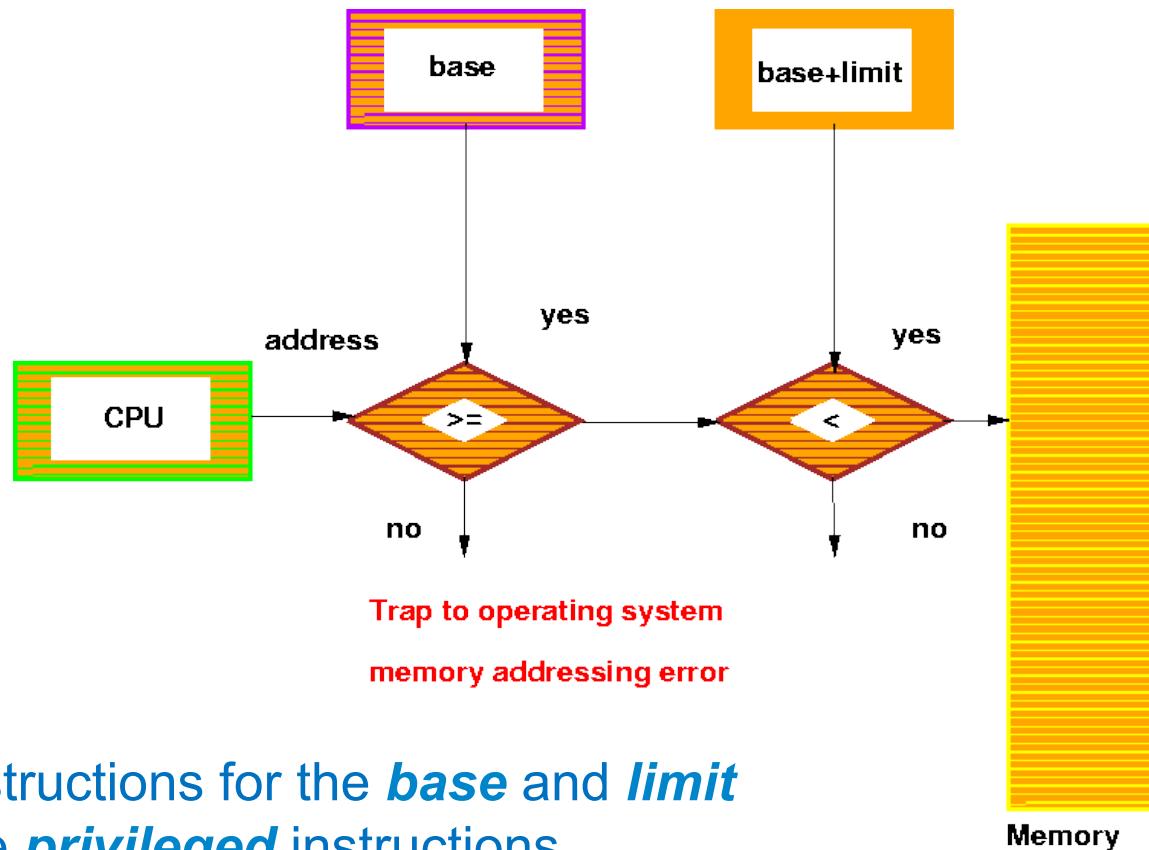
- When a process is running, only memory in that process address space must be accessible.
- When executing in kernel mode, the kernel has unrestricted access to all memory.

Memory Protection: **base and limit**

- To provide memory protection, add two registers that determine the range of legal addresses a program may address.
 - **Base Register** - holds smallest legal physical memory address.
 - **Limit register** - contains the size of the range.
- Memory outside the defined range is protected.
- Sometimes called **Base and Bounds** method

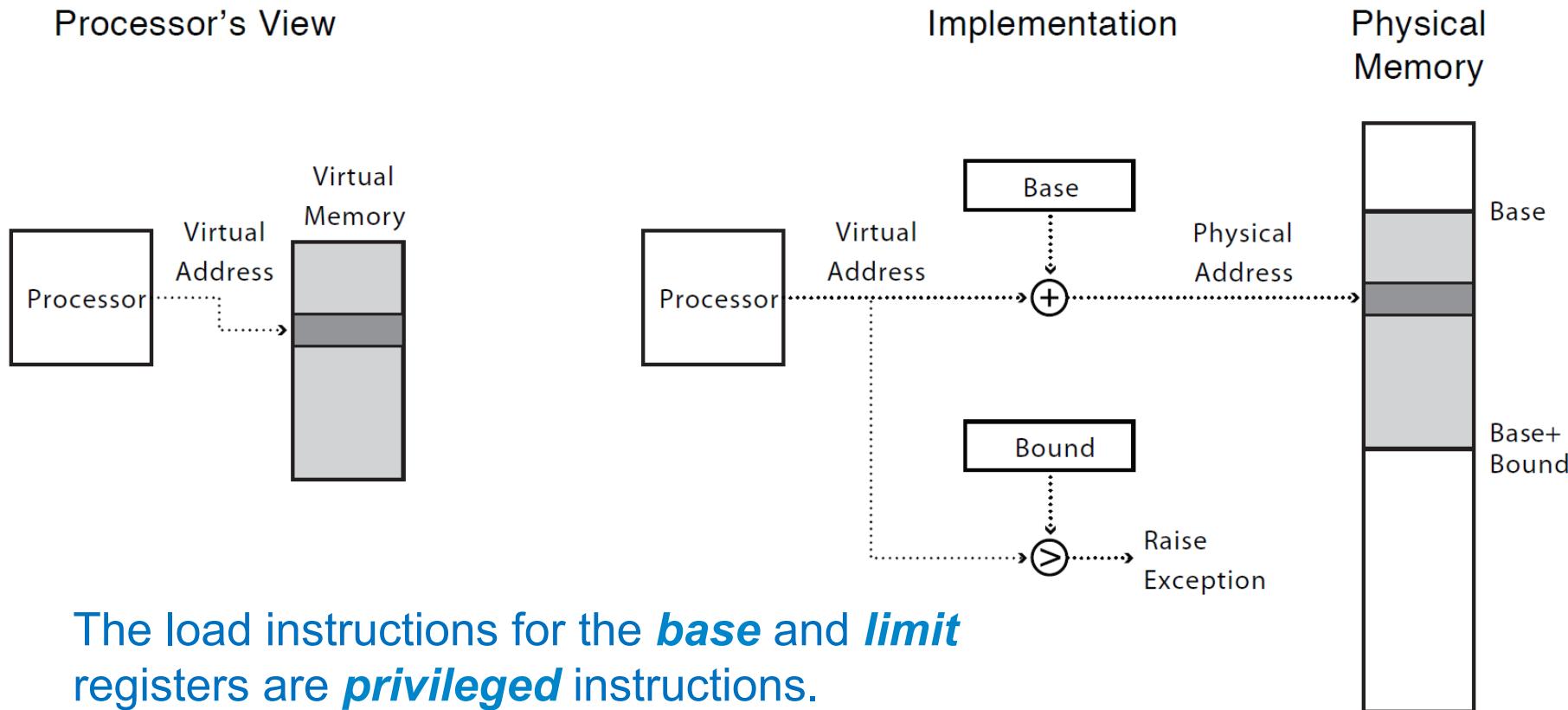


Hardware Address Protection



The load instructions for the **base** and **limit** registers are **privileged** instructions.

Virtual Address translation using the Base and Bounds method



The load instructions for the **base** and **limit** registers are **privileged** instructions.

I/O Protection



- All I/O instructions are privileged instructions.

Question



- Given the I/O instructions are privileged, how do users perform I/O?

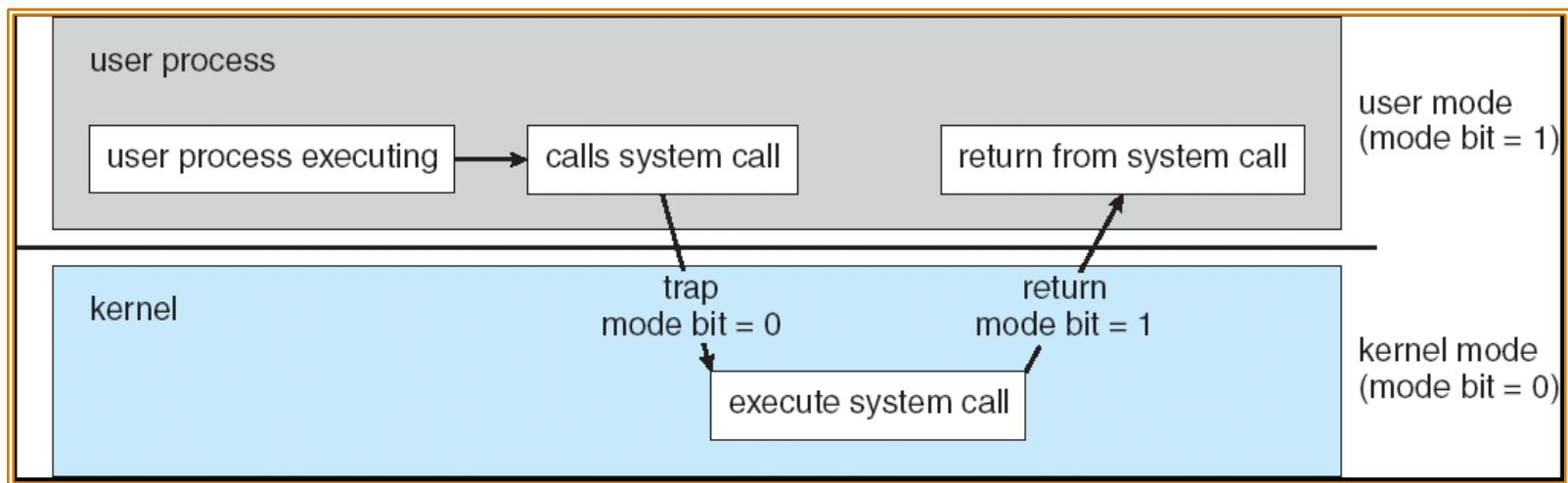
Question



- Given the I/O instructions are privileged, how do users perform I/O?
- Via system calls - the method used by a process to request action by the operating system.

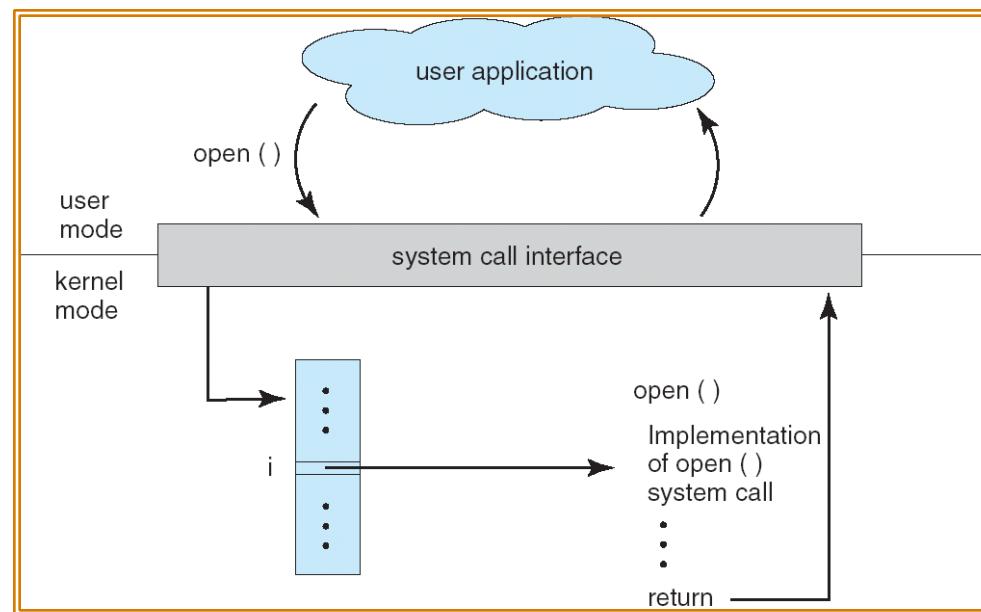
System Calls

- User code can issue a syscall, which causes a trap
- Kernel handles the syscall



System Calls

- Interface between applications and the OS.
- Application uses an assembly instruction to trap into the kernel
- Some higher level languages provide wrappers for system calls (e.g., C)
- System calls pass parameters between an application and OS via registers or memory
- Linux has about 300 system calls
 - `read()`, `write()`, `open()`, `close()`, `fork()`, `exec()`, `ioctl()`,.....



System **services** or system programs



- Convenient environment for program development and execution.
 - Command Interpreter (i.e., shell) - parses/executes other system programs
 - Window management
 - System libraries, e.g., libc

Command Interpreter System



- Commands that are given to the operating system via command statements that execute
 - Process creation and deletion, I/O handling, secondary storage management, main memory Management, file system access, protection, networking, etc.
- Obtains the next command and executes it.
- Programs that read and interpret control statements also called -
 - Command-line interpreter, shell (in UNIX)

Storage Structure



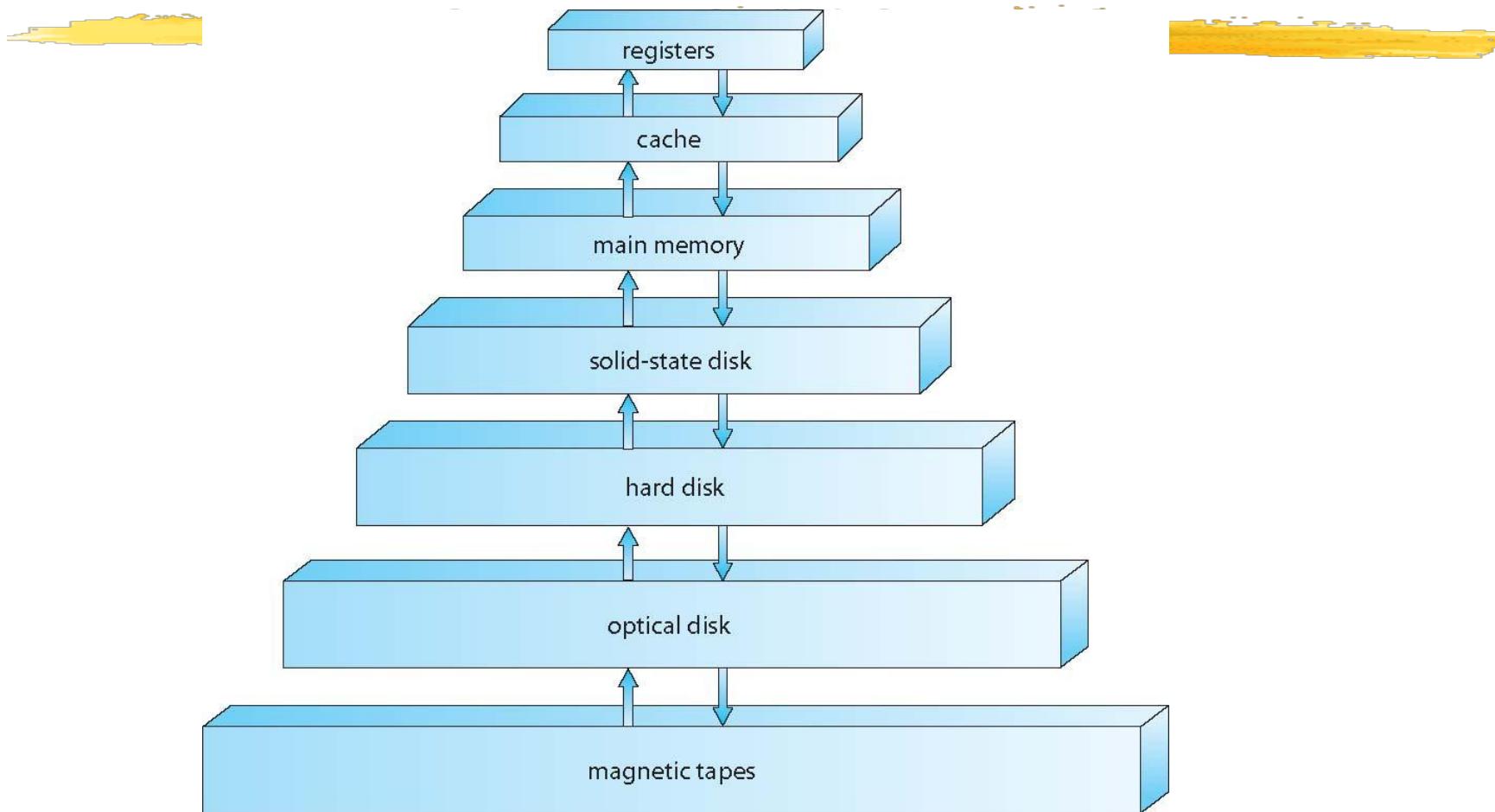
- Main memory - only large storage media that the CPU can access directly.
- Secondary storage - has large nonvolatile storage capacity.
 - Magnetic disks - rigid metal or glass platters covered with magnetic recording material.
 - Disk surface is logically divided into tracks, subdivided into sectors.
 - Disk controller determines logical interaction between device and computer.

Storage Hierarchy



- Storage systems are organized in a hierarchy based on
 - Speed
 - Cost
 - Volatility
- Caching - process of copying information into faster storage system; main memory can be viewed as fast cache for secondary storage.

Storage Device Hierarchy



Operating Systems: How are they organized?

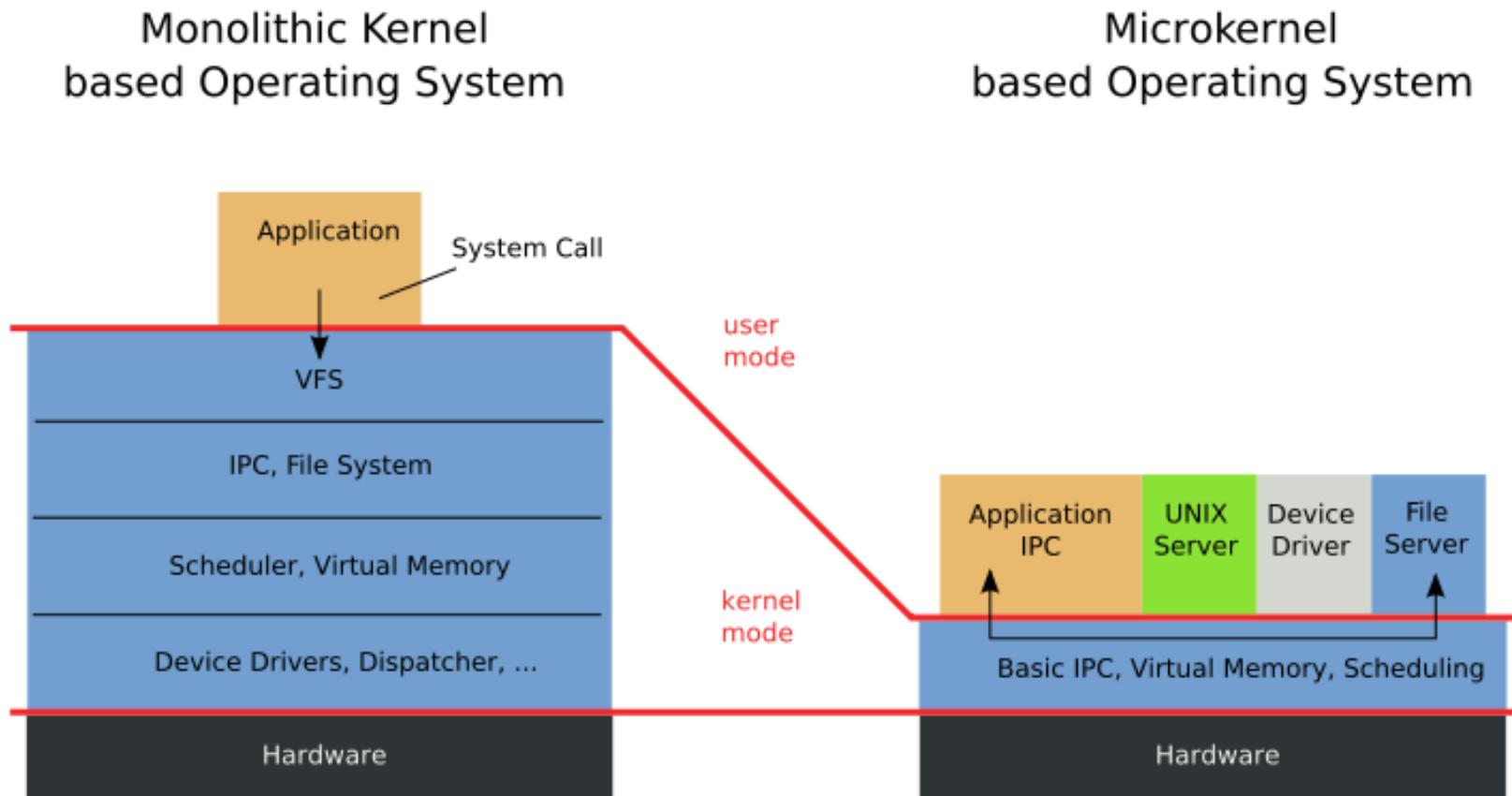


Monolithic vs. Microkernel OS



- Monolithic OSes have large kernels with a lot of components
 - Linux, Windows, Mac
- Microkernels moves as much from the kernel into “user” space
 - Small core OS components running at kernel level
 - OS Services built from many independent user-level processes
- Communication between modules with message passing
- Benefits:
 - Easier to extend a microkernel
 - Easier to port OS to new architectures
 - More reliable and more secure (less code is running in kernel mode)
- Detriments:
 - Performance overhead severe for naïve implementation

A microkernel OS



OS Task: Process Management



- Process - fundamental concept in OS
 - Process is an instance of a program in execution.
 - Process needs resources - CPU time, memory, files/data and I/O devices.
- OS is responsible for the following process management activities.
 - Process creation and deletion
 - Process suspension and resumption
 - Process synchronization and interprocess communication
 - Process interactions - deadlock detection, avoidance and correction

OS Task: Memory Management



- Main Memory is an array of addressable words or bytes that is quickly accessible.
- Main Memory is volatile.
- OS is responsible for:
 - Allocate and deallocate memory to processes.
 - Managing multiple processes within memory - keep track of which parts of memory are used by which processes. Manage the sharing of memory between processes.
 - Determining which processes to load when memory becomes available.

OS Task: Secondary Storage and I/O Management



- Since primary storage (i.e., main memory) is expensive and volatile, secondary storage is required for backup.
- Disk is the primary form of secondary storage.
 - OS performs storage allocation, free-space management, etc.
- I/O system in the OS consists of
 - Device driver interface that abstracts device details
 - Drivers for specific hardware devices

OS Task: File System Management



- File is a collection of related information - represents programs and data.
- OS is responsible for
 - File creation and deletion
 - Directory creation and deletion
 - Supporting primitives for file/directory manipulation.
 - Mapping files to disks (secondary storage).

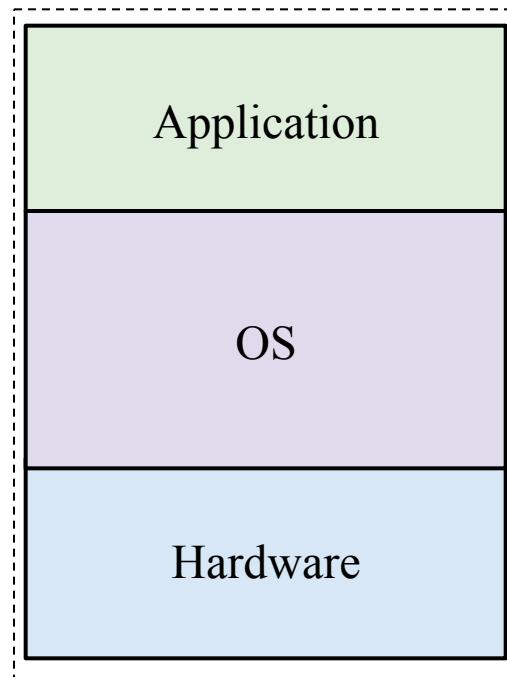
OS Task: Protection and Security



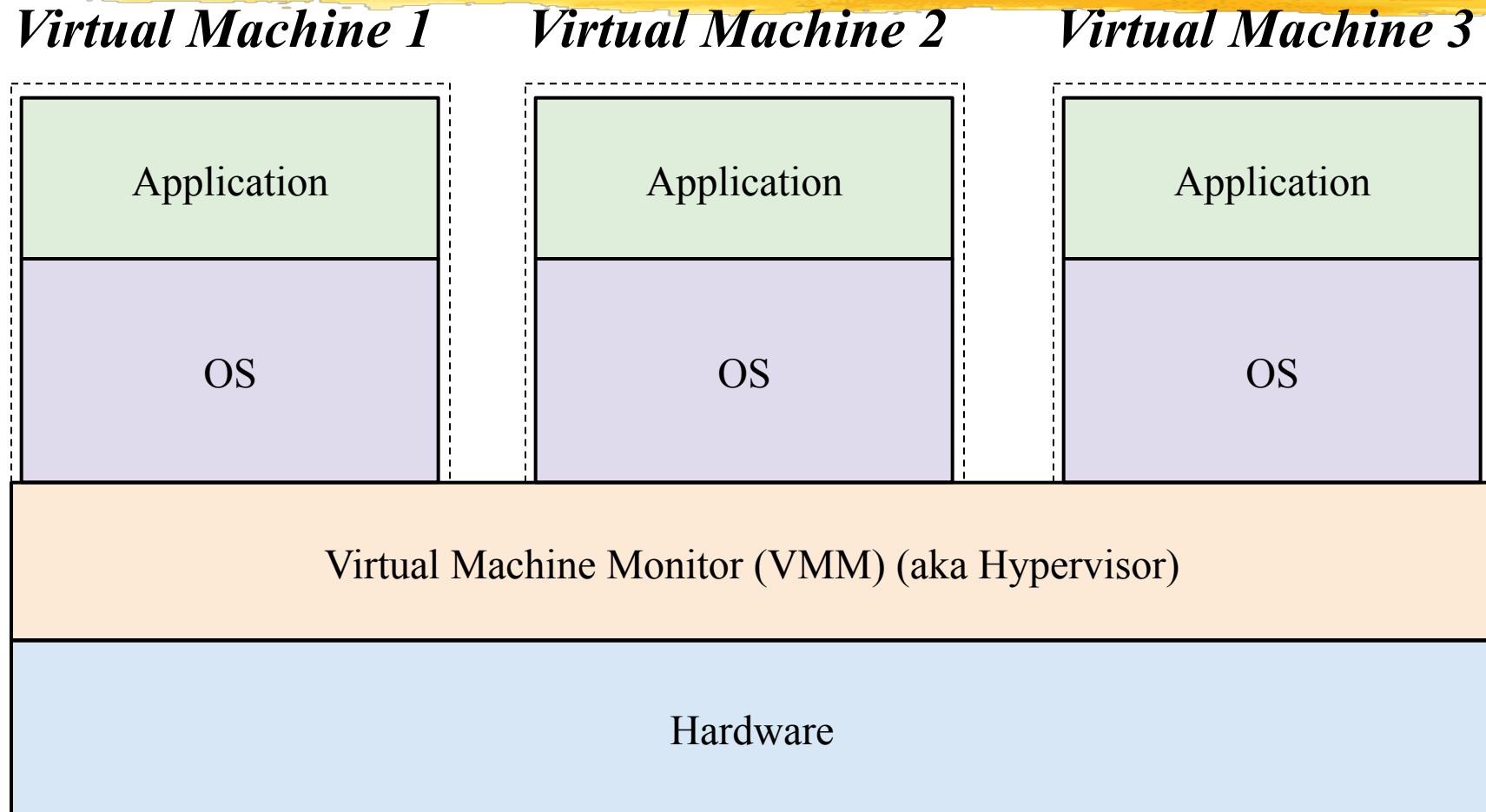
- Protection mechanisms control access of processes to user and system resources.
- Protection mechanisms must:
 - Distinguish between authorized and unauthorized use.
 - Specify access controls to be imposed on use.
 - Provide mechanisms for enforcement of access control.

Virtual Machines

Physical Machine



Virtual Machines



Virtual Machines



- Use cases
 - Resource configuration
 - Running multiple OSes, either the same or different OSes
 - Run existing OS binaries on different architecture

Summary of Lecture set 1



- What is an operating system?
- Operating systems history
- Computer system and operating system structure

ICS 143 - Principles of Operating Systems

Lectures 3 and 4 - Processes and Threads

Prof. Nalini Venkatasubramanian

nalini@ics.uci.edu

Outline

- Process Concept
 - Process Scheduling
 - Operations on Processes
 - Cooperating Processes
- Threads
- Interprocess Communication

Process Concept

- An operating system executes a variety of programs
 - batch systems - jobs
 - time-shared systems - user programs or tasks
 - Job, task and program used interchangeably
- Process - a program in execution
 - process execution proceeds in a sequential fashion
- A process contains
 - program counter, stack and data section

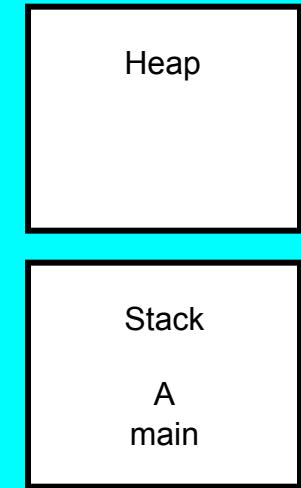
Process =? Program

```
main ()  
{  
    ...;  
}  
A () {  
    ...  
}
```

Program

```
main ()  
{  
    ...;  
}  
A () {  
    ...  
}
```

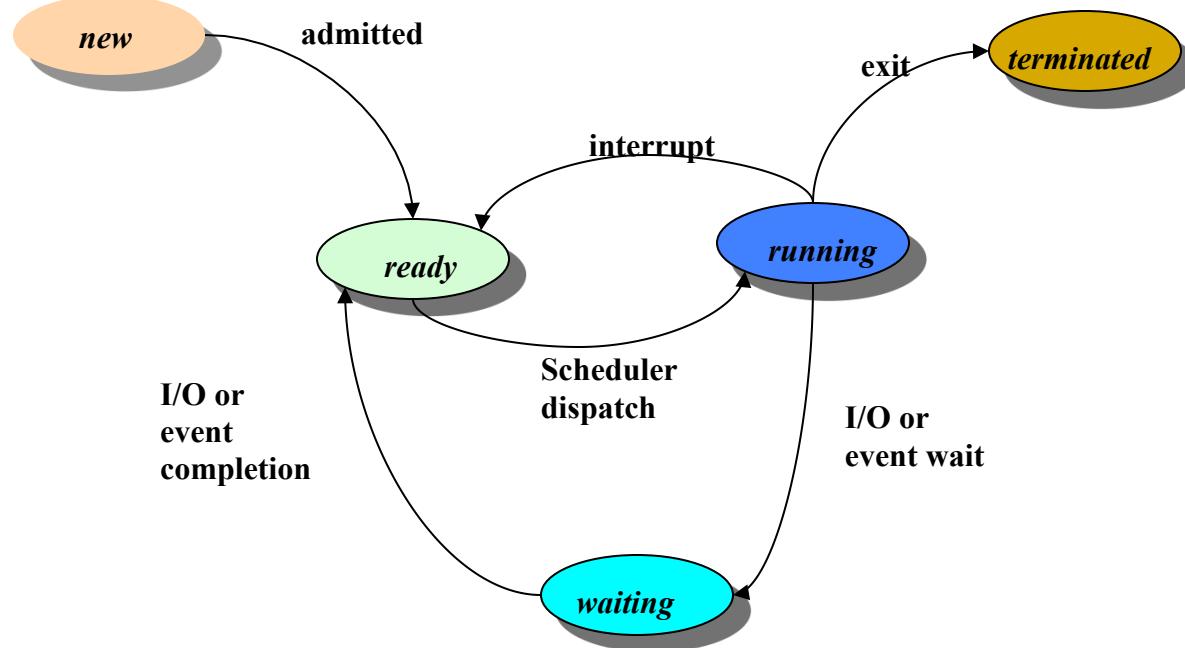
Process



- More to a process than just a program:
 - Program is just part of the process state
 - I run Vim or Notepad on lectures.txt, you run it on homework.java – Same program, different processes
- Less to a process than a program:
 - A program can invoke more than one process
 - A web browser launches multiple processes, e.g., one per tab

Process State

- A process changes state as it executes.

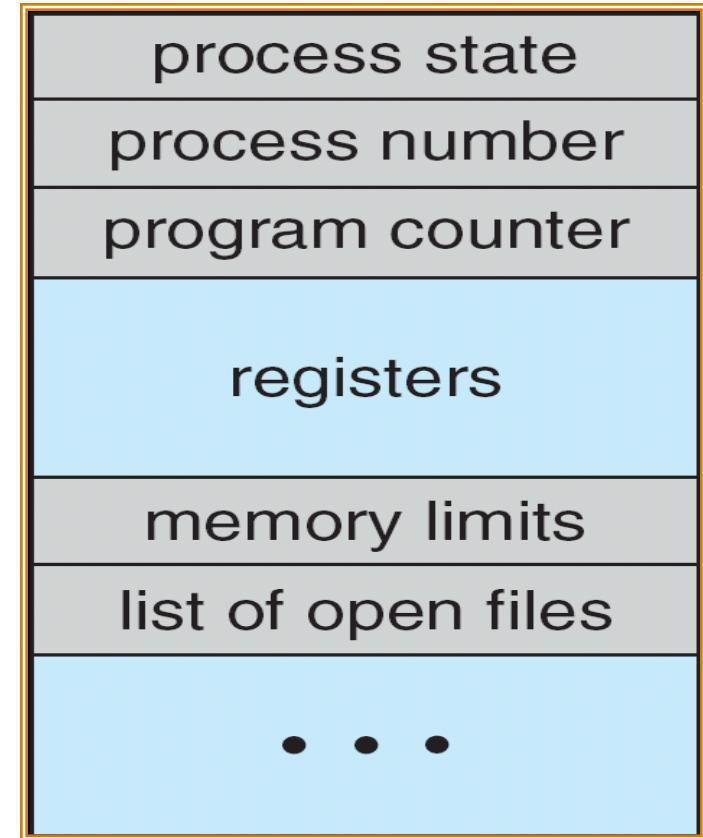


Process States

- New - The process is being created.
- Running - Instructions are being executed.
- Waiting - Waiting for some event to occur.
- Ready - Waiting to be assigned to a processor.
- Terminated - Process has finished execution.

Process Control Block

- Contains information associated with each process
 - Process State - e.g. new, ready, running etc.
 - Process Number – Process ID
 - Program Counter - address of next instruction to be executed
 - CPU registers - general purpose registers, stack pointer etc.
 - CPU scheduling information - process priority, pointer
 - Memory Management information - base/limit information
 - Accounting information - time limits, process number
 - I/O Status information - list of I/O devices allocated

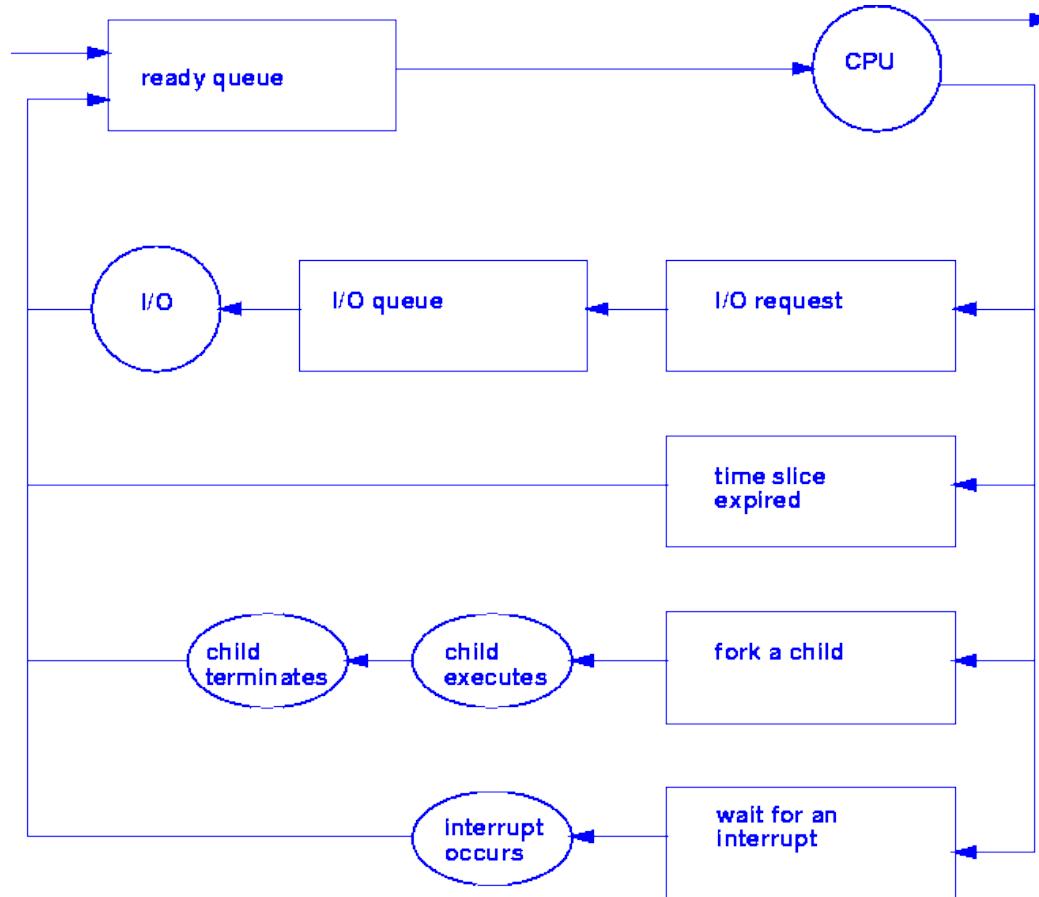


Process
Control
Block

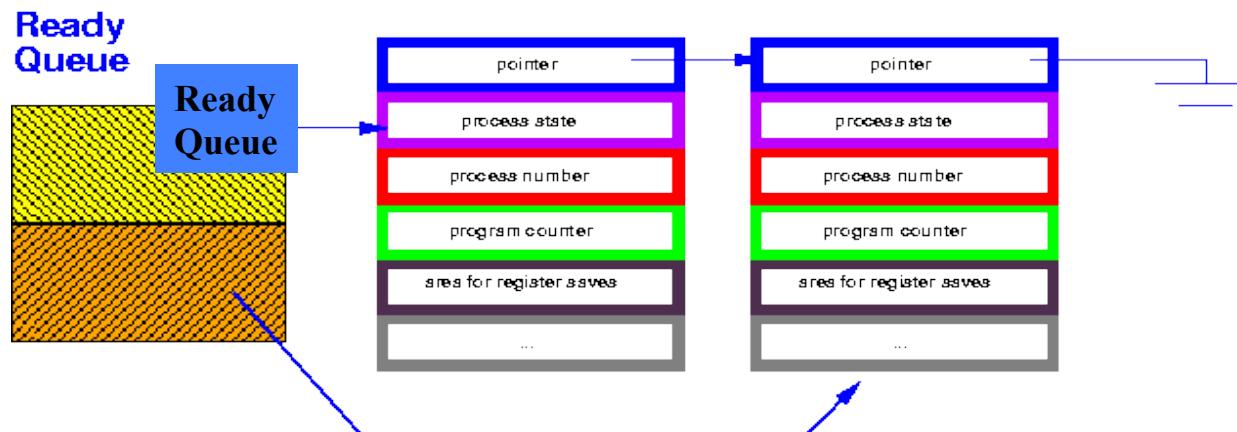
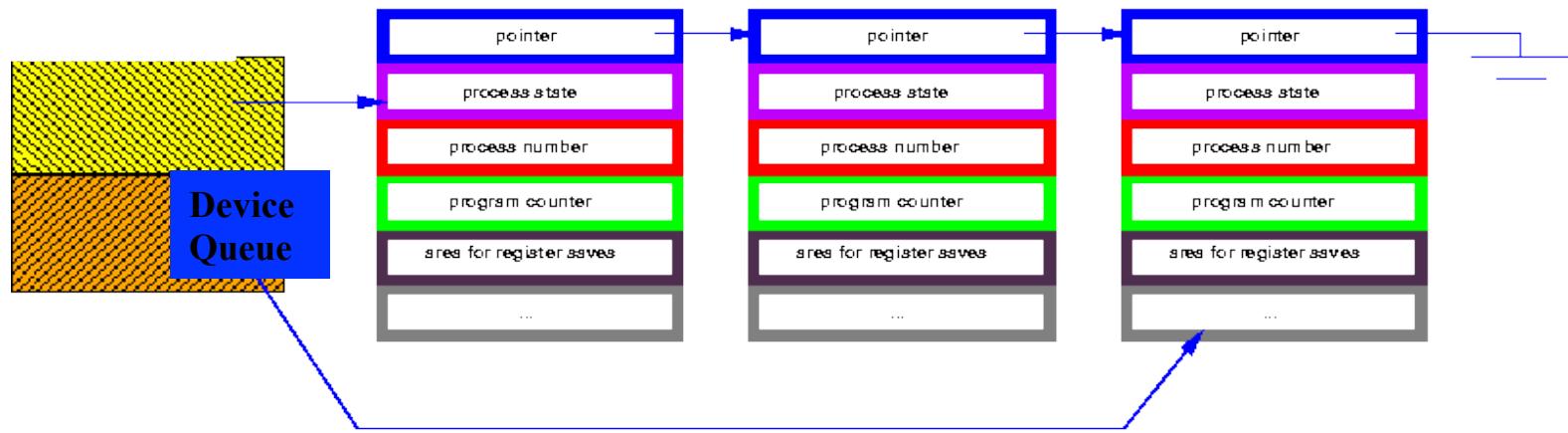
Representation of Process Scheduling

Process (PCB) moves from queue to queue

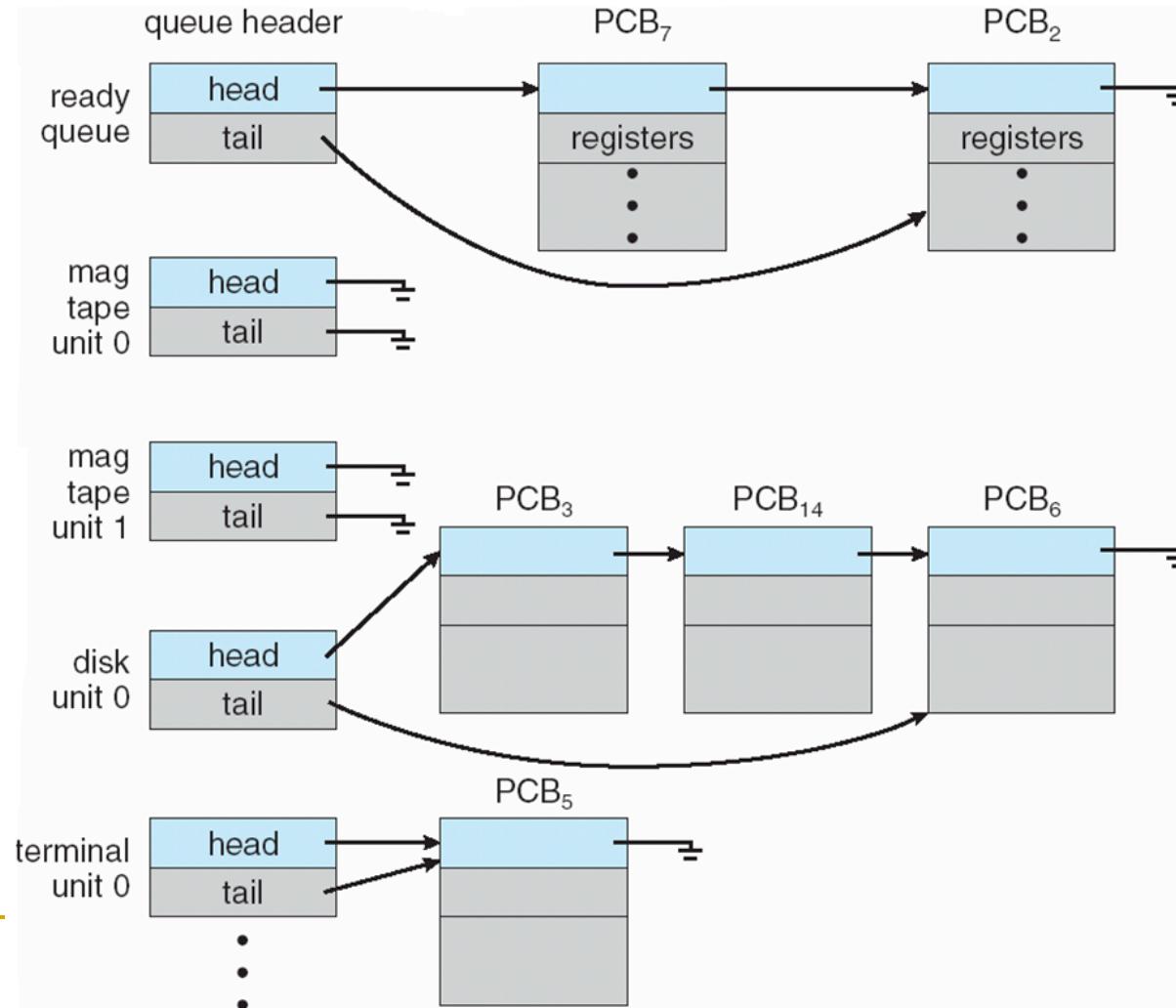
When does it move? Where? A scheduling decision



Process Queues



Ready Queue And Various I/O Device Queues

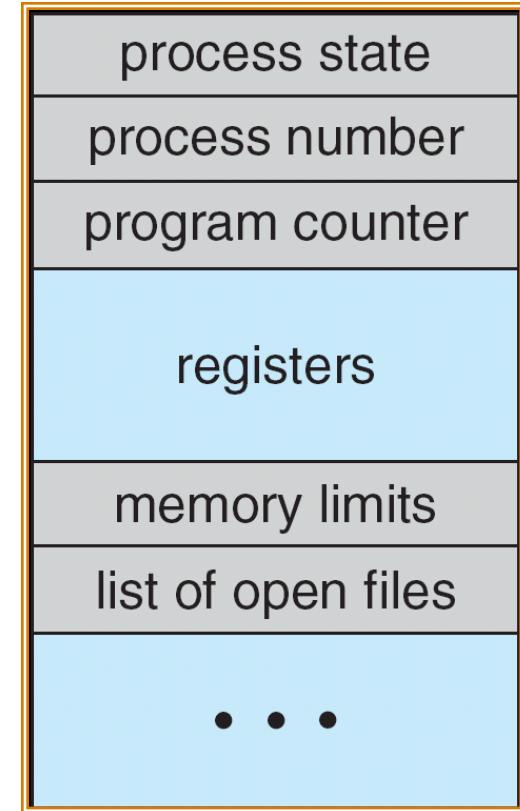


Process Scheduling Queues

- Job Queue - set of all processes in the system
- Ready Queue - set of all processes residing in main memory, ready and waiting to execute.
- Device Queues - set of processes waiting for an I/O device.
- Process migration between the various queues.
- Queue Structures - typically linked list, circular list etc.

Enabling Concurrency and Protection: Multiplex processes

- Only one process (PCB) active at a time
 - Current state of process held in PCB:
 - “snapshot” of the execution and protection environment
 - Process needs CPU, resources
- Give out CPU time to different processes (Scheduling):
 - Only one process “running” at a time
 - Give more time to important processes
- Give pieces of resources to different processes (Protection):
 - Controlled access to non-CPU resources
 - E.g. Memory Mapping: Give each process their own address space

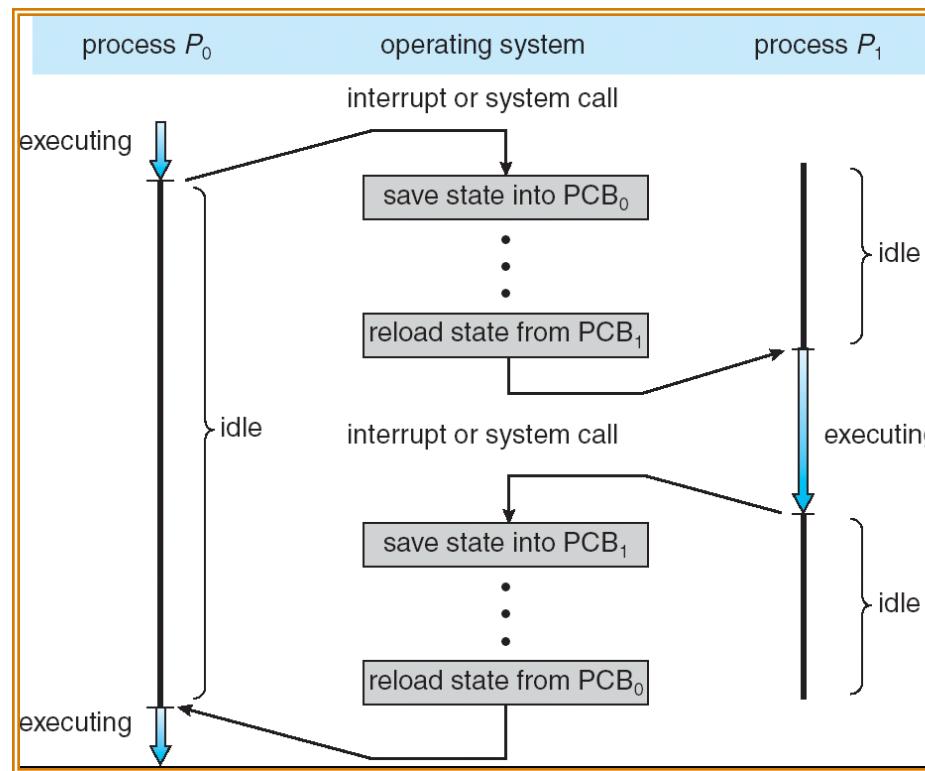


Process
Control
Block

Enabling Concurrency: Context Switch

- Task that switches CPU from one process to another process
 - the CPU must save the PCB state of the old process and load the saved PCB state of the new process.
- Context-switch time is overhead
 - System does no useful work while switching
 - Overhead sets minimum practical switching time; can become a bottleneck
- Time for context switch is dependent on hardware support (1- 1000 microseconds).

CPU Switch From Process to Process

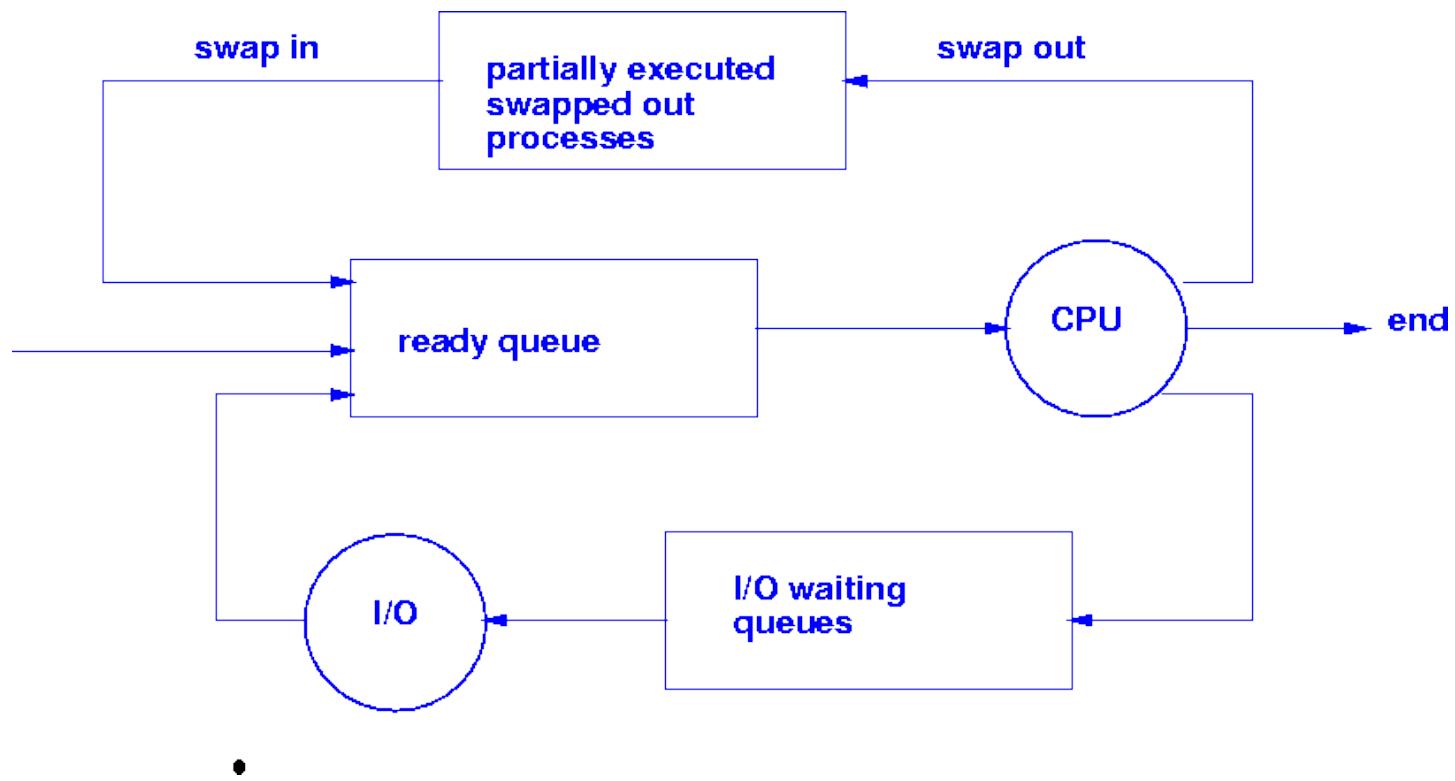


- Code executed in kernel above is overhead
 - Overhead sets minimum practical switching time

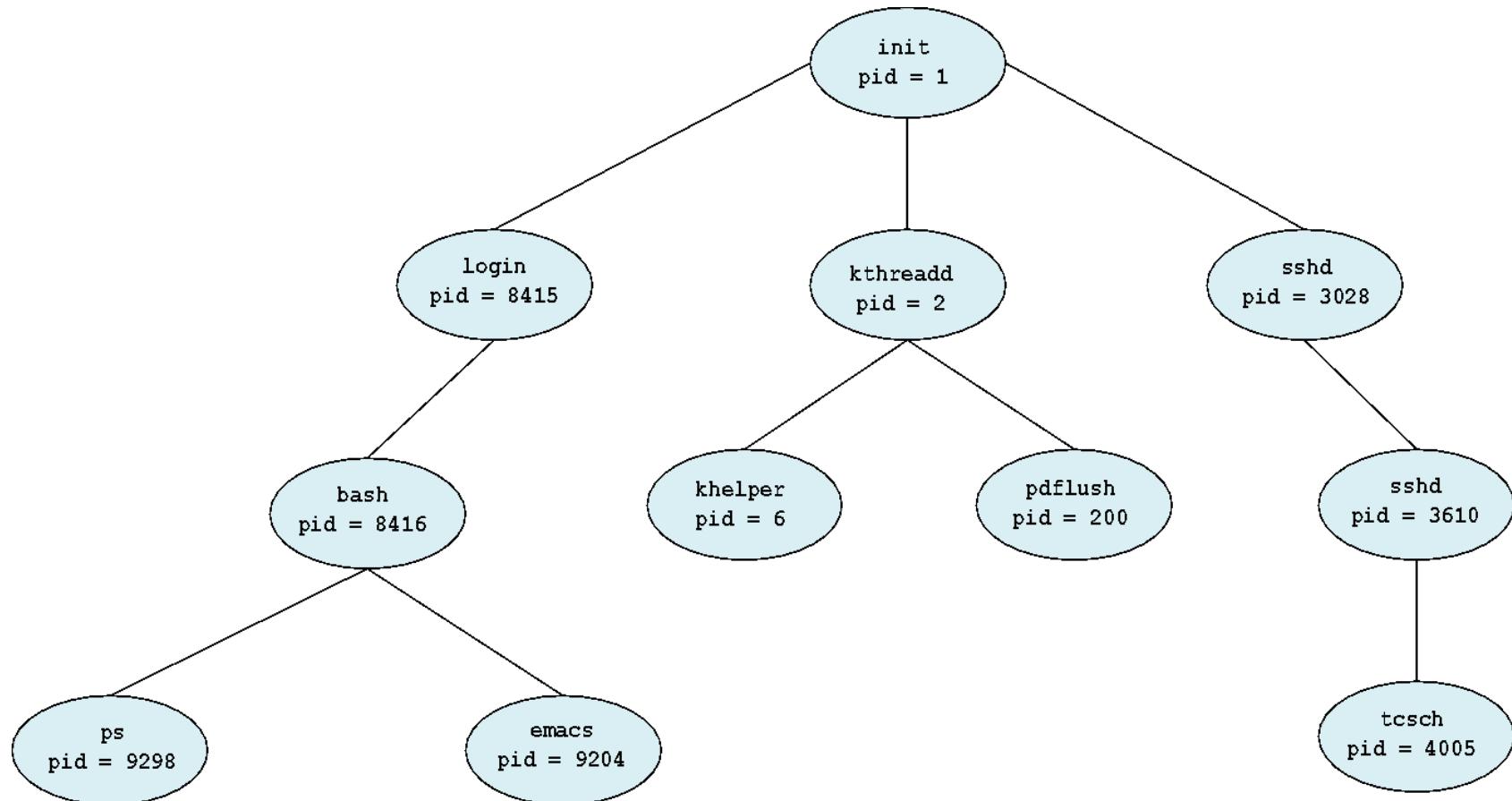
Schedulers

- **Long-term scheduler (or job scheduler) -**
 - selects which processes should be brought into the ready queue.
 - invoked very infrequently (seconds, minutes); may be slow.
 - controls the degree of multiprogramming
- **Short term scheduler (or CPU scheduler) -**
 - selects which process should execute next and allocates CPU.
 - invoked very frequently (milliseconds) - must be very fast
 - Sometimes the only scheduler in the system
- **Medium Term Scheduler**
 - swaps out process temporarily
 - balances load for better throughput

Medium Term (Time-sharing) Scheduler



A tree of processes in Linux



Process Profiles

■ I/O bound process -

- ❑ spends more time in I/O, short CPU bursts, CPU underutilized.

■ CPU bound process -

- ❑ spends more time doing computations; few very long CPU bursts, I/O underutilized.

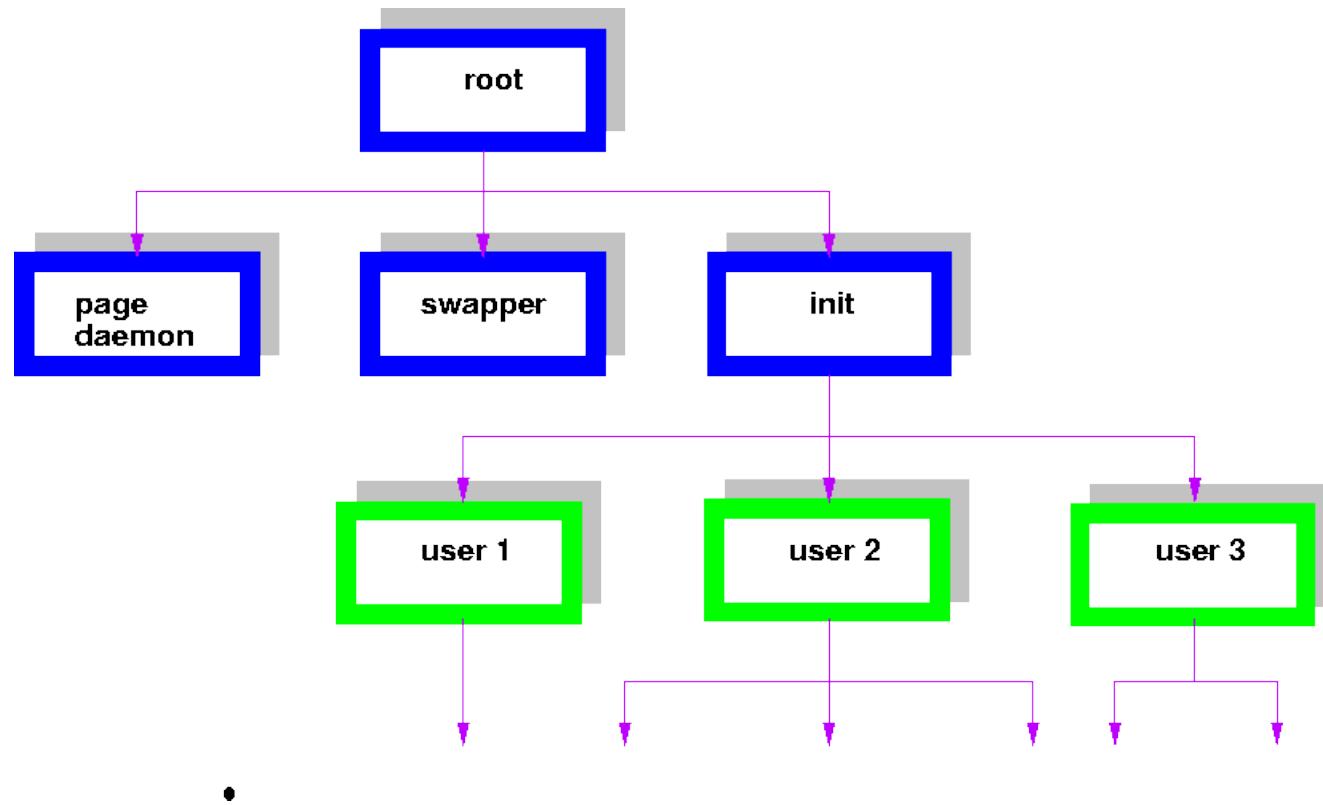
■ The right job mix:

- ❑ Long term scheduler - admits jobs to keep load balanced between I/O and CPU bound processes
- ❑ Medium term scheduler – ensures the right mix (by sometimes swapping out jobs and resuming them later)

Process Creation

- Processes are created and deleted dynamically
- Process which creates another process is called a *parent* process; the created process is called a *child* process.
- Result is a tree of processes
 - e.g. UNIX - processes have dependencies and form a hierarchy.
- Resources required when creating process
 - CPU time, files, memory, I/O devices etc.

UNIX Process Hierarchy



What does it take to create a process?

- Must construct new PCB
 - Inexpensive
- Must set up new page tables for address space
 - More expensive
- Copy data from parent process? (Unix `fork()`)
 - Semantics of Unix `fork()` are that the child process gets a complete copy of the parent memory and I/O state
 - Originally *very* expensive
 - Much less expensive with “copy on write”
- Copy I/O state (file handles, etc)
 - Medium expense

Process Creation

■ Resource sharing

- ❑ Parent and children share all resources.
- ❑ Children share subset of parent's resources - prevents many processes from overloading the system.
- ❑ Parent and children share no resources.

■ Execution

- ❑ Parent and child execute concurrently.
- ❑ Parent waits until child has terminated.

■ Address Space

- ❑ Child process is duplicate of parent process.
- ❑ Child process has a program loaded into it.

UNIX Process Creation

- Fork system call creates new processes
- execve system call is used after a fork to replace the processes memory space with a new program.

Process Termination

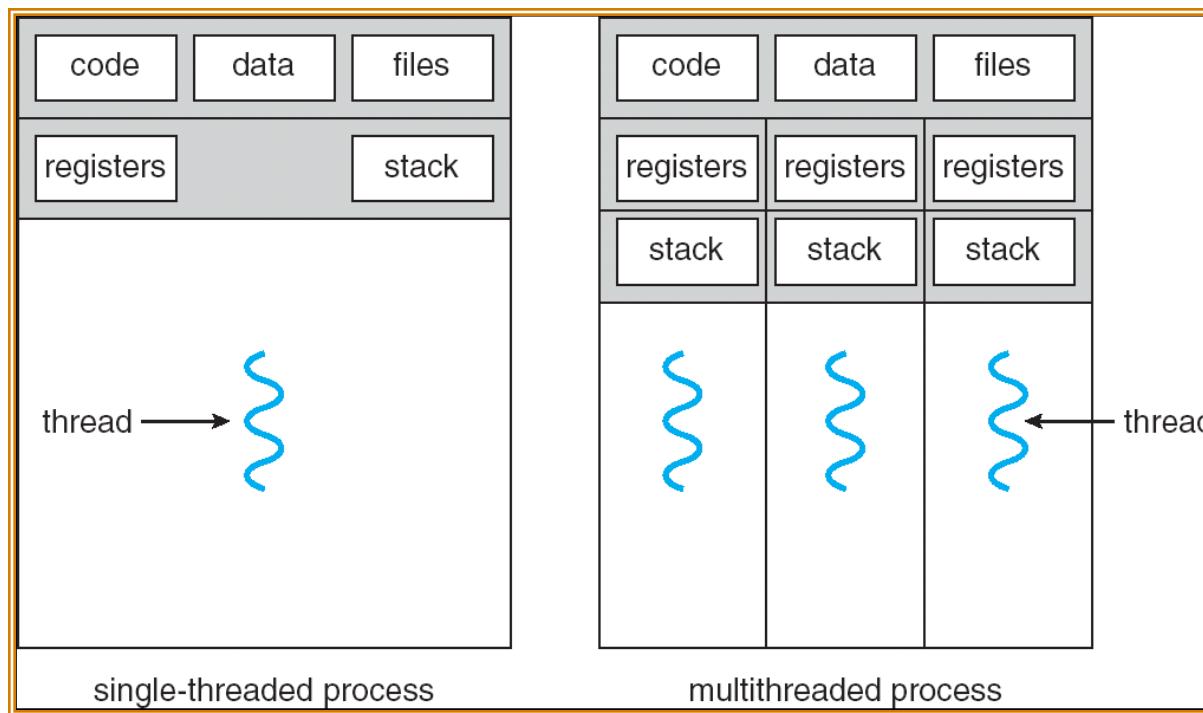
- Process executes last statement and asks the operating system to delete it (*exit*).
 - Output data from child to parent (via wait).
 - Process' resources are deallocated by operating system.
- Parent may terminate execution of child processes.
 - Child has exceeded allocated resources.
 - Task assigned to child is no longer required.
 - Parent is exiting
 - OS does not allow child to continue if parent terminates
 - Cascading termination

Threads

- Processes do not share resources well
 - high context switching overhead
- Idea: Separate concurrency from protection
- Multithreading: *a single program made up of a number of different concurrent activities*
- A thread (or lightweight process)
 - basic unit of CPU utilization; it consists of:
 - program counter, register set and stack space
 - A thread shares the following with peer threads:
 - code section, data section and OS resources (open files, signals)
 - No protection between threads
 - Collectively called a task.
- Heavyweight process is a task with one thread.



Single and Multithreaded Processes



- Threads encapsulate concurrency: “Active” component
- Address spaces encapsulate protection: “Passive” part
 - Keeps buggy program from trashing the system

Benefits

- Responsiveness
- Resource Sharing
- Economy
- Utilization of MP Architectures

Threads(Cont.)

- In a multiple threaded task, while one server thread is blocked and waiting, a second thread in the same task can run.
 - Cooperation of multiple threads in the same job confers higher throughput and improved performance.
 - Applications that require sharing a common buffer (i.e. producer-consumer) benefit from thread utilization.
- Threads provide a mechanism that allows sequential processes to make blocking system calls while also achieving parallelism.

Thread State

- State shared by all threads in process/addr space
 - Contents of memory (global variables, heap)
 - I/O state (file system, network connections, etc)
- State “private” to each thread
 - Kept in TCB ≡ Thread Control Block
 - CPU registers (including, program counter)
 - Execution stack
 - Parameters, Temporary variables
 - return PCs are kept while called procedures are executing

Threads (cont.)

- Thread context switch still requires a register set switch, but no memory management related work!!
- Thread states -
 - *ready, blocked, running, terminated*
- Threads share CPU and only one thread can run at a time.
- No protection among threads.

Examples: Multithreaded programs

■ Embedded systems

- ❑ Elevators, Planes, Medical systems, Wristwatches
- ❑ Single Program, concurrent operations

■ Most modern OS kernels

- ❑ Internally concurrent because have to deal with concurrent requests by multiple users
- ❑ But no protection needed within kernel

■ Database Servers

- ❑ Access to shared data by many concurrent users
- ❑ Also background utility processing must be done

More Examples: Multithreaded programs

■ Network Servers

- Concurrent requests from network
- Again, single program, multiple concurrent operations
- File server, Web server, and airline reservation systems

■ Parallel Programming (More than one physical CPU)

- Split program into multiple threads for parallelism
- This is called Multiprocessing

| # threads Per AS: | # of addr spaces: | One | Many |
|----------------------|----------------------|---|---|
| One | | MS/DOS, early Macintosh | Traditional UNIX |
| Many | | Embedded systems (Geoworks, VxWorks, JavaOS,etc) JavaOS, Pilot(PC) | Mach, OS/2, Linux Windows 9x??? Win NT to XP, Solaris, HP-UX, OS X |

Real operating systems have either

- One or many address spaces
- One or many threads per address space

Types of Threads

- Kernel-supported threads
- User-level threads
- Hybrid approach implements both user-level and kernel-supported threads (Solaris 2).

Kernel Threads

■ Supported by the Kernel

- Native threads supported directly by the kernel
- Every thread can run or block independently
- One process may have several threads waiting on different things

■ Downside of kernel threads: a bit expensive

- Need to make a crossing into kernel mode to schedule

■ Examples

- Windows XP/2000, Solaris, Linux, Tru64 UNIX, Mac OS X, Mach, OS/2

User Threads

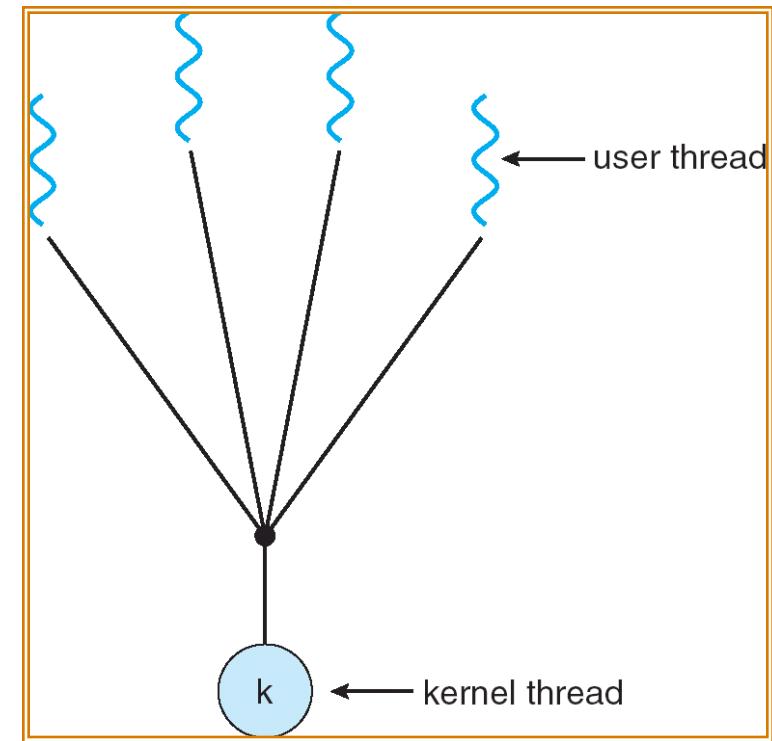
- Supported above the kernel, via a set of library calls at the user level.
 - Thread management done by user-level threads library
 - User program provides scheduler and thread package
 - May have several user threads per kernel thread
 - User threads may be scheduled non-preemptively relative to each other (only switch on yield())
- Advantages
 - Cheap, Fast
 - Threads do not need to call OS and cause interrupts to kernel
 - Disadv: If kernel is single threaded, system call from any thread can block the entire task.
- Example thread libraries:
 - POSIX Pthreads, Win32 threads, Java threads

Multithreading Models

- Many-to-One
- One-to-One
- Many-to-Many

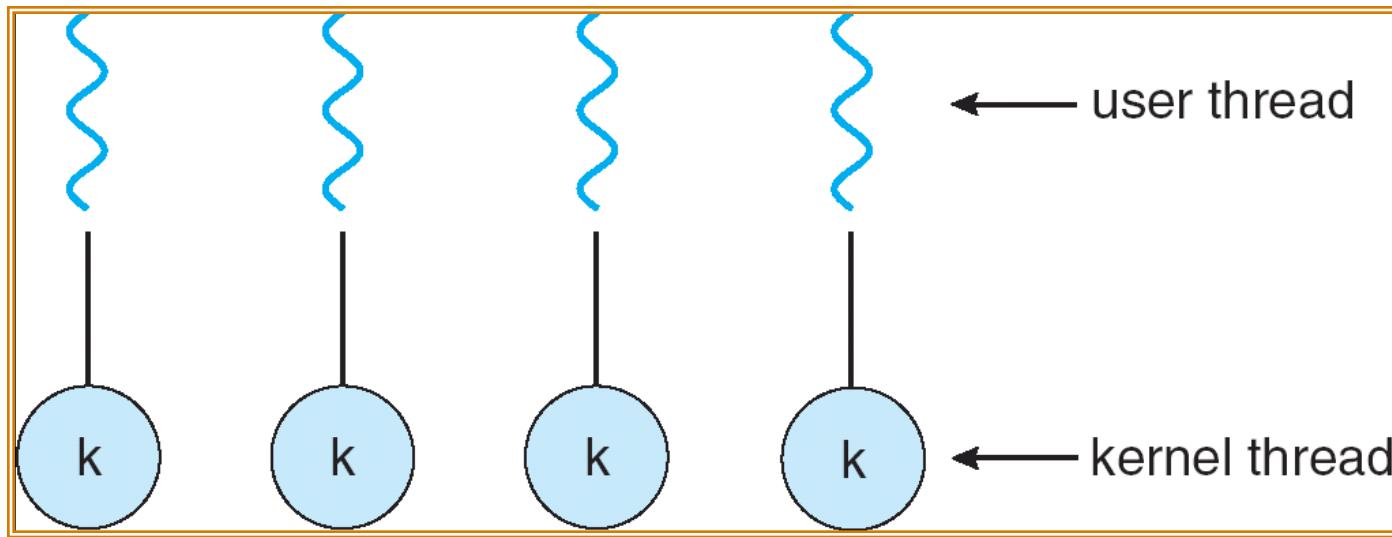
Many-to-One

- Many user-level threads mapped to single kernel thread
- Examples:
 - Solaris Green Threads
 - GNU Portable Threads



One-to-One

- Each user-level thread maps to kernel thread

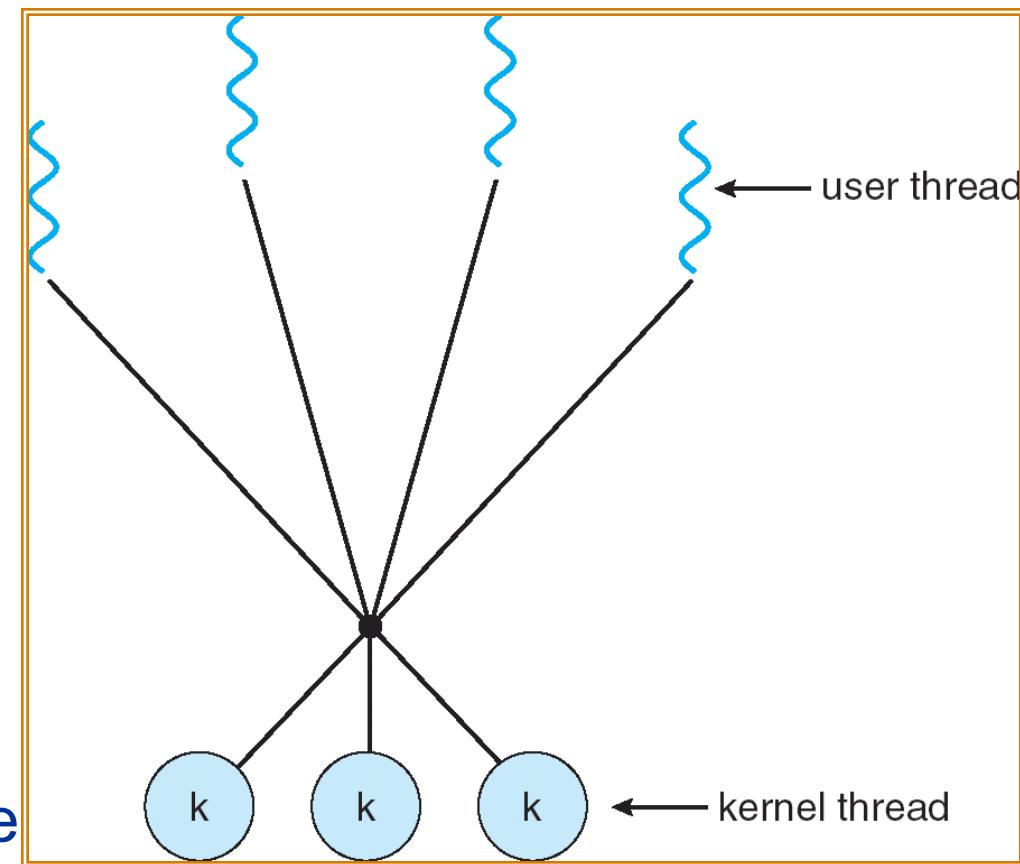


Examples

- Windows NT/XP/2000; Linux; Solaris 9 and later

Many-to-Many Model

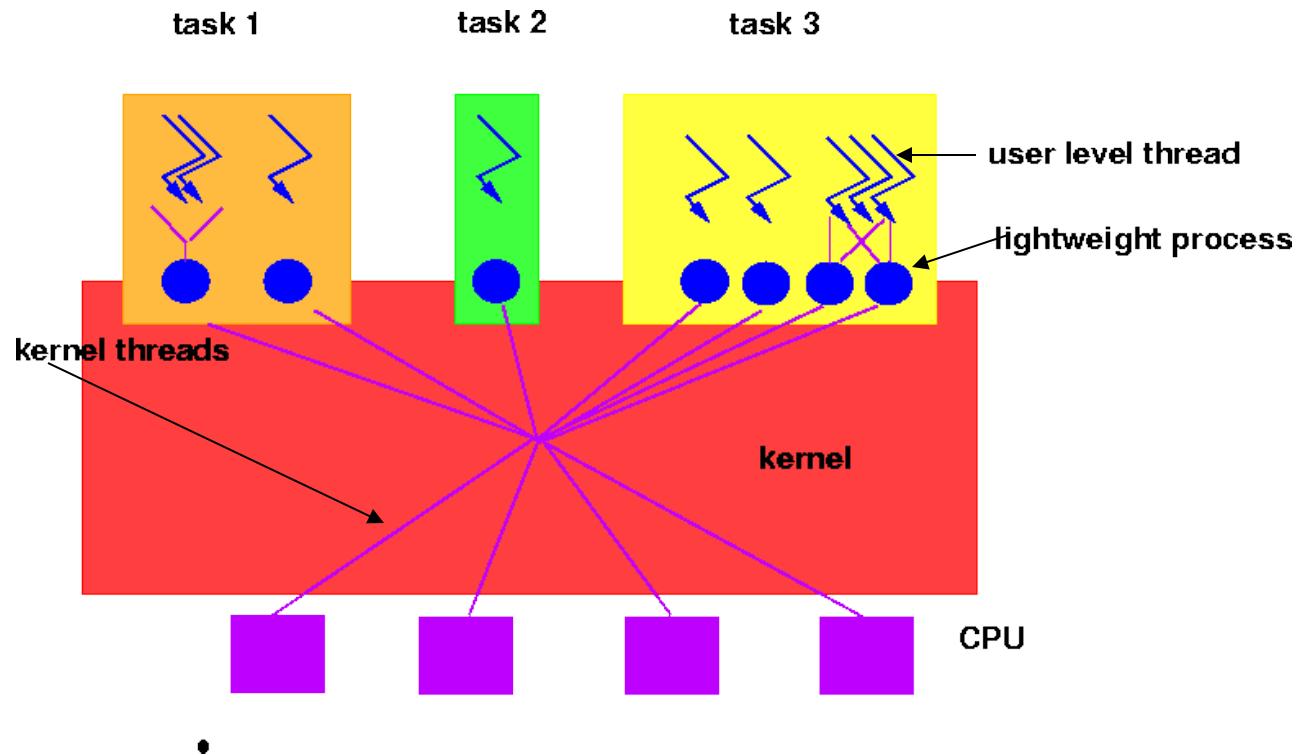
- Allows many user level threads to be mapped to many kernel threads
- Allows the operating system to create a sufficient number of kernel threads
- Solaris prior to version 9
- Windows NT/2000 with the *ThreadFiber* package



Thread Support in Solaris 2

- Solaris 2 is a version of UNIX with support for
 - kernel and user level threads, symmetric multiprocessing and real-time scheduling.
- Lightweight Processes (LWP)
 - intermediate between user and kernel level threads
 - each LWP is connected to exactly one kernel thread

Threads in Solaris 2



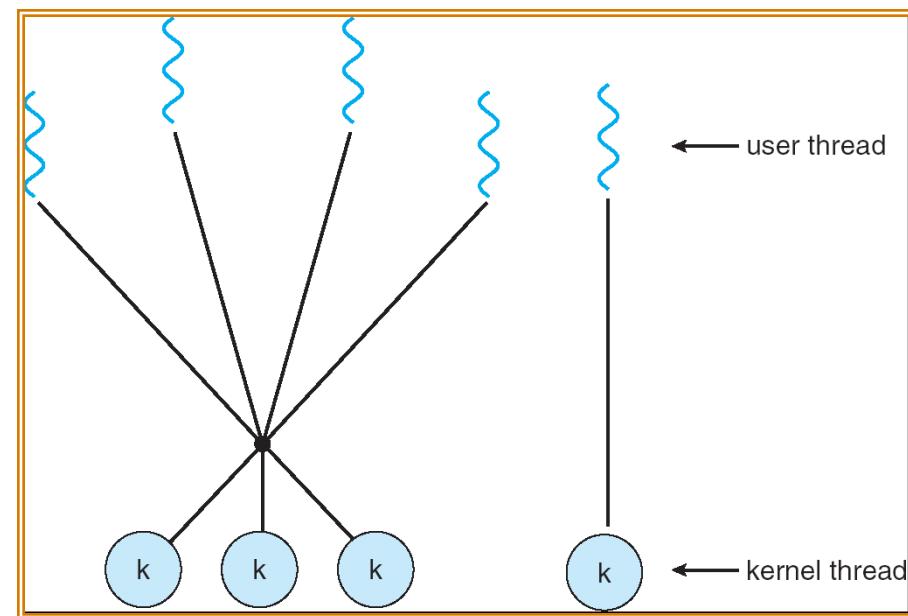
Threads in Solaris 2

■ Resource requirements of thread types

- Kernel Thread: small data structure and stack; thread switching does not require changing memory access information - relatively fast.
- Lightweight Process: PCB with register data, accounting and memory information - switching between LWP is relatively slow.
- User-level thread: only needs stack and program counter; no kernel involvement means fast switching. Kernel only sees the LWPs that support user-level threads.

Two-level Model

- Similar to M:M, except that it allows a user thread to be **bound** to kernel thread
- Examples
 - IRIX, HP-UX, Tru64 UNIX, Solaris 8 and earlier



Threading Issues

- Semantics of **fork()** and **exec()** system calls
- Thread cancellation
- Signal handling
- Thread pools
- Thread specific data

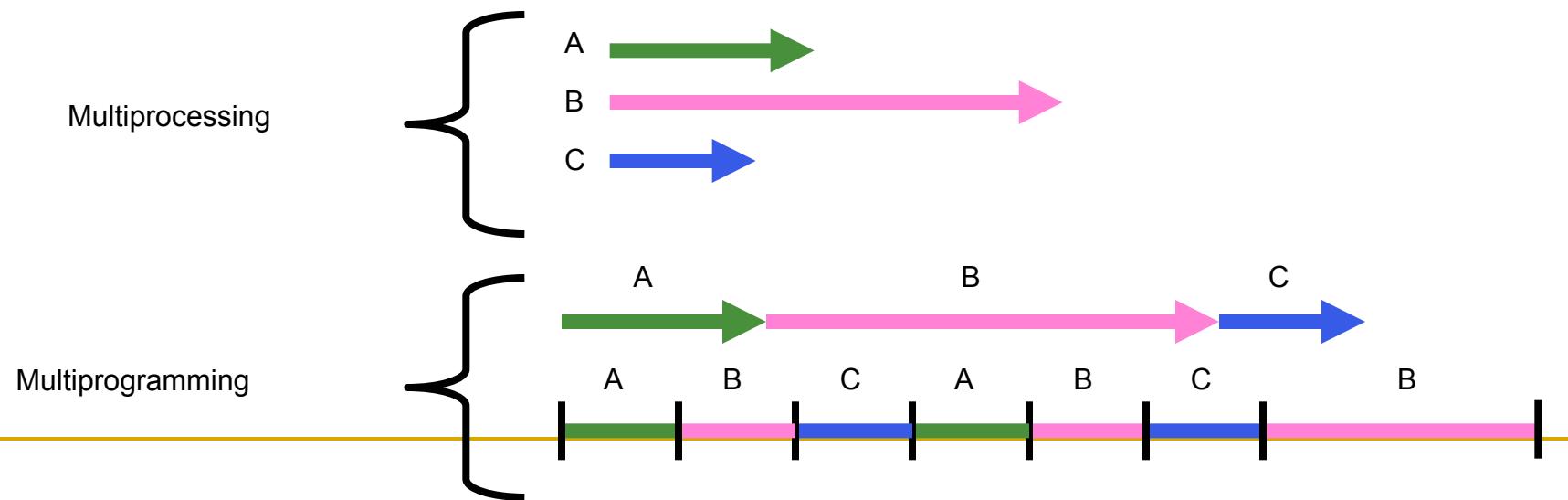
Multi(processing, programming, threading)

■ Definitions:

- Multiprocessing = Multiple CPUs
- Multiprogramming = Multiple Jobs or Processes
- Multithreading = Multiple threads per Process

■ What does it mean to run two threads “concurrently”?

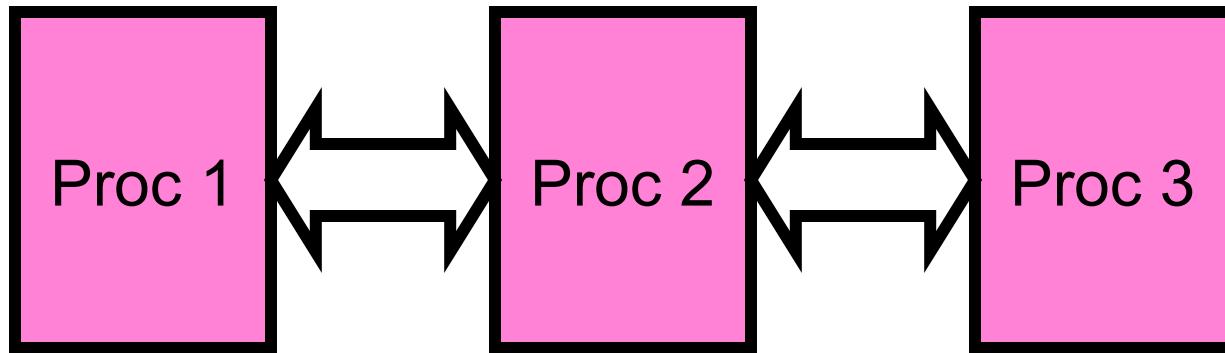
- Scheduler is free to run threads in any order and interleaving: FIFO, Random, ...
- Dispatcher can choose to run each thread to completion or time-slice in big chunks or small chunks



Cooperating Processes

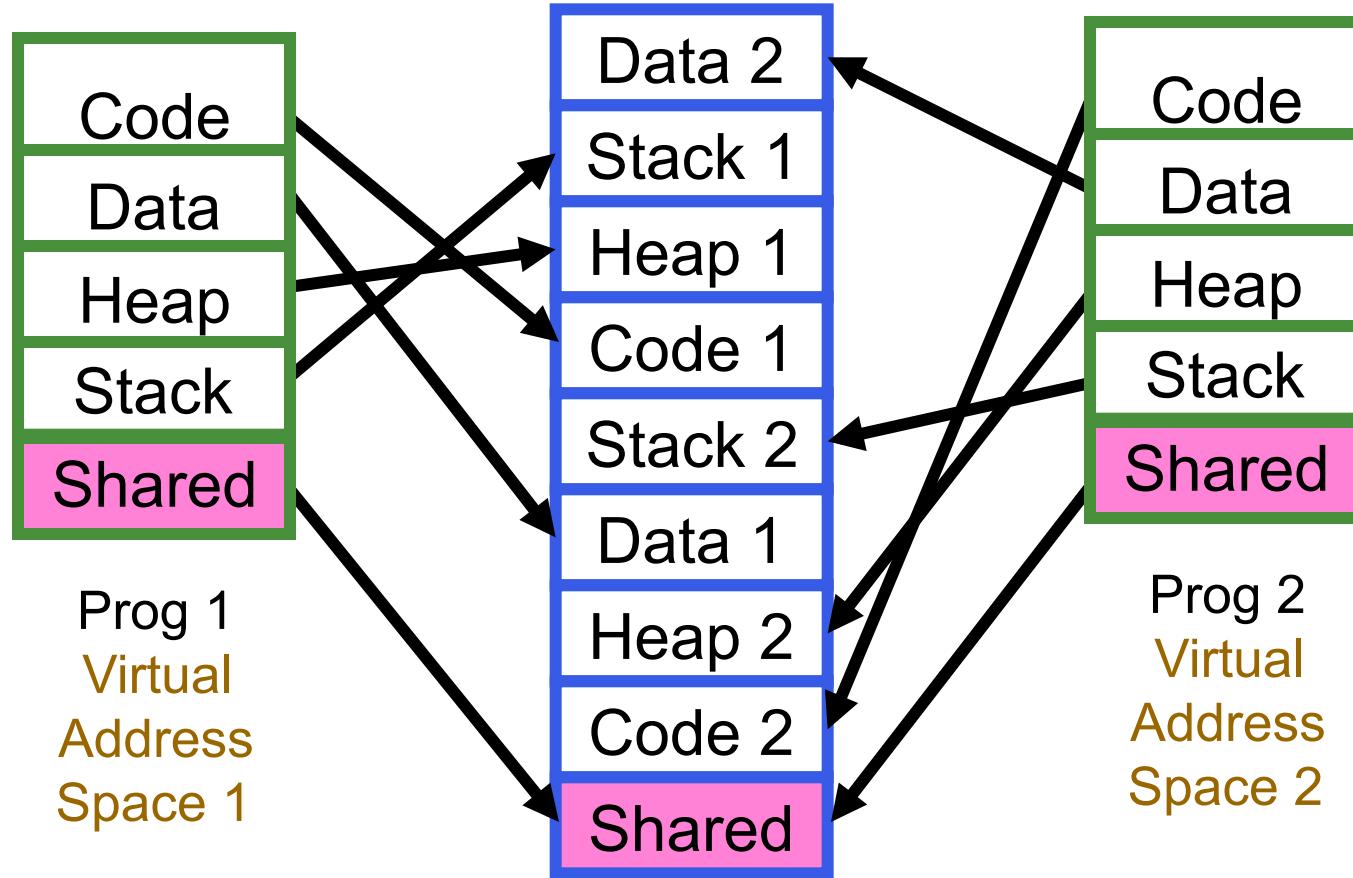
- Concurrent Processes can be
 - Independent processes
 - cannot affect or be affected by the execution of another process.
 - Cooperating processes
 - can affect or be affected by the execution of another process.
- Advantages of process cooperation:
 - Information sharing
 - Computation speedup
 - Modularity
 - Convenience(e.g. editing, printing, compiling)
- Concurrent execution requires
 - process communication and process synchronization

Interprocess Communication (IPC)



- Separate address space isolates processes
 - High Creation/Memory Overhead; (Relatively) High Context-Switch Overhead
- Mechanism for processes to communicate and synchronize actions.
 - Via shared memory - Accomplished by mapping addresses to common DRAM
 - Read and Write through memory
 - Via Messaging system - processes communicate without resorting to shared variables.
 - `send()` and `receive()` messages
 - Can be used over the network!
 - Messaging system and shared memory not mutually exclusive
 - can be used simultaneously within a single OS or a single process.

Shared Memory Communication



- Communication occurs by “simply” reading/writing to shared address page
 - Really low overhead communication
 - Introduces complex synchronization problems

Cooperating Processes via Message Passing

- IPC facility provides two operations.
 - send(*message*)** - message size can be fixed or variable
 - receive(*message*)**
- If processes P and Q wish to communicate, they need to:
 - establish a communication link between them
 - exchange messages via send/receive
- Fixed vs. Variable size message
 - Fixed message size - straightforward physical implementation, programming task is difficult due to fragmentation
 - Variable message size - simpler programming, more complex physical implementation.

Implementation Questions

- How are links established?
 - Can a link be associated with more than 2 processes?
 - How many links can there be between every pair of communicating processes?
 - What is the capacity of a link?
 - Fixed or variable size messages?
 - Unidirectional or bidirectional links?
-

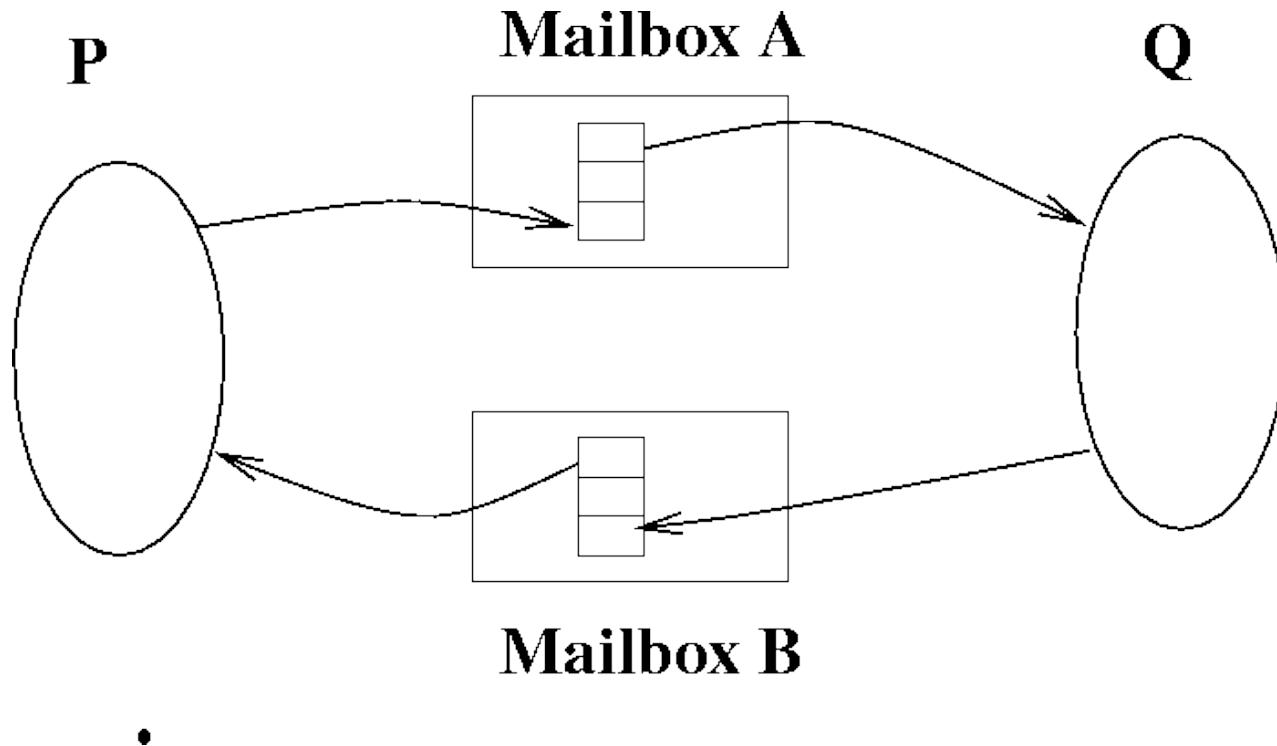
Direct Communication

- Sender and Receiver processes must name each other explicitly:
 - **send**(*P*, *message*) - send a message to process *P*
 - **receive**(*Q*, *message*) - receive a message from process *Q*
- Properties of communication link:
 - Links are established automatically.
 - A link is associated with exactly one pair of communicating processes.
 - Exactly one link between each pair.
 - Link may be unidirectional, usually bidirectional.

Indirect Communication

- Messages are directed to and received from mailboxes (also called ports)
 - Unique ID for every mailbox.
 - Processes can communicate only if they share a mailbox.
Send(A, message) /* send message to mailbox A */
Receive(A, message) /* receive message from mailbox A */
- Properties of communication link
 - Link established only if processes share a common mailbox.
 - Link can be associated with many processes.
 - Pair of processes may share several communication links
 - Links may be unidirectional or bidirectional

Indirect Communication using mailboxes



Mailboxes (cont.)

■ Operations

- create a new mailbox
- send/receive messages through mailbox
- destroy a mailbox

■ Issue: Mailbox sharing

- P1, P2 and P3 share mailbox A.
- P1 sends message, P2 and P3 receive... who gets message??

■ Possible Solutions

- disallow links between more than 2 processes
- allow only one process at a time to execute receive operation
- allow system to arbitrarily select receiver and then notify sender.

Message Buffering

- Link has some capacity - determine the number of messages that can reside temporarily in it.
- Queue of messages attached to link
 - Zero-capacity Queues: 0 messages
 - sender waits for receiver (synchronization is called *rendezvous*)
 - Bounded capacity Queues: Finite length of n messages
 - sender waits if link is full
 - Unbounded capacity Queues: Infinite queue length
 - sender never waits

Message Problems - Exception Conditions

- Process Termination
 - Problem: P(sender) terminates, Q(receiver) blocks forever.
 - Solutions:
 - System terminates Q.
 - System notifies Q that P has terminated.
 - Q has an internal mechanism(timer) that determines how long to wait for a message from P.
 - Problem: P(sender) sends message, Q(receiver) terminates.
In automatic buffering, P sends message until buffer is full or forever. In no-buffering scheme, P blocks forever.
 - Solutions:
 - System notifies P
 - System terminates P
 - P and Q use acknowledgement with timeout

Message Problems - Exception Conditions

■ Lost Messages

- OS guarantees retransmission
- sender is responsible for detecting it using timeouts
- sender gets an exception

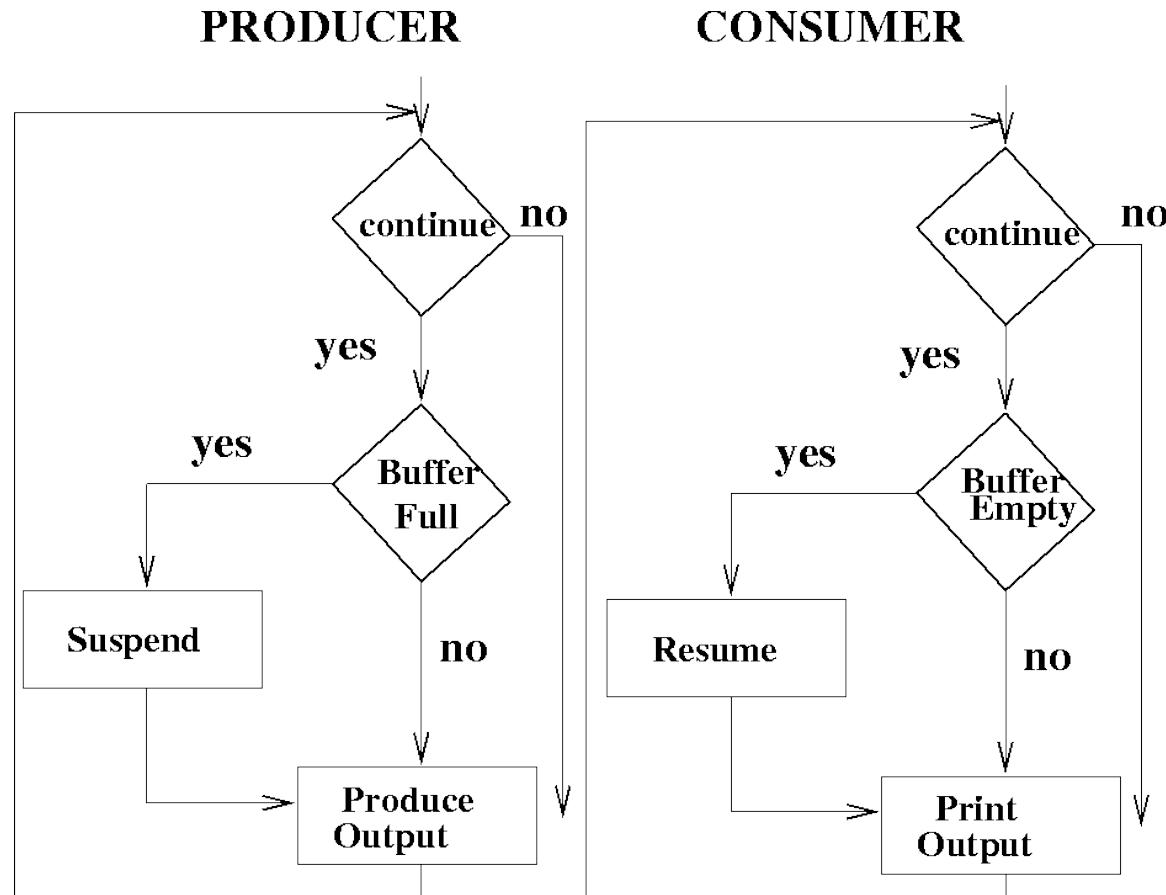
■ Scrambled Messages

- Message arrives from sender P to receiver Q, but information in message is corrupted due to noise in communication channel.
- Solution
 - need error detection mechanism, e.g. CHECKSUM
 - need error correction mechanism, e.g. retransmission

Producer-Consumer Problem

- Paradigm for cooperating processes;
 - producer process produces information that is consumed by a consumer process.
- We need buffer of items that can be filled by producer and emptied by consumer.
 - Unbounded-buffer places no practical limit on the size of the buffer. Consumer may wait, producer never waits.
 - Bounded-buffer assumes that there is a fixed buffer size. Consumer waits for new item, producer waits if buffer is full.
- Producer and Consumer must synchronize.

Producer-Consumer Problem



Bounded Buffer using IPC (messaging)

- Producer

 - repeat**

 - ...

 - produce an item in *nextp*;

 - ...

 - send**(*consumer*, *nextp*);

 - until** false;

- Consumer

 - repeat**

 - receive**(*producer*, *nextc*);

 - ...

 - consume item from *nextc*;

 - ...

 - until** false;

Bounded-buffer - Shared Memory Solution

■ Shared data

```
var n;  
type item = ....;  
var buffer: array[0..n-1] of item;  
in, out: 0..n-1;  
in := 0; out := 0; /* shared buffer = circular array */  
/* Buffer empty if in == out */  
/* Buffer full if (in+1) mod n == out */  
/* noop means 'do nothing' */
```

Bounded Buffer - Shared Memory Solution

- Producer process - creates filled buffers
 - repeat

...

produce an item in nextp

...

while $\text{in}+1 \bmod n = \text{out}$ **do** *noop*;

buffer[in] := nextp;

in := in+1 mod n;

until *false*;

Bounded Buffer - Shared Memory Solution

■ Consumer process - Empties filled buffers

repeat

```
while in = out do noop;  
nextc := buffer[out] ;  
out:= out+1 mod n;
```

...

consume the next item in *nextc*

...

until *false*

ICS 143 - Principles of Operating Systems



Lecture Set 3 - Process Synchronization
Prof. Nalini Venkatasubramanian
nalini@ics.uci.edu

Outline



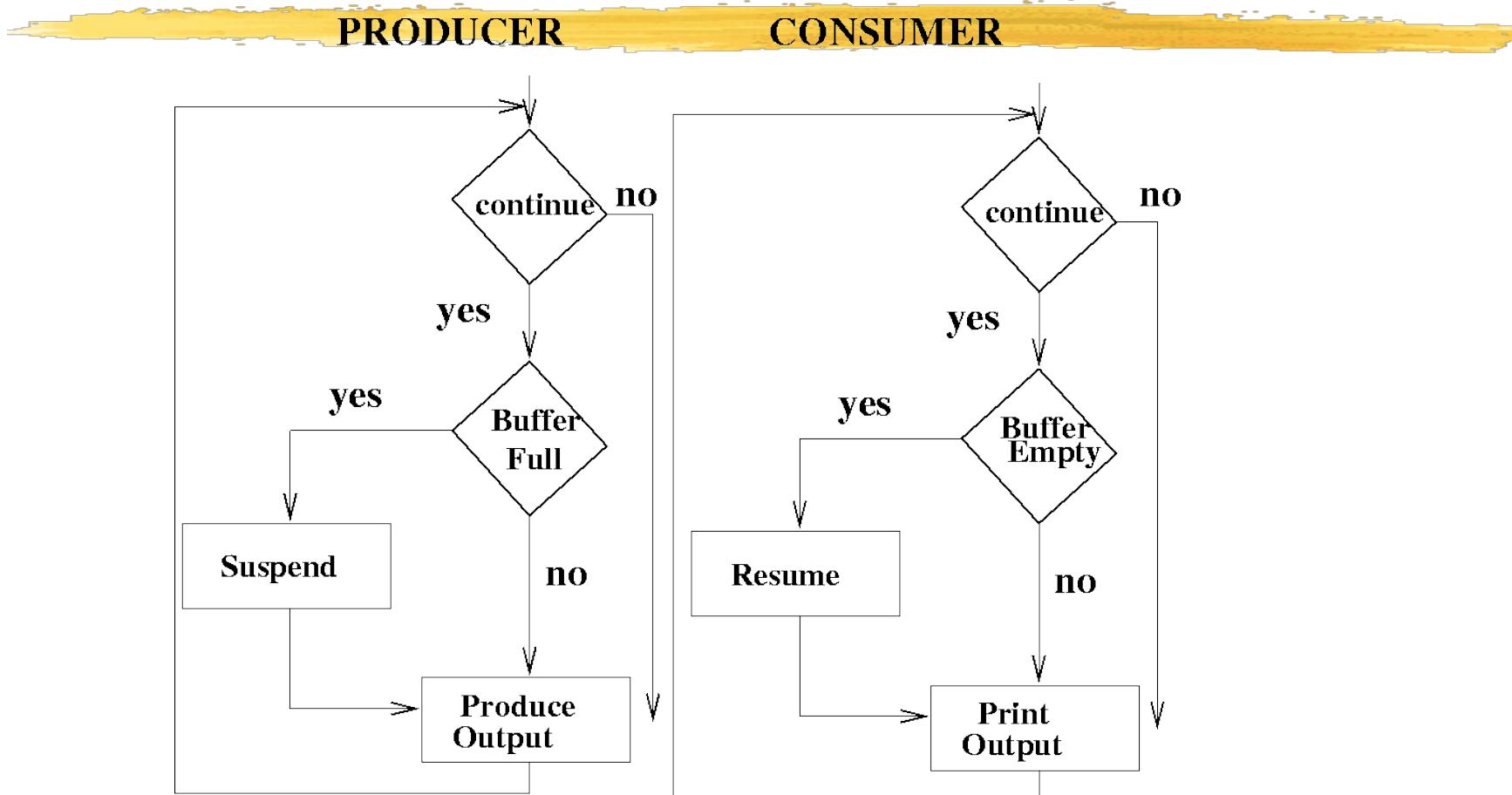
- The Critical Section Problem
- Synchronization Hardware
- Semaphores
- Classical Problems of Synchronization
- Critical Regions
- Monitors

Producer-Consumer Problem



- Paradigm for cooperating processes;
 - producer process produces information that is consumed by a consumer process.
- We need buffer of items that can be filled by producer and emptied by consumer.
 - Unbounded-buffer places no practical limit on the size of the buffer. Consumer may wait, producer never waits.
 - Bounded-buffer assumes that there is a fixed buffer size. Consumer waits for new item, producer waits if buffer is full.
- Producer and Consumer must synchronize.

Producer-Consumer Problem



Bounded Buffer using IPC (messaging)



| Producer

repeat

...

produce an item in *nextp*;

...

send(*consumer*, *nextp*);

until false;

| Consumer

repeat

receive(*producer*, *nextc*);

...

consume item from *nextc*;

...

until false;

Bounded-buffer - Shared Memory Solution



■ Shared data

```
var n;  
type item = ....;  
var buffer: array[0..n-1] of item;  
in, out: 0..n-1;  
in := 0; out := 0; /* shared buffer = circular array */  
/* Buffer empty if in == out */  
/* Buffer full if (in+1) mod n == out */  
/* noop means 'do nothing' */
```

Bounded Buffer - Shared Memory Solution



- Producer process - creates filled buffers

repeat

...

produce an item in *nextp*

...

while *in*+1 **mod** *n* = *out* **do** *noop*;

buffer[*in*] := *nextp*;

in := *in*+1 **mod** *n*;

until *false*;

Bounded Buffer - Shared Memory Solution



Consumer process - Empties filled buffers

repeat

while $in = out$ **do** *noop*;

$nextc := buffer[out]$;

$out := out + 1 \bmod n$;

...

consume the next item in $nextc$

...

until *false*

Shared data



- Concurrent access to shared data may result in data inconsistency.
- Maintaining data consistency requires mechanisms to ensure the orderly execution of cooperating processes.
- Shared memory solution to the bounded-buffer problem allows at most $(n-1)$ items in the buffer at the same time.

Bounded Buffer



- A solution that uses all N buffers is not that simple.
 - | Modify producer-consumer code by adding a variable *counter*, initialized to 0, incremented each time a new item is added to the buffer

■ Shared data

```
type item = ....;  
var buffer: array[0..n-1] of item;  
in, out: 0..n-1;  
counter: 0..n;  
in, out, counter := 0;
```

Bounded Buffer



- Producer process - creates filled buffers

repeat

...

produce an item in *nextp*

...

while *counter* = *n* **do** *noop*;

buffer[in] := *nextp*;

in := *in+1 mod n*;

counter := *counter+1*;

until *false*;

Bounded Buffer



- Consumer process - Empties filled buffers

```
repeat
    while counter = 0 do noop;
    nextc := buffer[out];
    out := out+1 mod n;
    counter := counter - 1;
    ...
    consume the next item in nextc
    ...
until false;
```

- The statements

```
counter := counter + 1;
counter := counter - 1;
```

must be executed *atomically*.

- Atomic Operations

- An operation that runs to completion or not at all.

Race Condition



- **counter++** could be implemented as

```
register1 = counter  
register1 = register1 + 1  
counter = register1
```

- **counter--** could be implemented as

```
register2 = counter  
register2 = register2 - 1  
counter = register2
```

- Consider this execution interleaving with “count = 5” initially (**we expect count = 5 in the end too**):

```
S0: producer execute register1 = counter      {register1 = 5}  
S1: producer execute register1 = register1 + 1  {register1 = 6}  
S2: consumer execute register2 = counter      {register2 = 5}  
S3: consumer execute register2 = register2 - 1  {register2 = 4}  
S4: producer execute counter = register1       {counter = 6 }  
S5: consumer execute counter = register2       {counter = 4 !!}
```

Problem is at the lowest level



- | If threads are working on separate data, scheduling doesn't matter:

Thread A

x = 1;

Thread B

y = 2;

- | However, What about (Initially, y = 12):

Thread A

x = 1;

x = y+1;

Thread B

y = 2;

y = y*2;

- | What are the possible values of x?
- | Or, what are the possible values of x below?

Thread A

x = 1;

Thread B

x = 2;

- | X could be non-deterministic (1, 2??)



The Critical-Section Problem



■ N processes all competing to use shared data.

- Structure of process P_i --- Each process has a code segment, called the critical section, in which the shared data is accessed.

```
repeat
    entry section    /* enter critical section */
    critical section /* access shared variables */
    exit section     /* leave critical section */
    remainder section /* do other work */
until false
```

■ Problem

- Ensure that when one process is executing in its critical section, no other process is allowed to execute in its critical section.

Solution: Critical Section Problem - Requirements



I Mutual Exclusion

- If process P_i is executing in its critical section, then no other processes can be executing in their critical sections.

I Progress

- If no process is executing in its critical section and there exists some processes that wish to enter their critical section, then the selection of the processes that will enter the critical section next cannot be postponed indefinitely.

I Bounded Waiting

- A bound must exist on the number of times that other processes are allowed to enter their critical sections after a process has made a request to enter its critical section and before that request is granted.

Solution: Critical Section Problem - Requirements



- Assume that each process executes at a nonzero speed in the critical section. That is, assume that each process finishes executing the critical section once entered
- No assumption concerning relative speed of the n processes.
- Assume that a process can get stuck in its remainder section indefinitely, e.g., in a non-terminating while loop

Solution: Critical Section Problem -- Initial Attempt



- Only 2 processes, P0 and P1
- General structure of process Pi (Pj)

```
repeat
    entry section
    critical section
    exit section
    remainder section
until false
```
- Processes may share some common variables to synchronize their actions.

Algorithm 1



I Shared Variables:

- | **var** *turn*: (0..1);
| initially *turn* = 0;
- | *turn* = *i* \Rightarrow P_i can enter its critical section

I Process P_i

```
repeat
    while turn <> i do no-op;
        critical section
        turn := j;
        remainder section
    until false
```

Satisfies mutual exclusion, but not progress.

Algorithm 1



- Satisfies mutual exclusion
 - The turn is equal to either i or j and hence one of P_i and P_j can enter the critical section
- Does not satisfy progress
 - Example: P_i finishes the critical section and then gets stuck indefinitely in its remainder section. Then P_j enters the critical section, finishes, and then finishes its remainder section. P_j then tries to enter the critical section again, but it cannot since turn was set to i by P_j in the previous iteration. Since P_i is stuck in the remainder section, turn will be equal to i indefinitely and P_j can't enter although it wants to. Hence no process is in the critical section and hence no progress.
- We don't need to discuss/consider bounded wait when progress is not satisfied

Algorithm 2



I Shared Variables

- | **var** *flag*: array (0..1) **of** boolean;
| initially *flag*[0] = *flag*[1] = false;
- | *flag*[*i*] = true \Rightarrow Pi ready to enter its critical section

I Process P_i

```
repeat
    flag[i] := true;
    while flag[j] do no-op;
        critical section
    flag[i]:= false;
        remainder section
until false
```

.

Algorithm 2



- Satisfies mutual exclusion
 - If P_i enters, then $\text{flag}[i] = \text{true}$, and hence P_j will not enter.
- Does not satisfy progress
 - Can block indefinitely.... Progress requirement not met
 - Example: There can be an interleaving of execution in which P_i and P_j both first set their flags to true and then both check the other process' flag. Therefore, both get stuck at the entry section
- We don't need to discuss/consider bounded wait when progress is not satisfied

Algorithm 3



I Shared Variables

- | **var** *flag*: array (0..1) **of** boolean;
| initially *flag*[0] = *flag*[1] = false;
- | *flag*[*i*] = true \Rightarrow Pi ready to enter its critical section

I Process P_i

```
repeat
    while flag[j] do no-op;
        flag[i] := true;                                critical section
        flag[i]:= false;                               remainder section
    until false
```

Algorithm 3



- ***Does not satisfy mutual exclusion***
 - Example: There can be an interleaving of execution in which both first check the other process' flag and see that it is false. Then they both enter the critical section.
- We don't need to discuss/consider progress and bounded wait when mutual exclusion is not satisfied

Algorithm 4



- | Combined Shared Variables of algorithms 1 and 2
- | Process Pi

repeat

flag[i] := true;

turn := j;

while (*flag[j]* **and** *turn=j*) **do** *no-op*;

critical section

flag[i]:= false;

remainder section

until *false*

YES!!! Meets all three requirements, solves the critical section problem for 2 processes.

Also called “Peterson’s solution”

Algorithm 4



- Satisfies mutual exclusion
 - If one process enters the critical section, it means that either the other process was not ready to enter or it was this process' turn to enter. In either case, the other process will not enter the critical section
- Satisfies progress
 - If one process exits the critical section, it sets its ready flag to false and hence the other process can enter. Moreover, there is no interleaving in the entry section that can block both.
- Satisfies bounded wait
 - If a process is waiting in the entry section, it will be able to enter at some point since the other process will either set its ready flag to false or will set to turn to this process.

Bakery Algorithm



■ Critical section for n processes

- | Before entering its critical section, process receives a number. Holder of the smallest number enters critical section.
- | If processes P_i and P_j receive the same number,
 - if $i \leq j$, then P_i is served first; else P_j is served first.
- | The numbering scheme always generates numbers in increasing order of enumeration; i.e. 1,2,3,3,3,4,4,5,5

Bakery Algorithm (cont.)



■ Notation -

- Lexicographic order(ticket#, process id#)
 - | $(a,b) < (c,d)$ if $(a < c)$ or if $((a=c) \text{ and } (b < d))$
 - | $\max(a_0, \dots, a_{n-1})$ is a number, k , such that $k \geq a_i$ for $i = 0, \dots, n-1$

■ Shared Data

var *choosing: array[0..n-1] of boolean;* (initialized to *false*)
number: array[0..n-1] of integer; (initialized to 0)

Bakery Algorithm (cont.)

```
repeat
    choosing[i] := true;
    number[i] := max(number[0], number[1],...,number[n-1]) +1;
    choosing[i] := false;
    for j := 0 to n-1
        do begin
            while choosing[j] do no-op;
            while number[j] <> 0
                and (number[j],j) < (number[i],i) do no-op;
        end;
        critical section
        number[i]:= 0;
        remainder section
    until false;
```

Supporting Synchronization

| <i>Programs</i> | <i>Shared Programs</i> |
|-------------------------|---|
| <i>Higher-level API</i> | <i>Locks Semaphores Monitors Send/Receive CCregions</i> |
| <i>Hardware</i> | <i>Load/Store Disable Ints Test&Set Comp&Swap</i> |

- We are going to implement various synchronization primitives using atomic operations
 - Everything is pretty painful if only atomic primitives are load and store
 - Need to provide inherent support for synchronization at the hardware level
 - Need to provide primitives useful at software/user level

Hardware Solutions for Synchronization



- Load/store - Atomic Operations required for synchronization
 - | Showed how to protect a critical section with only atomic load and store ⇒ pretty complex!
- Mutual exclusion solutions presented depend on memory hardware having read/write cycle.
 - If multiple reads/writes could occur to the same memory location at the same time, this would not work.
 - Processors with caches but no cache coherency cannot use the solutions
- In general, it is impossible to build mutual exclusion without a primitive that provides some form of mutual exclusion.
 - | How can this be done in the hardware???
 - | How can this be simplified in software???

Synchronization Hardware



- Test and modify the content of a word atomically - **Test-and-set instruction**

```
function Test-and-Set (var target: boolean): boolean;  
begin  
    Test-and-Set := target;  
    target := true;  
end;
```

- Similarly “**SWAP**” instruction

Mutual Exclusion with Test-and-Set



- Shared data: var lock: boolean (initially false)
- Process Pi

```
repeat
    while Test-and-Set (lock) do no-op;
        critical section
        lock := false;
        remainder section
    until false;
```

Bounded Waiting Mutual Exclusion with Test-and-Set

```
var j : 0..n-1;
    key : boolean;
repeat
    waiting [i] := true; key := true;
    while waiting[i] and key do key := Test-and-Set(lock);
    waiting [i] := false;
    critical section
        j := i+1 mod n;
        while (j <> i) and (not waiting[j]) do j := j + 1 mod n;
        if j = i then lock := false;
        else waiting[j] := false;
    remainder section
until false;
```

Hardware Support: Other examples



- **swap (&address, register)** { /* x86 */
 temp = M[address];
 M[address] = register;
 register = temp;
}
- **compare&swap (&address, reg1, reg2)** { /* 68000 */
 if (reg1 == M[address]) {
 M[address] = reg2;
 return success;
 } else {
 return failure;
 }
}
- **load-linked&store conditional(&address)** {
 /* R4000, alpha */
 loop:
 ll r1, M[address];
 movi r2, 1; /* Can do arbitrary comp */
 sc r2, M[address];
 beqz r2, loop;
}

Mutex Locks



- Previous solutions are complicated and generally inaccessible to application programmers
- OS designers build software tools to solve critical section problem
- Simplest is mutex lock
- Protect a critical section by first **acquire()** a lock then **release()** the lock
 - Boolean variable indicating if lock is available or not
- Calls to **acquire()** and **release()** must be atomic
 - Usually implemented via hardware atomic instructions
 - But this solution requires **busy waiting**
 - This lock therefore called a **spinlock**

acquire() and release()

- `acquire() {
 while (!available)
 ; /* busy wait */
 available = false;;
}`

- Semantics of

- `release() {
 available = true;
}`

- Critical section implementation

```
do {  
    acquire lock  
    critical section  
    release lock  
    remainder section  
} while (true);
```

Semaphore



- Semaphore S - integer variable (non-negative)
 - used to represent number of abstract resources
- Can only be accessed via two indivisible (atomic) operations
 - $\text{wait}(S)$: **while** $S \leq 0$ **do** no-op
 $S := S - 1;$
 - $\text{signal}(S)$: $S := S + 1;$
 - P or *wait* used to acquire a resource, waits for semaphore to become positive, then decrements it by 1
 - V or *signal* releases a resource and increments the semaphore by 1, waking up a waiting P, if any
 - If P is performed on a $count \leq 0$, process must wait for V or the release of a resource.

P(): "proberen" (to test) ; V() "verhogen" (to increment) in Dutch

Example: Critical Section for n Processes



I Shared variables

```
var mutex: semaphore  
initially mutex = 1
```

I Process P_i

```
repeat  
    wait(mutex);  
        critical section  
    signal(mutex);  
        remainder section  
until false
```

Semaphore as a General Synchronization Tool



- Execute B in P_j only after A execute in P_i
- Use semaphore flag initialized to 0
- Code:

| | |
|------------------------------|----------------------------|
| P_i | P_j |
| : | : |
| : | : |
| A | $\text{wait}(\text{flag})$ |
| $\text{signal}(\text{flag})$ | B |

Problem...



- Locks prevent conflicting actions on shared data
 - | Lock before entering critical section and before accessing shared data
 - | Unlock when leaving, after accessing shared data
 - | Wait if locked
- All Synchronization involves waiting
 - | **Busy Waiting**, uses CPU that others could use. This type of semaphore is called a *spinlock*.
 - | Waiting thread may take cycles away from thread holding lock (no one wins!)
 - | OK for short times since it prevents a context switch.
 - | Priority Inversion: If busy-waiting thread has higher priority than thread holding lock \Rightarrow no progress!
 - | Should *sleep* if waiting for a long time
- For longer runtimes, need to modify P and V so that processes can *block* and *resume*.

Semaphore Implementation



- Define a semaphore as a record

```
type semaphore = record  
    value: integer;  
    L: list of processes;  
end;
```

- Assume two simple operations

- | *block* suspends the process that invokes it.
- | *wakeup(P)* resumes the execution of a blocked process *P*.

Semaphore Implementation(cont.)

- I Semaphore operations are now defined as

*wait (S): S.value := S.value -1;
if S.value < 0
then begin
 add this process to S.L;
 block;
end;*

*signal (S): S.value := S.value +1;
if S.value <= 0
then begin
 remove a process P from S.L;
 wakeup(P);
end;*

Block/Resume Semaphore Implementation



- If process is blocked, enqueue PCB of process and call scheduler to run a different process.
- Semaphores are executed atomically;
 - | no two processes execute *wait* and *signal* at the same time.
 - | Mutex can be used to make sure that two processes do not change count at the same time.
 - If an interrupt occurs while mutex is held, it will result in a long delay.
 - Solution: Turn off interrupts during critical section.

Deadlock and Starvation



- | Deadlock - two or more processes are waiting indefinitely for an event that can be caused by only one of the waiting processes.

- | Let S and Q be semaphores initialized to 1

| | |
|--------------------|--------------------|
| P_0 | P_1 |
| <i>wait(S);</i> | <i>wait(Q);</i> |
| <i>wait(Q);</i> | <i>wait(S);</i> |
| : | : |
| <i>signal (S);</i> | <i>signal (Q);</i> |
| <i>signal (Q);</i> | <i>signal (S);</i> |

- | Starvation- indefinite blocking. A process may never be removed from the semaphore queue in which it is suspended.

Two Types of Semaphores



- Counting Semaphore - integer value can range over an unrestricted domain.
- Binary Semaphore - integer value can range only between 0 and 1; simpler to implement.
- Can implement a counting semaphore S as a binary semaphore.

Classical Problems of Synchronization



- Bounded Buffer Problem
- Readers and Writers Problem
- Dining-Philosophers Problem

Bounded Buffer Problem



■ Shared data

```
type item = ....;  
var buffer: array[0..n-1] of item;  
full, empty, mutex : semaphore;  
nextp, nextc :item;  
full := 0; empty := n; mutex := 1;
```

Bounded Buffer Problem



■ Producer process - creates filled buffers

repeat

...

produce an item in *nextp*

...

wait (empty);

wait (mutex);

...

add *nextp* to buffer

...

signal (mutex);

signal (full);

until *false*;

Bounded Buffer Problem



■ Consumer process - Empties filled buffers

repeat

wait (full);

wait (mutex);

...

remove an item from buffer to nextc

...

signal (mutex);

signal (empty);

...

consume the next item in nextc

...

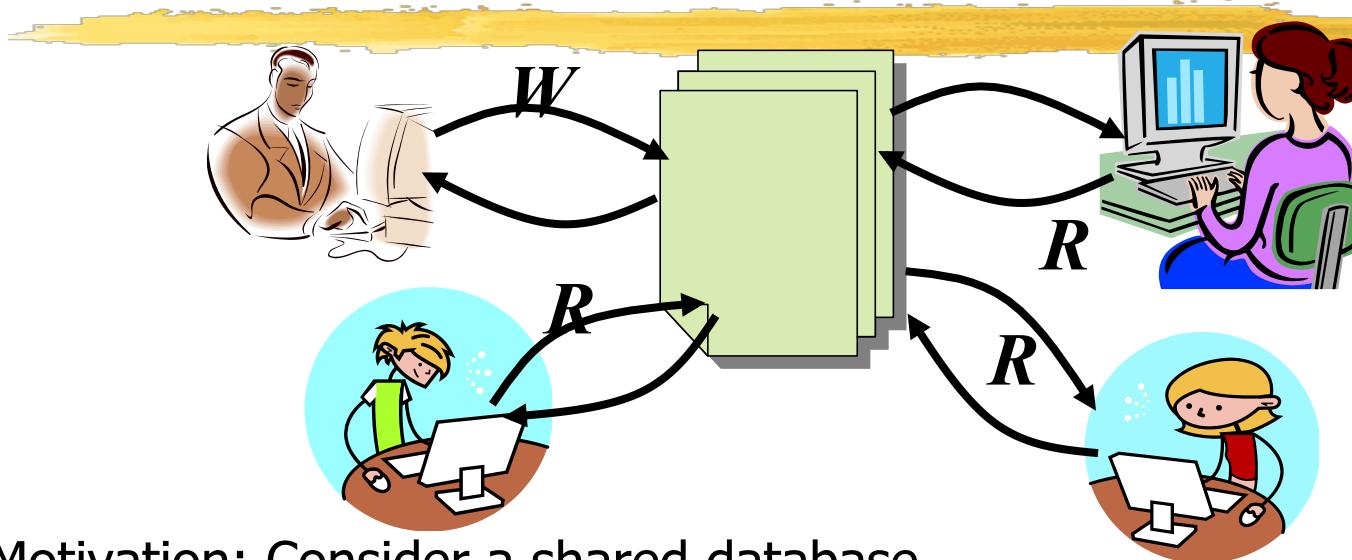
until false;

Discussion



- ASymmetry?
 - Producer does: P(empty), V(full)
 - Consumer does: P(full), V(empty)
- Is order of P's important?
 - Yes! Can cause deadlock
- Is order of V's important?
 - No, except that it might affect scheduling efficiency

Readers/Writers Problem



- Motivation: Consider a shared database
 - Two classes of users:
 - Readers – never modify database
 - Writers – read and modify database
 - Is using a single lock on the whole database sufficient?
 - Like to have many readers at the same time
 - Only one writer at a time

Readers-Writers Problem



■ Shared Data

```
var mutex, wrt: semaphore (=1);  
readcount: integer (= 0);
```

■ Writer Process

```
wait(wrt);  
...  
writing is performed  
...  
signal(wrt);
```

Readers-Writers Problem



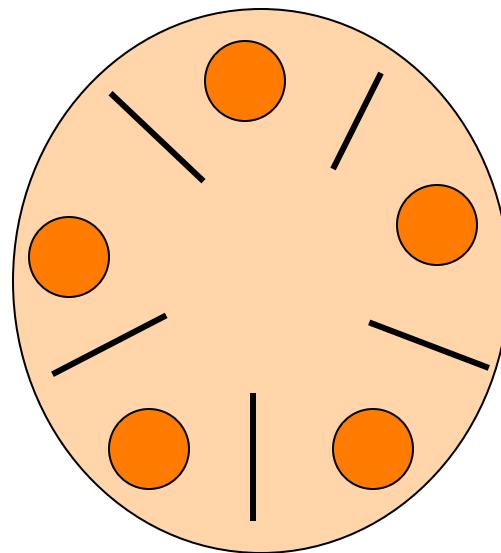
Reader process

```
wait(mutex);
readcount := readcount +1;
if readcount = 1 then wait(wrt);
signal(mutex);

...
reading is performed
```

```
...
wait(mutex);
readcount := readcount - 1;
if readcount = 0 then signal(wrt);
signal(mutex);
```

Dining-Philosophers Problem



Shared Data

```
var chopstick: array [0..4] of semaphore (=1 initially);
```

Dining Philosophers Problem



■ Philosopher i :

repeat

wait (chopstick[i]);

wait (chopstick[$i+1 \bmod 5$]);

...

eat

...

signal (chopstick[i]);

signal (chopstick[$i+1 \bmod 5$]);

...

think

...

until *false;*

Higher Level Synchronization



■ Timing errors are still possible with semaphores

| Example 1

```
signal (mutex);  
...  
critical region
```

```
wait (mutex);  
...
```

| Example 2

```
wait(mutex);  
...  
critical region
```

```
wait (mutex);  
...
```

| Example 3

```
wait(mutex);  
...  
critical region
```

```
Forgot to signal  
...
```

Motivation for Other Sync. Constructs



- Semaphores are a huge step up from loads and stores
 - Problem is that semaphores are dual purpose:
 - They are used for both mutex and scheduling constraints
 - Example: the fact that flipping of P's in bounded buffer gives deadlock is not immediately obvious. How do you prove correctness to someone?
- Idea: allow manipulation of a shared variable only when condition (if any) is met – ***conditional critical region***
- Idea : Use *locks* for mutual exclusion and *condition variables* for scheduling constraints
 - ***Monitor:*** a lock (for mutual exclusion) and zero or more condition variables (for scheduling constraints) to manage concurrent access to shared data
 - Some languages like Java provide this natively

Conditional Critical Regions



- High-level synchronization construct
- A shared variable v of type T is declared as:

var v : **shared** T

- Variable v is accessed only inside statement

region v **when** B **do** S

where B is a boolean expression.

While statement S is being executed, no other process can access variable v .

Critical Regions (cont.)



- Regions referring to the same shared variable exclude each other in time.
- When a process tries to execute the region statement,
region v when B do S
 - the Boolean expression *B* is evaluated.
 - If *B* is true, statement *S* is executed.
 - If it is false, the process is delayed until *B* becomes true and no other process is in the region associated with *v*.

Example - Bounded Buffer



I Shared variables

```
var buffer: shared record  
    pool:array[0..n-1] of item;  
    count,in,out: integer;  
end;
```

I Producer Process inserts *nextp* into the shared buffer

```
region buffer when count < n  
    do begin  
        pool[in] := nextp;  
        in := in+1 mod n;  
        count := count + 1;  
    end;
```

Bounded Buffer Example



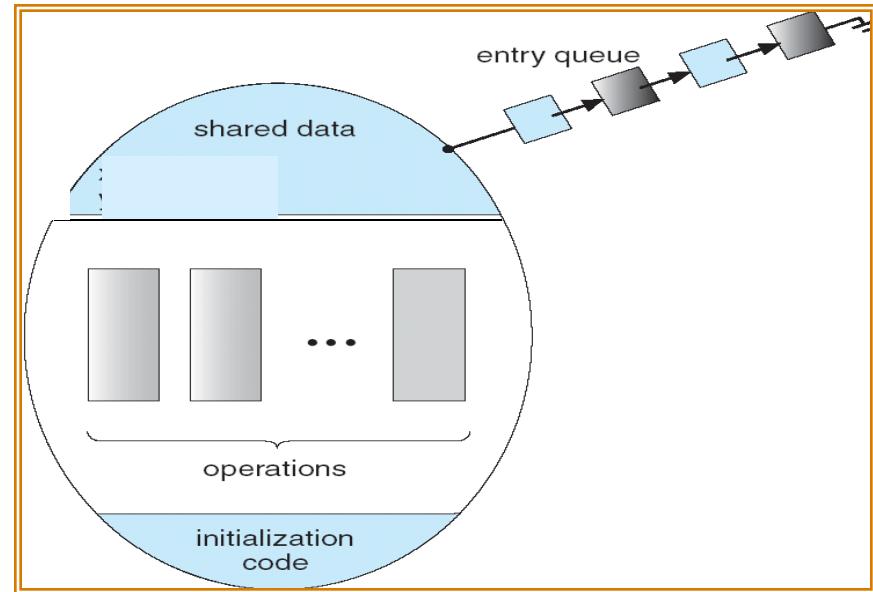
- Consumer Process removes an item from the shared buffer and puts it in *nextc*

```
region buffer when count > 0
    do begin
        nextc := pool[out];
        out := out+1 mod n;
        count := count -1;
    end;
```

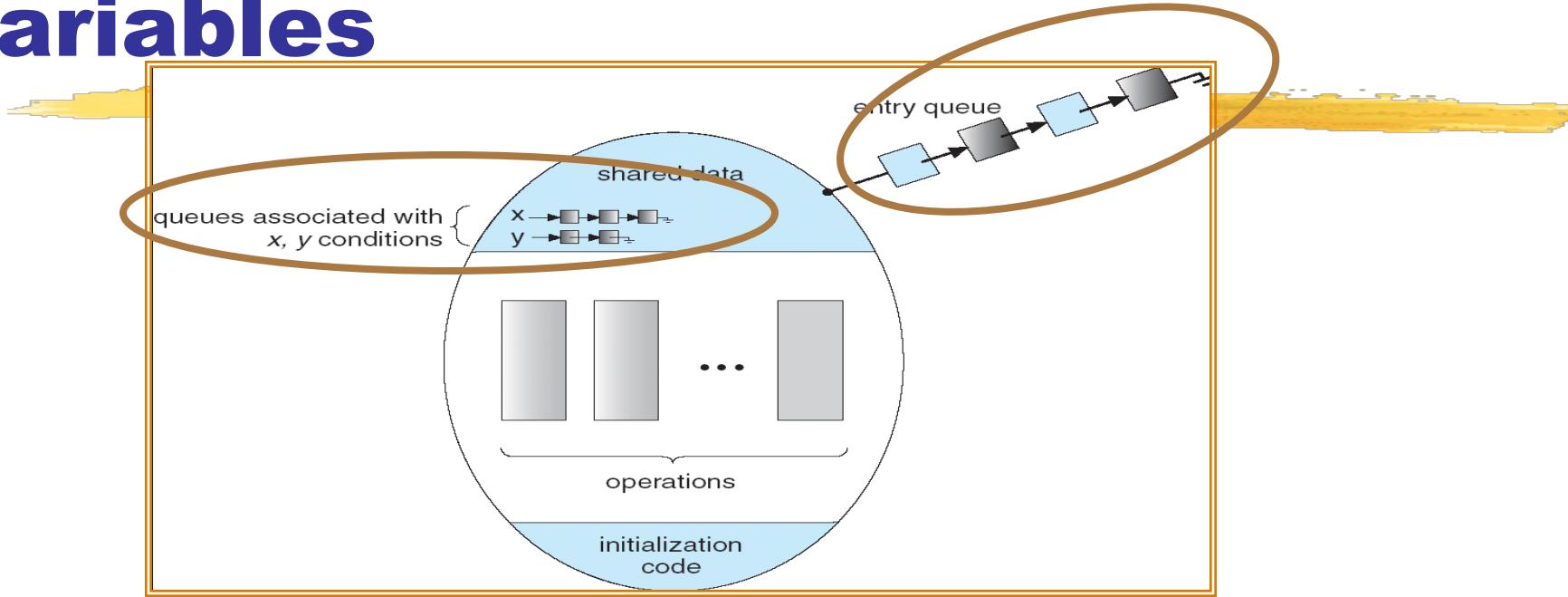
Monitors

High-level synchronization construct that allows the safe sharing of an abstract data type among concurrent processes.

```
type monitor-name = monitor  
variable declarations  
procedure entry P1 (...);  
  begin ... end;  
procedure entry P2 (...);  
  begin ... end;  
  ...  
procedure entry Pn(...);  
  begin ... end;  
begin  
  initialization code  
end.
```



Monitor with Condition Variables



- Lock: the lock provides mutual exclusion to shared data
 - Always acquire before accessing shared data structure
 - Always release after finishing with shared data
 - Lock initially free
- Condition Variable: a queue of threads waiting for something *inside* a critical section
 - Key idea: make it possible to go to sleep inside critical section by atomically releasing lock at time we go to sleep

Monitors with condition variables



- To allow a process to wait within the monitor, a condition variable must be declared, as:

var *x,y*: *condition*

- Condition variable can only be used within the operations *wait* and *signal*. Queue is associated with condition variable.
 - The operation
`x.wait;`
means that the process invoking this operation is suspended until another process invokes
`x.signal;`
 - The `x.signal` operation resumes exactly one suspended process. If no process is suspended, then the signal operation has no effect.

Dining Philosophers

```
type dining-philosophers= monitor
var state: array[0..4] of (thinking, hungry, eating);
var self: array[0..4] of condition;
// condition where philosopher I can delay himself when hungry
but // is unable to obtain chopstick(s)
procedure entry pickup (i :0..4);
begin
    state[i] := hungry;
    test(); //test that your left and right neighbors are not eating
    if state [i] <> eating then self [i].wait;
end;

procedure entry putdown (i:0..4);
begin
    state[i] := thinking;
    test (i + 4 mod 5 ); // signal one neighbor
    test (i + 1 mod 5 ); // signal other neighbor
end;
```

Dining Philosophers (cont.)

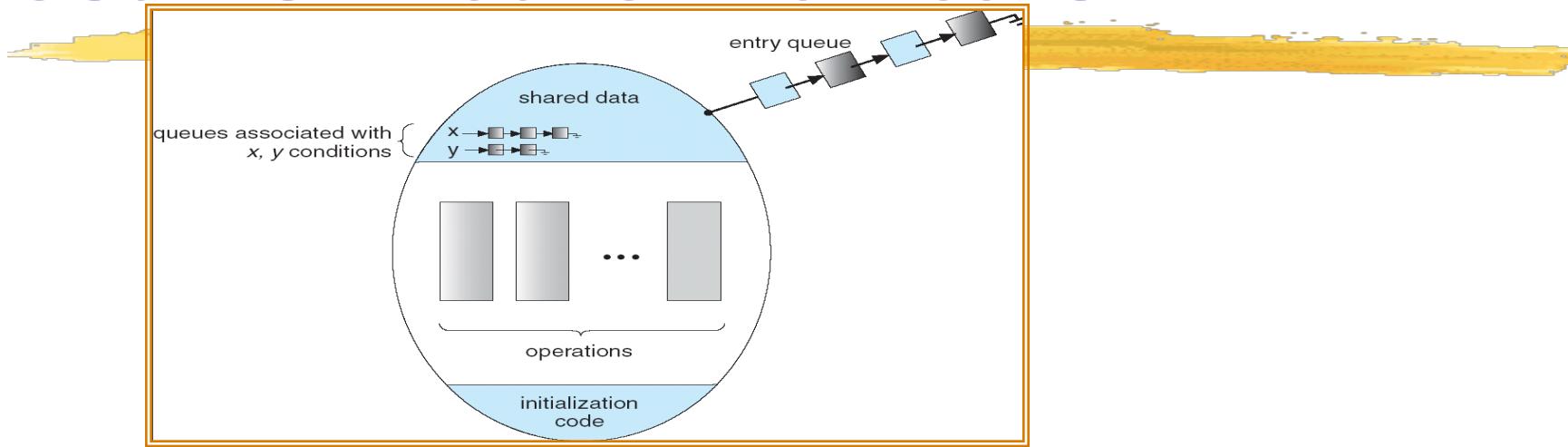
```
procedure test (k :0..4);
begin
  if state [k + 4 mod 5] <> eating
    and state [k ] = hungry
    and state [k + 1 mod 5] <> eating
  then
    begin
      state[k] := eating;
      self [k].signal;
    end;
  end;

begin
  for i := 0 to 4
    do state[i] := thinking;
end;
```

Additional (extra) slides



Mesa vs. Hoare monitors



Who proceeds next – signaler or waiter?

- I Hoare-style monitors(most textbooks):
 - | Signaler gives lock, CPU to waiter; waiter runs immediately
 - | Waiter gives up lock, processor back to signaler when it exits critical section or if it waits again
- I Mesa-style monitors (most real operating systems):
 - | Signaler keeps lock and processor
 - | Waiter placed on ready queue with no special priority
 - | Practically, need to check condition again after wait (condition may no longer be true!)

Implementing S (counting sem.) as a Binary Semaphore



■ Data Structures

```
var S1 : binary-semaphore;  
      S2 : binary-semaphore;  
      S3 : binary-semaphore;  
      C: integer;
```

■ Initialization

```
S1 = S3 =1;  
S2 = 0;  
C = initial value of semaphore S;
```

Implementing S



Wait operation

```
wait(S3);  
wait(S1);  
C := C-1;  
if C < 0  
then begin  
    signal(S1);  
    wait(S2);  
end  
else signal(S1);  
signal(S3);
```

Signal operation

```
wait(S1);  
C := C + 1;  
if C <= 0 then signal(S2);  
signal(S1);
```

Implementing Regions



■ Region x when B do S

```
var mutex, first-delay, second-delay: semaphore;  
first-count, second-count: integer;
```

■ Mutually exclusive access to the critical section is provided by mutex.

If a process cannot enter the critical section because the Boolean expression B is false,

it initially waits on the first-delay semaphore;
moved to the second-delay semaphore before it is allowed to
reevaluate B .

Implementation



- Keep track of the number of processes waiting on *first-delay* and *second-delay*, with *first-count* and *second-count* respectively.
- The algorithm assumes a FIFO ordering in the queueing of processes for a semaphore.
- For an arbitrary queueing discipline, a more complicated implementation is required.

Implementing Regions

```
wait(mutex);
while not B
  do begin
    first-count := first-count +1;
    if second-count > 0
      then signal (second-delay);
      else signal (mutex);
    wait(first-delay);
    first-count := first-count -1;
    second-count := second-count + 1;
    if first-count > 0 then signal (first-delay)
      else signal (second-delay);
    wait(second-delay);
    second-count := second-count -1;
  end;
S;
if first-count > 0 then signal (first-delay);
  else if second-count > 0
    then signal (second-delay);
    else signal (mutex);
```

CS 143A - Principles of Operating Systems



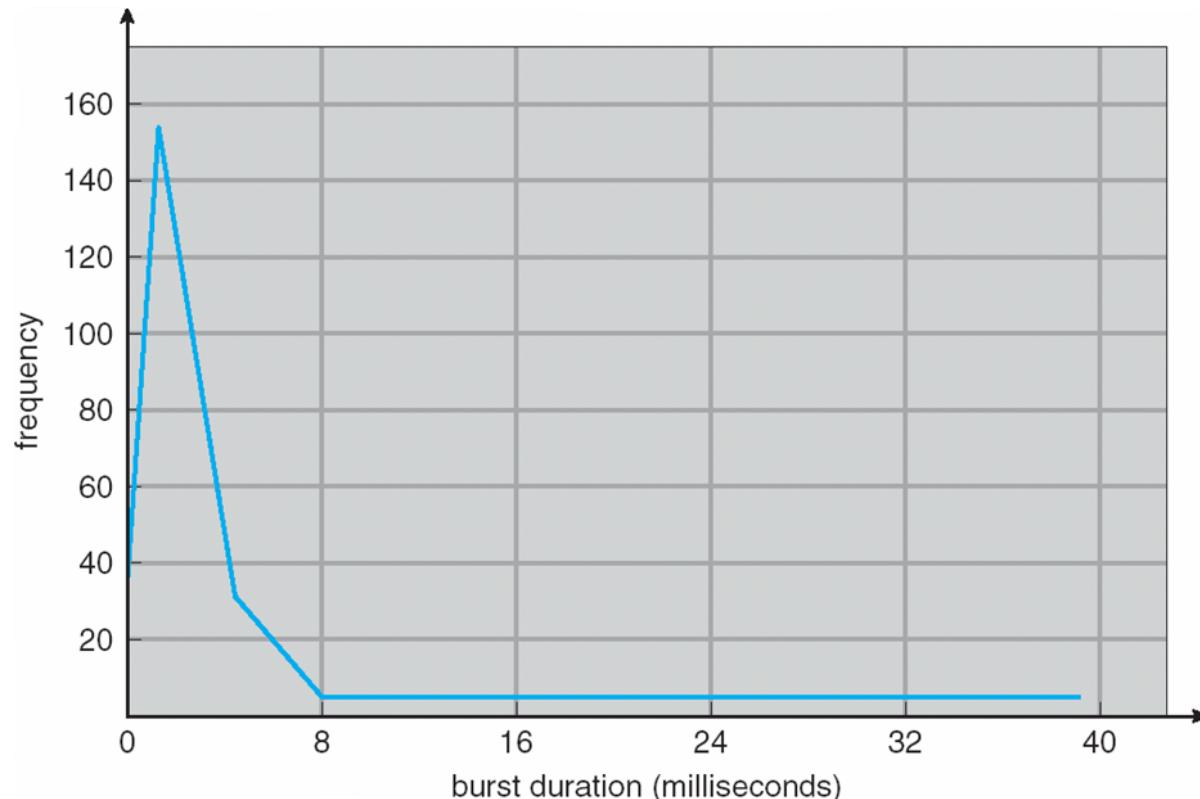
Lecture 4 - CPU Scheduling
Prof. Nalini Venkatasubramanian
nalini@ics.uci.edu

Outline



- Basic Concepts
- Scheduling Objectives
- Levels of Scheduling
- Scheduling Criteria
- Scheduling Algorithms
 - | FCFS, Shortest Job First, Priority, Round Robin, Multilevel
- Multiple Processor Scheduling
- Real-time Scheduling
- Algorithm Evaluation

CPU Burst Distribution

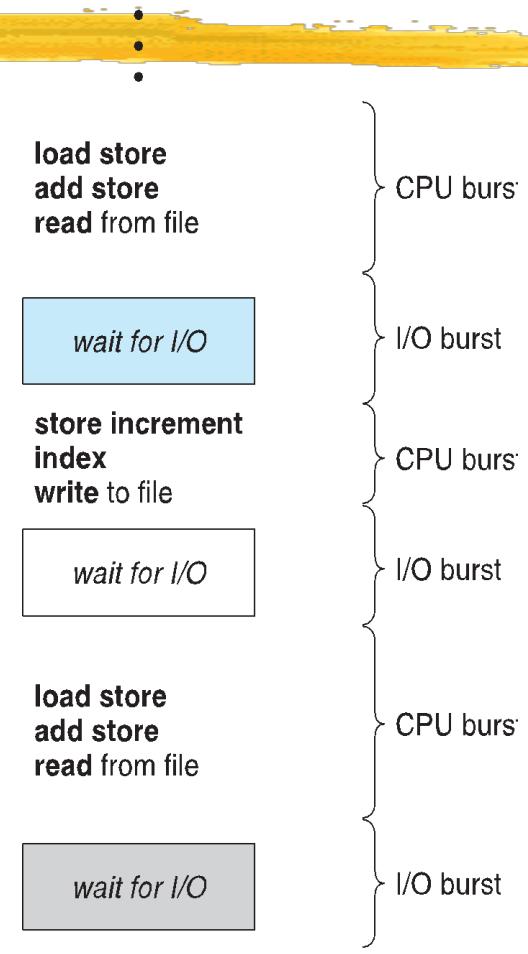


Basic Concepts

■ Maximum CPU utilization obtained with multiprogramming.

■ CPU-I/O Burst Cycle

- | Process execution consists of a cycle of CPU execution and I/O wait.



Scheduling Objectives



- Enforcement of fairness
 - in allocating resources to processes
- Enforcement of priorities
- Make best use of available system resources
- Give preference to processes holding key resources.
- Give preference to processes exhibiting good behavior.
- Degrade gracefully under heavy loads.

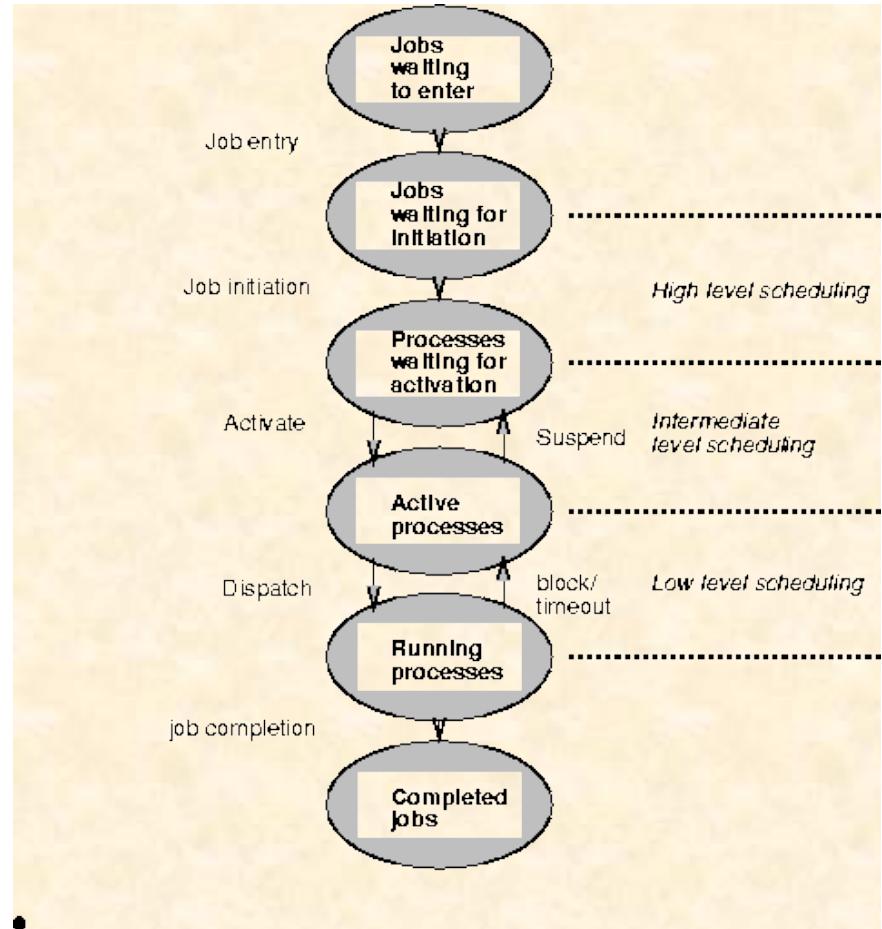
Program Behavior Issues



- I/O boundedness
 - | short burst of CPU before blocking for I/O
- CPU boundedness
 - | extensive use of CPU before blocking for I/O
- Urgency and Priorities
- Frequency of preemption
- Process execution time
- Time sharing
 - | amount of execution time process has already received.

Levels of Scheduling

- High Level / Job Scheduling
 - Selects jobs allowed to compete for CPU and other system resources.
- Intermediate Level / Medium Term Scheduling
 - Selects which jobs to temporarily suspend/resume to smooth fluctuations in system load.
- Low Level (CPU) Scheduling or Dispatching
 - Selects the ready process that will be assigned the CPU.
 - Ready Queue contains PCBs of processes.



CPU Scheduler



- Selects from among the processes in memory that are ready to execute, and allocates the CPU to one of them.

- Non-preemptive Scheduling

- Once CPU has been allocated to a process, the process keeps the CPU until
 - Process exits OR
 - Process switches to waiting state

- Preemptive Scheduling

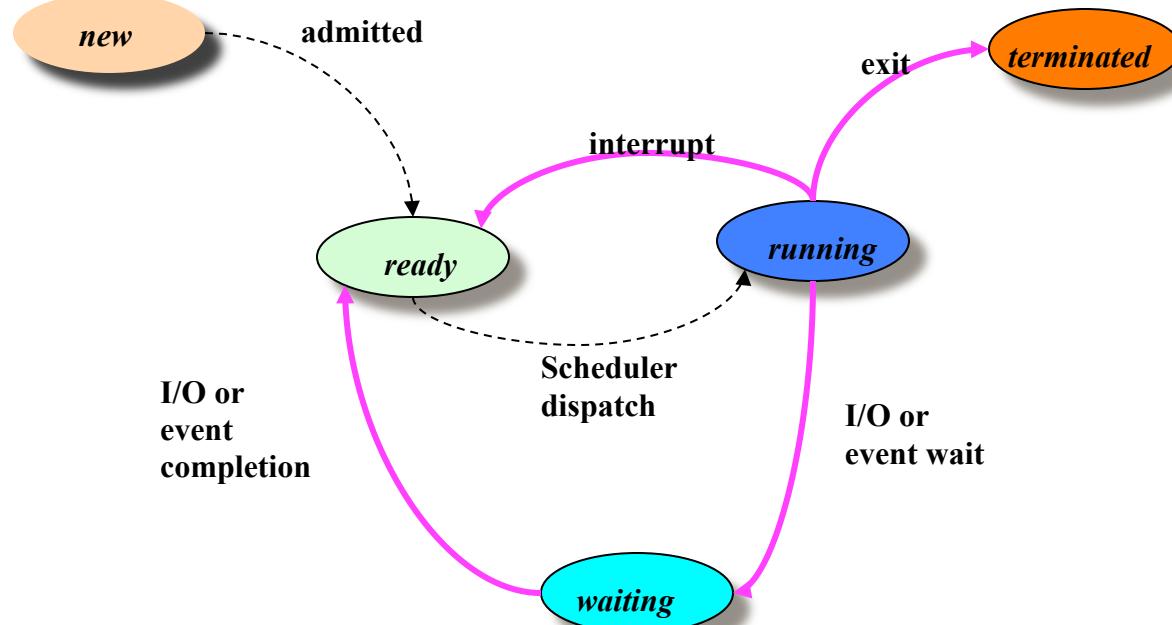
- Process can be interrupted and must release the CPU.
 - Need to coordinate access to shared data

CPU Scheduling Decisions



- CPU scheduling decisions may take place when a process:
 1. switches from running state to waiting state
 2. switches from running state to ready state
 3. switches from waiting to ready
 4. terminates
- Scheduling under 1 and 4 is non-preemptive.
- All other scheduling is preemptive.

CPU scheduling decisions



Dispatcher



- OS dispatcher module gives control of the CPU to the process selected by the short-term scheduler. This involves:
 - switching context
 - switching to user mode
 - jumping to the proper location in the user program to restart (continue) that program

- Dispatch Latency:
 - | time it takes for the dispatcher to stop one process and start another running.
 - | Dispatcher must be fast.

Scheduling Criteria



■ CPU Utilization

- | Keep the CPU and other resources as busy as possible

■ Throughput

- | # of processes that complete their execution per time unit.

■ Turnaround time

- | amount of time to execute a particular process from its entry time.

Scheduling Criteria (cont.)



■ Waiting time

- | amount of time a process has been waiting in the ready queue.

■ Response Time (in a time-sharing environment)

- | amount of time it takes from when a request was submitted until the first response is produced, NOT output.

Optimization Criteria



- Optimize overall system
 - Max CPU Utilization
 - Max Throughput
- Optimize individual processes' performance
 - Min Turnaround time
 - Min Waiting time
 - Min Response time

First Come First Serve (FCFS) Scheduling



- Policy: Process that requests the CPU *FIRST* is allocated the CPU *FIRST*.
 - FCFS is a non-preemptive scheduling algorithm.
- Implementation - using FIFO queues
 - incoming process is added to the tail of the queue.
 - Process selected for execution is taken from head of queue.
- Performance metric - Average waiting time in queue.
- Gantt Charts are used to visualize schedules.

First-Come, First-Served(FCFS) Scheduling

Example

| Process | Burst Time |
|---------|------------|
| P1 | 24 |
| P2 | 3 |
| P3 | 3 |

Gantt Chart for Schedule



- Suppose the arrival order for the processes is
 - | P1, P2, P3
- Waiting time
 - | P1 = 0;
 - | P2 = 24;
 - | P3 = 27;
- Average waiting time
 - | $(0+24+27)/3 = 17$

FCFS Scheduling (cont.)

Example

| Process | Burst Time |
|---------|------------|
| P1 | 24 |
| P2 | 3 |
| P3 | 3 |

Gantt Chart for Schedule



Suppose the arrival order for the processes is

| P2, P3, P1

Waiting time

| P1 = 6; P2 = 0; P3 = 3;

Average waiting time

| $(6+0+3)/3 = 3$, better..

Convoy Effect:

- short process behind long process, e.g. 1 CPU bound process, many I/O bound processes.

Shortest-Job-First (SJF) Scheduling



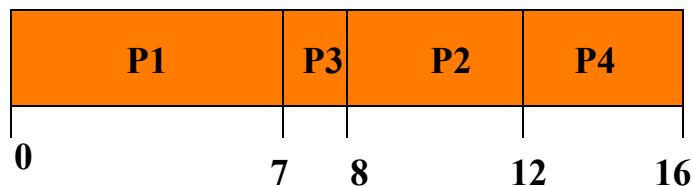
- | Associate with each process the length of its next CPU burst. Use these lengths to schedule the process with the shortest time.
- | Two Schemes:
 - | Scheme 1: Non-preemptive
 - Once CPU is given to the process it cannot be preempted until it completes its CPU burst.
 - | Scheme 2: Preemptive
 - If a new CPU process arrives with CPU burst length less than remaining time of current executing process, preempt. Also called Shortest-Remaining-Time-First (SRTF).
 - | SJF is optimal - gives minimum average waiting time for a given set of processes.

Non-Preemptive SJF Scheduling

Example

| Process | Arrival Time | Burst Time |
|---------|--------------|------------|
| P1 | 0 | 7 |
| P2 | 2 | 4 |
| P3 | 4 | 1 |
| P4 | 5 | 4 |

Gantt Chart for Schedule



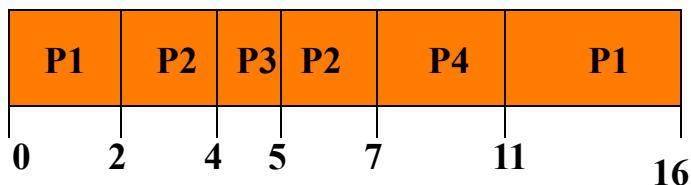
Average waiting time =
 $(0+6+3+7)/4 = 4$

Preemptive SJF Scheduling (SRTF)

Example

| Process | Arrival Time | Burst Time |
|---------|--------------|------------|
| P1 | 0 | 7 |
| P2 | 2 | 4 |
| P3 | 4 | 1 |
| P4 | 5 | 4 |

Gantt Chart for Schedule



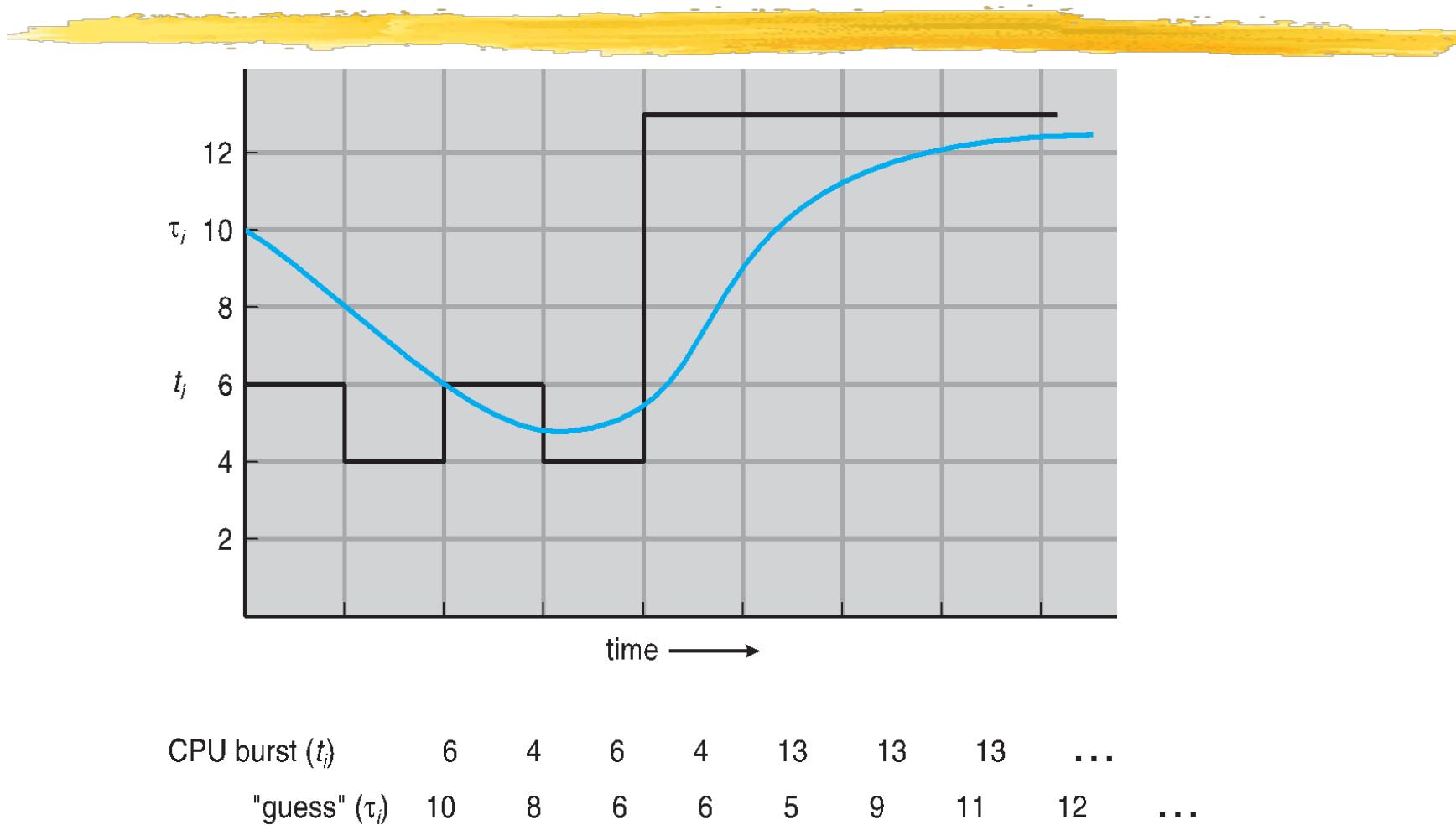
Average waiting time =
 $(9+1+0+2)/4 = 3$

Determining Length of Next CPU Burst



- One can only estimate the length of burst.
 - | Halting problem: cannot (in general) determine if a program will run forever on some input or complete in finite time
- Estimate next CPU burst by an *exponentially-weighted moving average* over previous ones:
 - | t_n = actual length of n^{th} burst
 - | τ_{n+1} = predicted value for the next CPU burst
 - | α = weight, $0 \leq \alpha \leq 1$
 - | Define
 - $\tau_{n+1} = \alpha t_n + (1 - \alpha) \tau_n$

Prediction of the length of the next CPU burst



Exponential Averaging(cont.)



- $\alpha = 0$
 - | $\tau_{n+1} = \tau_n$; Recent history does not count
- $\alpha = 1$
 - | $\tau_{n+1} = t_n$; Only the actual last CPU burst counts.
- Similarly, expanding the formula:
 - |
$$\begin{aligned}\tau_{n+1} &= \alpha t_n + (1-\alpha) \alpha t_{n-1} + \dots + \\ &\quad (1-\alpha)^j \alpha t_{n-j} + \dots \\ &\quad (1-\alpha)^{(n+1)} \tau_0\end{aligned}$$
 - Each successive term has less weight than its predecessor.

Priority Scheduling



- A priority value (integer) is associated with each process. Can be based on
 - Cost to user
 - Importance to user
 - Aging
 - %CPU time used in last X hours.
- CPU is allocated to process with the highest priority.
 - | Preemptive
 - | Non-preemptive

Priority Scheduling (cont.)



- SJF is a priority scheme where the priority is the predicted next CPU burst time.
 - | Higher priority number → lower priority
 - | E.g. priority 0 is higher priority than priority 1
- Problem
 - | Starvation!! - Low priority processes may never execute.
- Solution
 - | Aging - as time progresses increase the priority of the process.

Round Robin (RR)



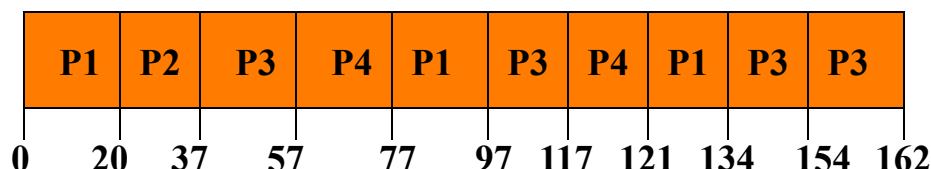
- Each process gets a small unit of CPU time
 - *Time quantum* usually 10-100 milliseconds.
 - After this time has elapsed, the process is preempted, moved to the end of the ready queue, and the process at the head of the ready queue is allocated the CPU.
- n processes, time quantum = q
 - Each process gets $1/n$ CPU time in chunks of at most q at a time.
 - No process waits more than $(n-1)q$ time units.
 - Performance considerations:
 - Time slice q too large – FIFO (FCFS) behavior
 - Time slice q too small - Overhead of context switch is too expensive.
 - Heuristic - 70-80% of jobs block within time slice

Round Robin Example

Time Quantum = 20

| Process | Burst Time |
|---------|------------|
| P1 | 53 |
| P2 | 17 |
| P3 | 68 |
| P4 | 24 |

Gantt Chart for Schedule



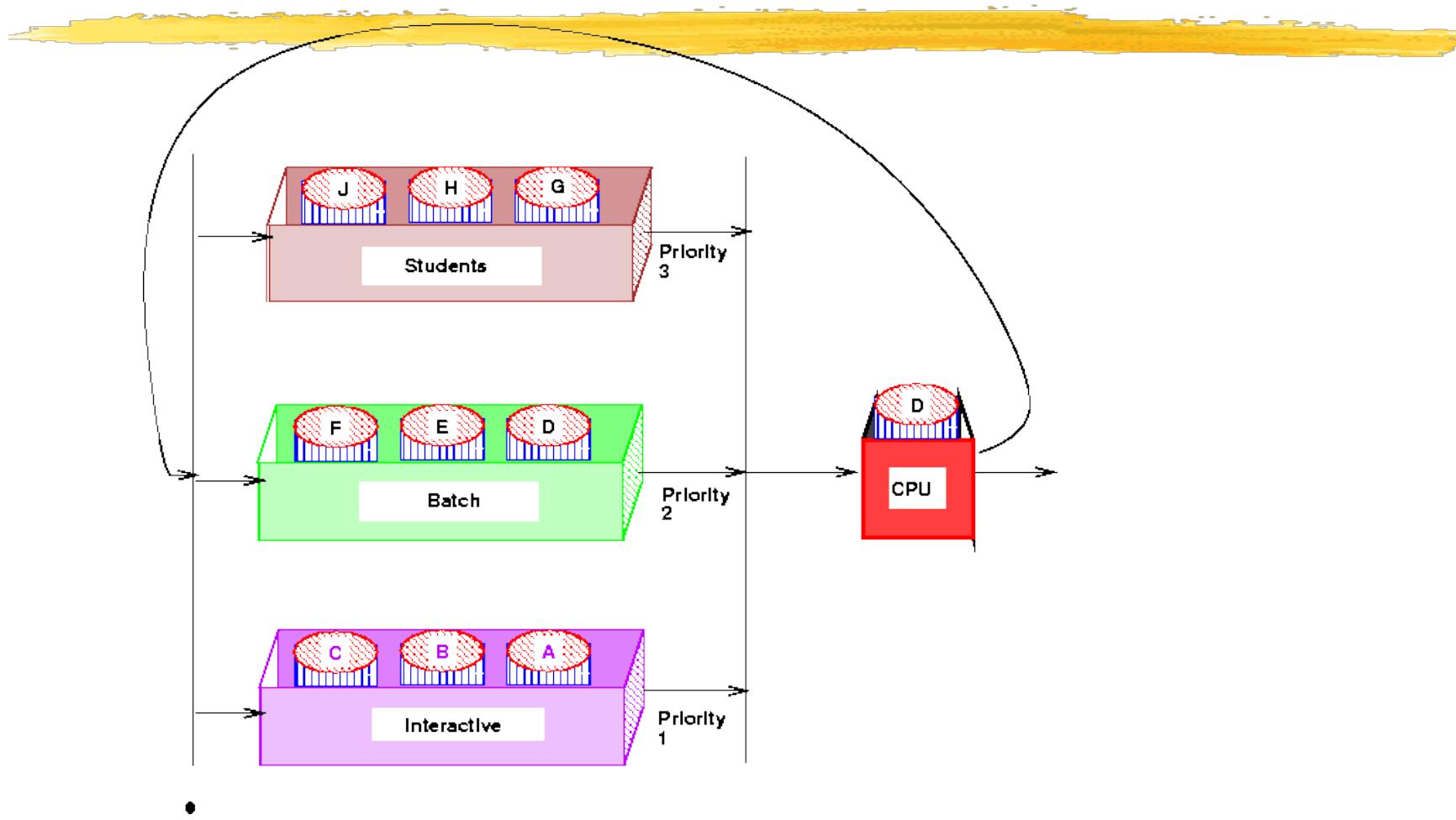
Typically, higher average turnaround time than SRTF, but better response time

Multilevel Queue



- Ready Queue partitioned into separate queues
 - Example: system processes, foreground (interactive), background (batch), student processes...
- Each queue has its own scheduling algorithm
 - Example: foreground (RR), background (FCFS)
- Processes assigned to one queue permanently.
- Scheduling must be done between the queues
 - Fixed priority - serve all from foreground, then from background. Possibility of starvation.
 - Time slice - Each queue gets some CPU time that it schedules - e.g. 80% foreground (RR), 20% background (FCFS)

Multilevel Queues



Multilevel Feedback Queue



- Multilevel Queue with priorities
- A process can *move* between the queues.
 - Aging can be implemented this way.
- Parameters for a multilevel feedback queue scheduler:
 - number of queues.
 - scheduling algorithm for each queue.
 - method used to determine when to upgrade a process.
 - method used to determine when to demote a process.
 - method used to determine which queue a process will enter when that process needs service.

Multilevel Feedback Queues



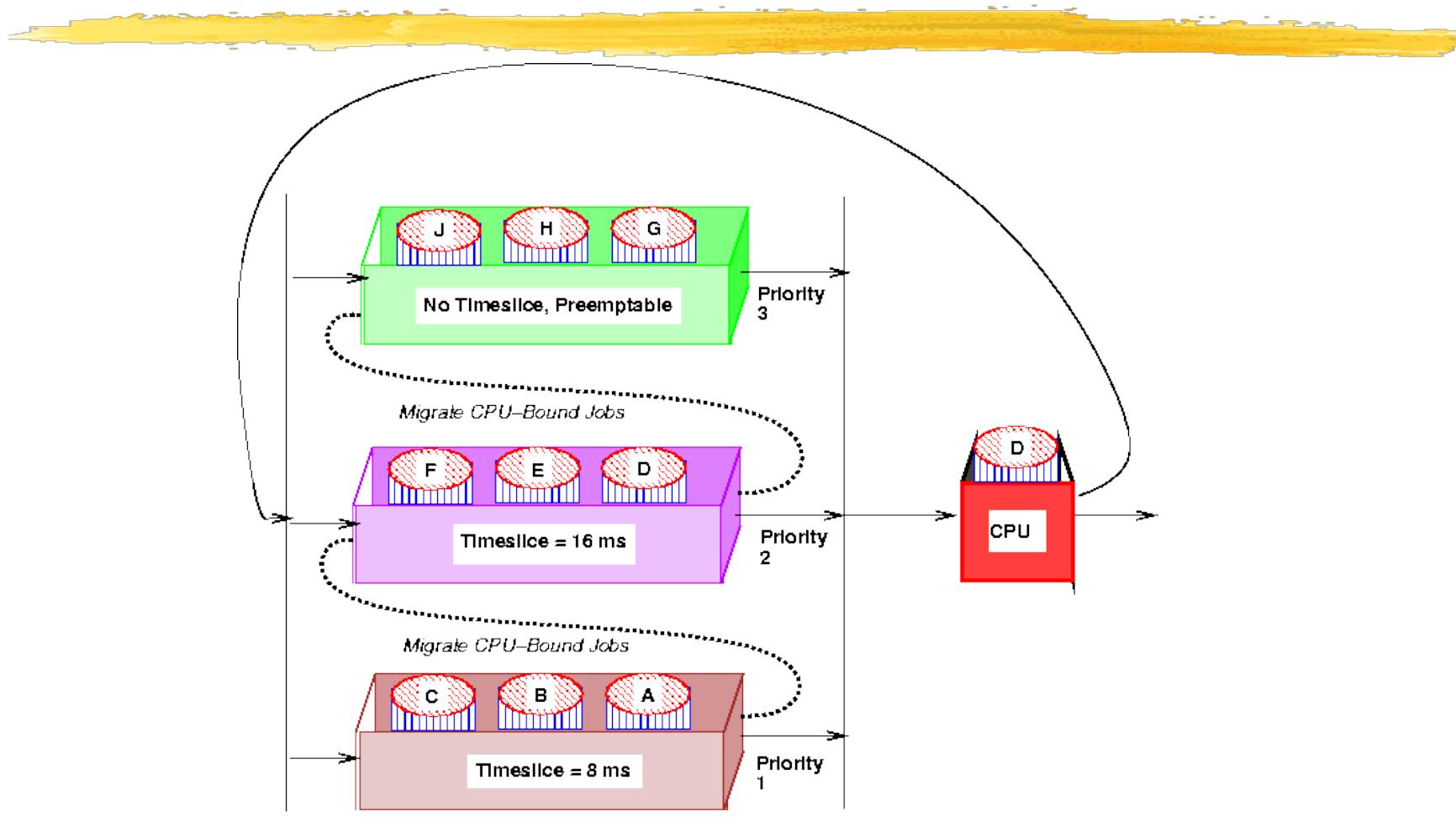
Example: Three Queues -

- Q0 - time quantum 8 milliseconds (FCFS)
- Q1 - time quantum 16 milliseconds (FCFS)
- Q2 - FCFS

Scheduling

- New job enters Q0 - When it gains CPU, it receives 8 milliseconds. If job does not finish, move it to Q1.
- At Q1, when job gains CPU, it receives 16 more milliseconds. If job does not complete, it is preempted and moved to Q2.

Multilevel Feedback Queues



Multiple-Processor Scheduling



- CPU scheduling becomes more complex when multiple CPUs are available.
 - | Have one ready queue accessed by each CPU.
 - Self scheduled - each CPU dispatches a job from ready Q
 - Manager-worker - one CPU schedules the other CPUs
- Homogeneous processors within multiprocessor.
 - Permits Load Sharing
- Asymmetric multiprocessing
 - only 1 CPU runs kernel, others run user programs
 - alleviates need for data sharing

Real-Time Scheduling



■ Hard Real-time Computing -

- required to complete a critical task within a guaranteed amount of time.

■ Soft Real-time Computing -

- requires that critical processes receive priority over less important ones.

■ Types of real-time Schedulers

- Periodic Schedulers - Fixed Arrival Rate
- Demand-Driven Schedulers - Variable Arrival Rate
- Deadline Schedulers - Priority determined by deadline
- ...

Issues in Real-time Scheduling



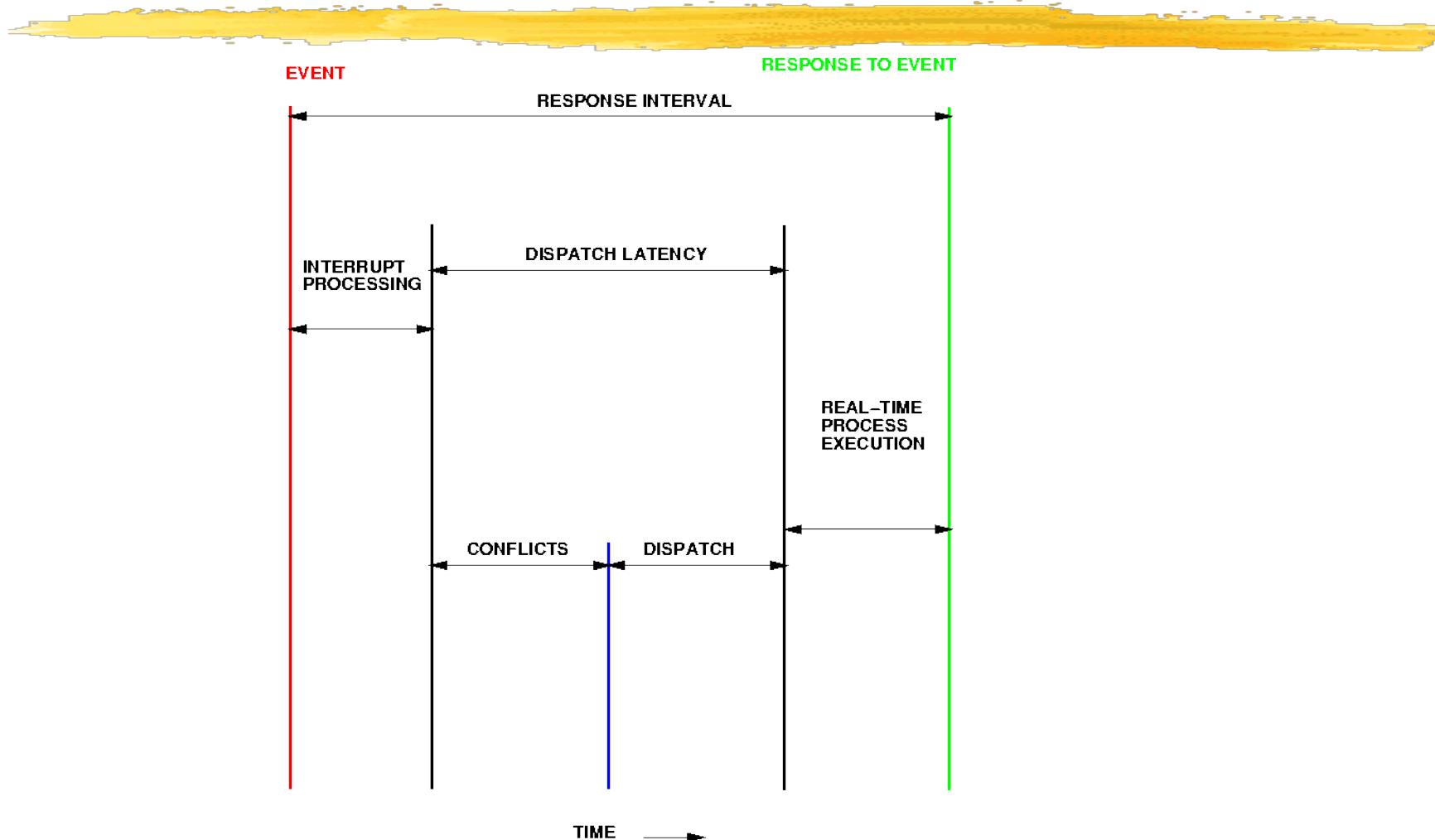
■ Dispatch Latency

- Problem - Need to keep dispatch latency small, OS may enforce process to wait for system call or I/O to complete.
- Solution - Make system calls preemptible, determine safe criteria such that kernel can be interrupted.

■ Priority Inversion and Inheritance

- Problem: Priority Inversion
 - Higher Priority Process P0 needs kernel resource currently being used by another lower priority process P1.
 - P0 must wait for P1, but P1 isn't getting scheduled in CPU!
- Solution: Priority Inheritance
 - Low priority process now inherits high priority until it has completed use of the resource in question.

Real-time Scheduling - Dispatch Latency



Scheduling Algorithm Evaluation



Deterministic Modeling

- Takes a particular predetermined workload and defines the performance of each algorithm for that workload.
- Too specific, requires exact knowledge to be useful.

Queuing Models and Queuing Theory

- Use distributions of CPU and I/O bursts. Knowing *arrival* and *service* rates - can compute utilization, average queue length, average wait time, etc...
- Little's formula: $n = \lambda \times W$
 - n is the average queue length, λ is the avg. arrival rate and W is the avg. waiting time in queue.

Other techniques: Simulations, Implementation

ICS 143 - Principles of Operating Systems

Lectures Set 5- Deadlocks

Prof. Nalini Venkatasubramanian

nalini@ics.uci.edu

Outline

- System Model
- Deadlock Characterization
- Methods for handling deadlocks
- Deadlock Prevention
- Deadlock Avoidance
- Deadlock Detection
- Recovery from Deadlock
- Combined Approach to Deadlock Handling

The Deadlock Problem

- A set of blocked processes each holding a resource and waiting to acquire a resource held by another process in the set.

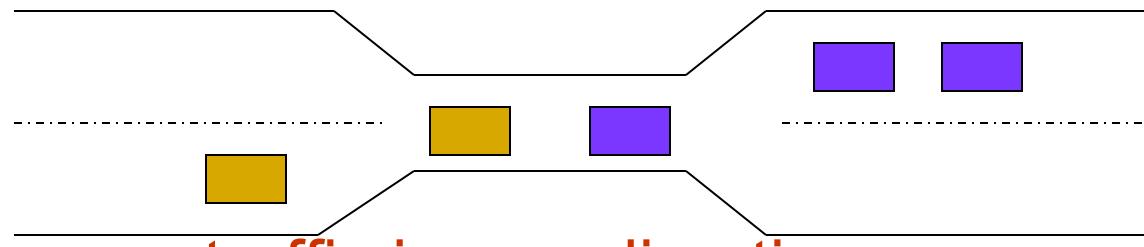
- Example 1
 - System has 2 tape drives. P1 and P2 each hold one tape drive and each needs the other one.
- Example 2
 - Semaphores A and B each initialized to 1

| P_0 | P_1 |
|-----------|-----------|
| $wait(A)$ | $wait(B)$ |
| $wait(B)$ | $wait(A)$ |

Definitions

- A process is *deadlocked* if it is waiting for an event that will never occur.
Typically, more than one process will be involved in a deadlock (the deadly embrace).
- A process is *indefinitely postponed* if it is delayed repeatedly over a long period of time while the attention of the system is given to other processes,
 - i.e. the process is ready to proceed but never gets the CPU.

Example - Bridge Crossing



- ❑ Assume traffic in one direction.
 - Each section of the bridge is viewed as a resource.
- ❑ If a deadlock occurs, it can be resolved only if one car backs up (preempt resources and rollback).
 - Several cars may have to be backed up if a deadlock occurs.
 - Starvation is possible

Resources

- **Resource**
 - commodity required by a process to execute
- **Resources can be of several types**
 - **Serially Reusable Resources**
 - CPU cycles, memory space, I/O devices, files
 - acquire -> use -> release
 - **Consumable Resources**
 - Produced by a process, needed by a process - e.g. Messages, buffers of information, interrupts
 - create ->acquire ->use
 - Resource ceases to exist after it has been used

System Model

- Resource types
 - R_1, R_2, \dots, R_m
- Each resource type R_i has W_i instances
- Assume serially reusable resources
 - request -> use -> release

Conditions for Deadlock

- ❑ The following 4 conditions are necessary and sufficient for deadlock (must hold simultaneously)
 - Mutual Exclusion:
 - ❑ Only one process at a time can use the resource.
 - Hold and Wait:
 - ❑ Processes hold resources already allocated to them while waiting for other resources.
 - No preemption:
 - ❑ Resources are released by processes holding them only after that process has completed its task.
 - Circular wait:
 - ❑ A circular chain of processes exists in which each process waits for one or more resources held by the next process in the chain.

Resource Allocation Graph

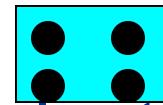
- A set of vertices V and a set of edges E
- V is partitioned into 2 types
 - $P = \{P_1, P_2, \dots, P_n\}$ - the set of processes in the system
 - $R = \{R_1, R_2, \dots, R_n\}$ - the set of resource types in the system
- Two kinds of edges
 - Request edge - Directed edge $P_i \rightarrow R_j$
 - Assignment edge - Directed edge $R_j \rightarrow P_i$

Resource Allocation Graph

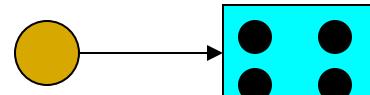
- Process



- Resource type with 4 instances



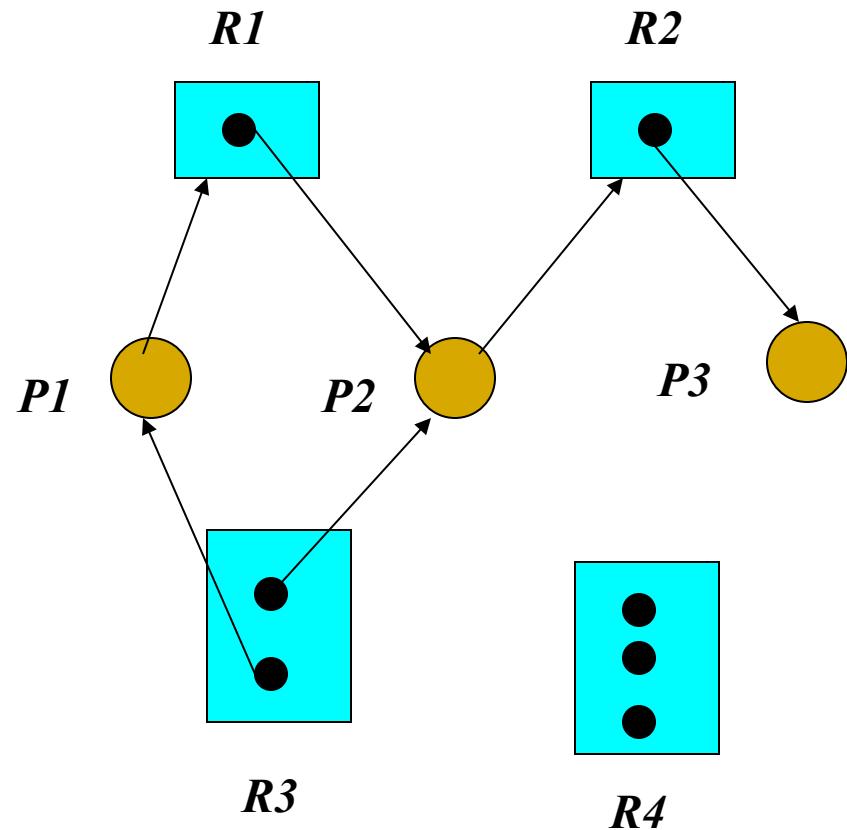
- P_i requests instance of R_j



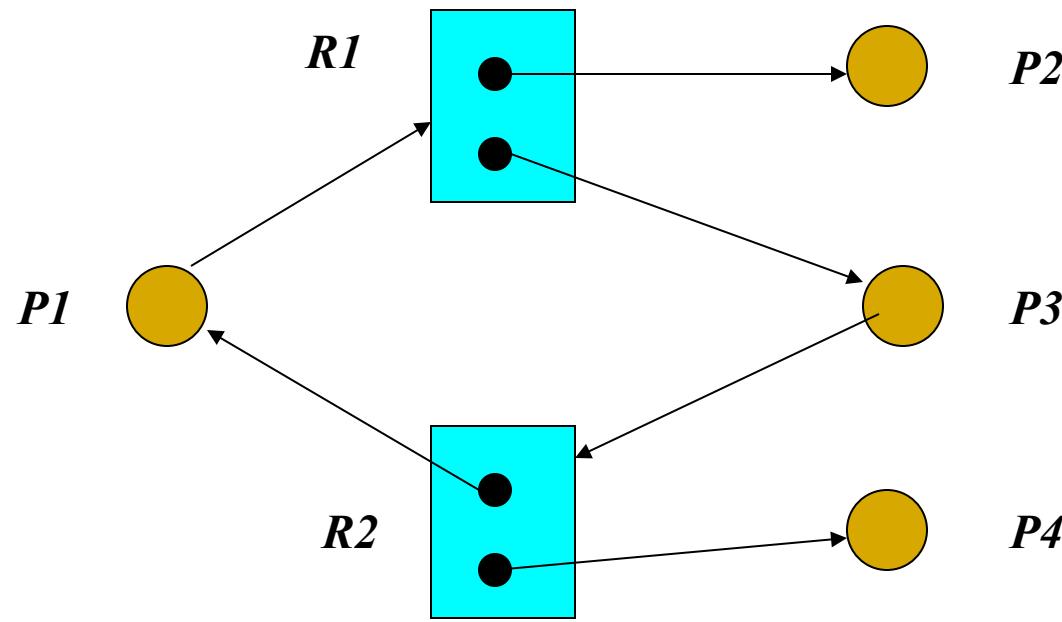
- P_i is holding an instance of R_j



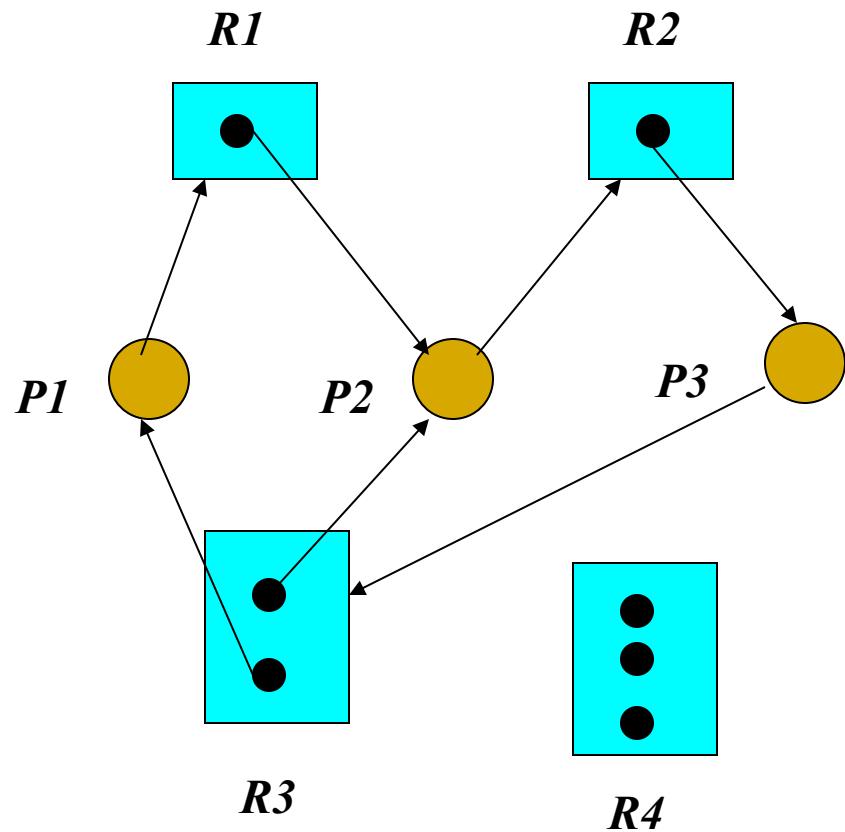
Graph with no cycles



Graph with cycles



Graph with cycles and deadlock



Basic facts

- If graph contains no cycles
 - NO DEADLOCK
- If graph contains a cycle
 - if only one instance per resource type, then deadlock
 - if several instances per resource type, possibility of deadlock.

Resource



Process



Resource Type

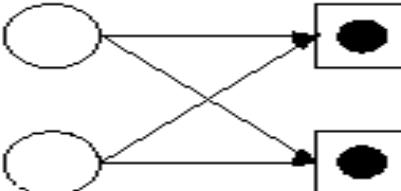


2 Processes

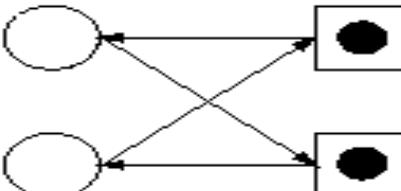


2 resources

Processes
request
2 resources
each

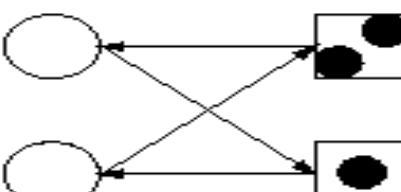


Deadlock



Cycle in resource
graph

Deadlock
may not
occur if
there are
enough
resources



Cycle in resource
graph

Methods for handling deadlocks

- Ensure that the system will never enter a deadlock state.
- Allow the system to potentially enter a deadlock state, detect it and then recover
- Ignore the problem and pretend that deadlocks never occur in the system;
 - Used by many operating systems, e.g. UNIX

Deadlock Management

- ❑ Prevention
 - ❑ Design the system in such a way that deadlocks can never occur
- ❑ Avoidance
 - ❑ Impose less stringent conditions than for prevention, allowing the possibility of deadlock but sidestepping it as it occurs.
- ❑ Detection
 - ❑ Allow possibility of deadlock, determine if deadlock has occurred and which processes and resources are involved.
- ❑ Recovery
 - ❑ After detection, clear the problem, allow processes to complete and resources to be reused. May involve destroying and restarting processes.

Deadlock Prevention

- ❑ If any one of the conditions for deadlock (with reusable resources) is denied, deadlock is impossible.
- ❑ Restrain ways in which requests can be made
 - Mutual Exclusion
 - ❑ non-issue for sharable resources
 - ❑ cannot deny this for non-sharable resources (important)
 - Hold and Wait - guarantee that when a process requests a resource, it does not hold other resources.
 - ❑ Force each process to acquire all the required resources at once. Process cannot proceed until all resources have been acquired.
 - ❑ Low resource utilization, starvation possible

Deadlock Prevention (cont.)

■ No Preemption

- If a process that is holding some resources requests another resource that cannot be immediately allocated to it, the process releases the resources currently being held.
- Preempted resources are added to the list of resources for which the process is waiting.
- Process will be restarted only when it can regain its old resources as well as the new ones that it is requesting.

■ Circular Wait

- Impose a total ordering of all resource types.
- Require that processes request resources in increasing order of enumeration; if a resource of type N is held, process can only request resources of types > N.

Deadlock Avoidance

- Set of resources, set of customers, banker
- Rules
 - Each customer tells banker maximum number of resources it needs.
 - Customer borrows resources from banker.
 - Customer returns resources to banker.
 - Customer eventually pays back loan.
- Banker only lends resources if the system will be in a *safe state* after the loan.

Deadlock Avoidance

- Requires that the system has some additional apriori information available.
 - Simplest and most useful model requires that each process declare the maximum number of resources of each type that it may need.
 - The deadlock-avoidance algorithm dynamically examines the resource-allocation state to ensure that there can never be a circular-wait condition.
 - Resource allocation state is defined by the number of available and allocated resources, and the maximum demands of the processes.

Safe state

- When a process requests an available resource, system must decide if immediate allocation leaves the system in a safe state.
- System is in safe state if there exists a safe sequence of all processes.
- Sequence $\langle P_1, P_2, \dots, P_n \rangle$ is safe, if for each P_i , the resources that P_i can still request can be satisfied by currently available resources + resources held by P_j with $j < i$.
 - If P_i resource needs are not available, P_i can wait until all P_j have finished.
 - When P_j is finished, P_i can obtain needed resources, execute, return allocated resources, and terminate.
 - When P_i terminates, P_{i+1} can obtain its needed resources...

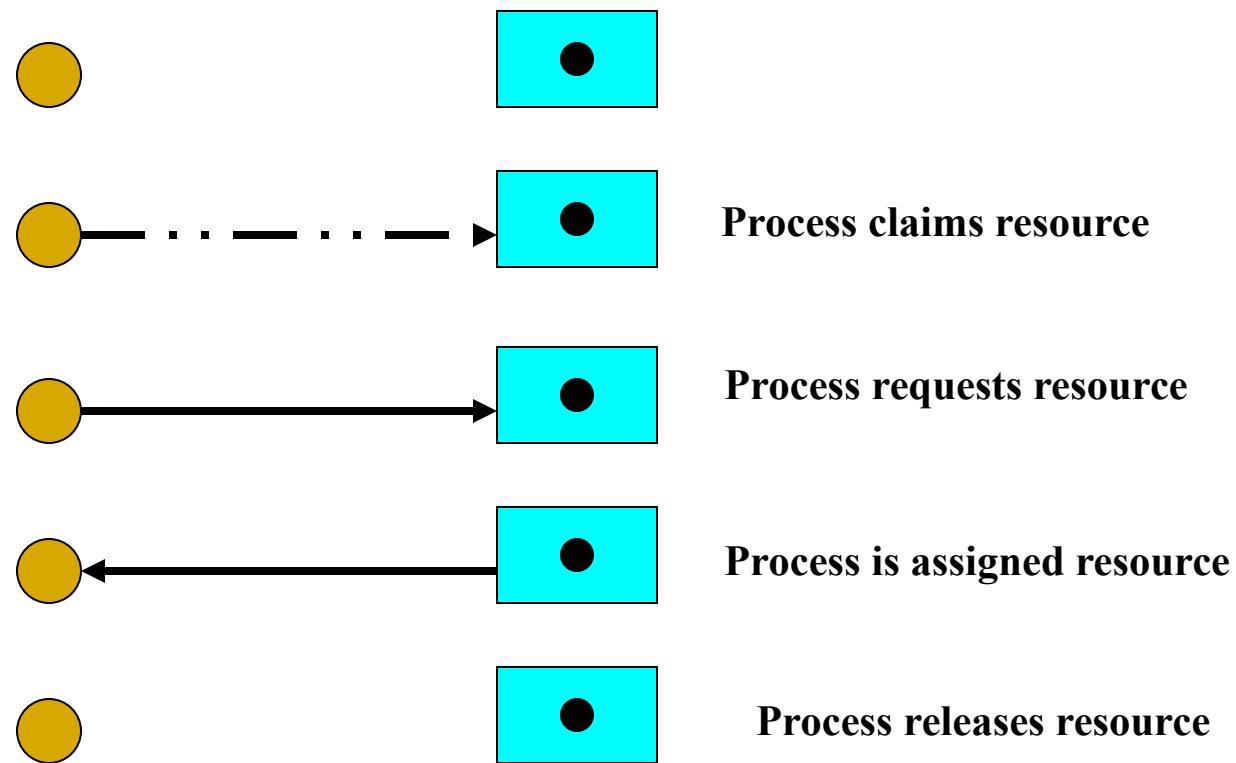
Basic Facts

- If a system is in a safe state \Rightarrow no deadlocks.
- If a system is in unsafe state \Rightarrow possibility of deadlock.
- Avoidance \Rightarrow ensure that a system will never reach an unsafe state.

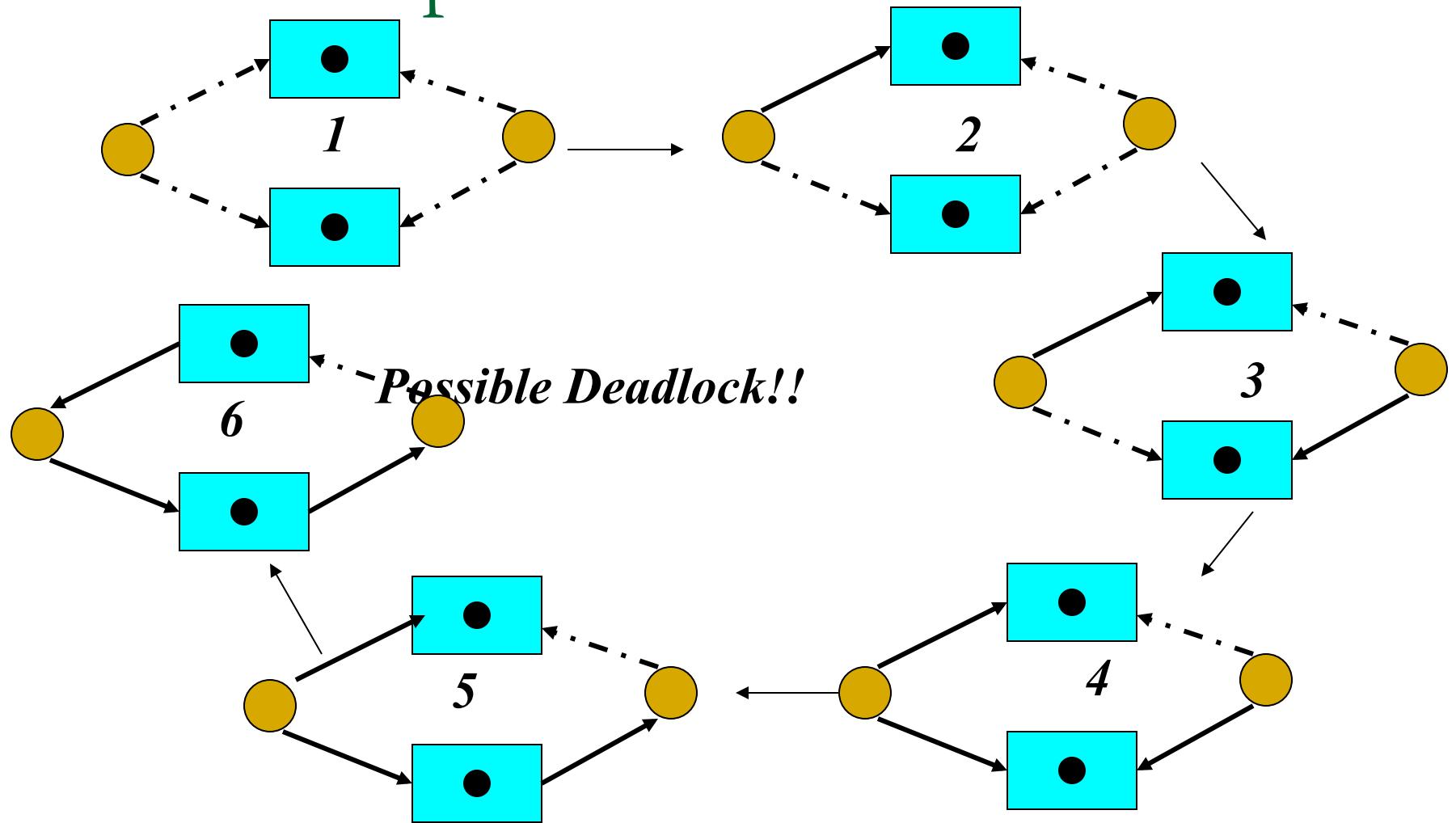
Resource Allocation Graph Algorithm

- Used for deadlock avoidance when there is only one instance of each resource type.
 - Claim edge: $P_i \rightarrow R_j$ indicates that process P_i may request resource R_j ; represented by a dashed line.
 - Claim edge converts to request edge when a process requests a resource.
 - When a resource is released by a process, assignment edge reconverts to claim edge.
 - Resources must be claimed a priori in the system.
- If request assignment does not result in the formation of a cycle in the resource allocation graph - safe state, else unsafe state.

Claim Graph



Claim Graph



Banker's Algorithm

- Used for multiple instances of each resource type.
- Each process must a priori claim maximum use of each resource type.
- When a process requests a resource it may have to wait.
- When a process gets all its resources it must return them in a finite amount of time.

Data Structures for the Banker's Algorithm

- Let n = number of processes and m = number of resource types.
 - *Available*: Vector of length m . If $\text{Available}[j] = k$, there are k instances of resource type R_j available.
 - *Max*: $n \times m$ matrix. If $\text{Max}[i,j] = k$, then process P_i may request at most k instances of resource type R_j .
 - *Allocation*: $n \times m$ matrix. If $\text{Allocation}[i,j] = k$, then process P_i is currently allocated k instances of resource type R_j .
 - *Need*: $n \times m$ matrix. If $\text{Need}[i,j] = k$, then process P_i may need k more instances of resource type R_j to complete its task.

$$\text{Need}[i,j] = \text{Max}[i,j] - \text{Allocation}[i,j]$$

Safety Algorithm

- Let $Work$ and $Finish$ be vectors of length m and n , respectively. Initialize
 - $Work := Available$
 - $Finish[i] := false$ for $i = 1, 2, \dots, n$.
- Find an i (i.e. process P_i) such that both:
 - $Finish[i] = false$
 - $Need_i \leq Work$
 - If no such i exists, go to step 4.
- $Work := Work + Allocation_i$
 - $Finish[i] := true$
 - go to step 2
- If $Finish[i] = true$ for all i , then the system is in a safe state.

Resource-Request Algorithm for Process P_i

- Request_i = request vector for process P_i . If $\text{Request}_i[j] = k$, then process P_i wants k instances of resource type R_j .
 - STEP 1: If $\text{Request}(i) \leq \text{Need}(i)$, go to step 2. Otherwise, raise error condition, since process has exceeded its maximum claim.
 - STEP 2: If $\text{Request}(i) \leq \text{Available}$, go to step 3. Otherwise, P_i must wait since resources are not available.
 - STEP 3: Pretend to allocate requested resources to P_i by modifying the state as follows:
$$\begin{aligned}\text{Available} &:= \text{Available} - \text{Request}(i); \\ \text{Allocation}(i) &:= \text{Allocation}(i) + \text{Request}(i); \\ \text{Need}(i) &:= \text{Need}(i) - \text{Request}(i);\end{aligned}$$
 - If safe \Rightarrow resources are allocated to P_i .
 - If unsafe \Rightarrow P_i must wait and the old resource-allocation state is restored.

Example of Banker's Algorithm

- 5 processes
 - P0 - P4;
- 3 resource types
 - A(10 instances), B (5 instances), C (7 instances)
- Snapshot at time T0

| | Allocation | | | Max | | | Available | | |
|----|------------|---|---|-----|---|---|-----------|---|---|
| | A | B | C | A | B | C | A | B | C |
| P0 | 0 | 1 | 0 | 7 | 5 | 3 | 3 | 3 | 2 |
| P1 | 2 | 0 | 0 | 3 | 2 | 2 | | | |
| P2 | 3 | 0 | 2 | 9 | 0 | 2 | | | |
| P3 | 2 | 1 | 1 | 2 | 2 | 2 | | | |
| P4 | 0 | 0 | 2 | 4 | 3 | 3 | | | |

Example (cont.)

- The content of the matrix *Need* is defined to be *Max - Allocation*.
- The system is in a safe state since the sequence $\langle P_1, P_3, P_4, P_2, P_0 \rangle$ satisfies safety criteria.

| | Need | | |
|----|------|---|---|
| | A | B | C |
| P0 | 7 | 4 | 3 |
| P1 | 1 | 2 | 2 |
| P2 | 6 | 0 | 0 |
| P3 | 0 | 1 | 1 |
| P4 | 4 | 3 | 1 |

Example: P1 requests (1,0,2)

- Check to see that Request \leq Available
 - $((1,0,2) \leq (3,3,2)) \Rightarrow \text{true.}$

| | Allocation | | | Need | | | Available | | |
|----|------------|---|---|------|---|---|-----------|---|---|
| | A | B | C | A | B | C | A | B | C |
| P0 | 0 | 1 | 0 | 7 | 4 | 3 | 2 | 3 | 0 |
| P1 | 3 | 0 | 2 | 0 | 2 | 0 | | | |
| P2 | 3 | 0 | 2 | 6 | 0 | 0 | | | |
| P3 | 2 | 1 | 1 | 0 | 1 | 1 | | | |
| P4 | 0 | 0 | 2 | 4 | 3 | 1 | | | |

Example (cont.)

- Executing the safety algorithm shows that sequence $\langle P_1, P_3, P_4, P_0, P_2 \rangle$ satisfies safety requirement.
- Can request for $(3,3,0)$ by P_4 be granted?
- Can request for $(0,2,0)$ by P_0 be granted?

Deadlock Detection

- Allow system to enter deadlock state
- Detection Algorithm
- Recovery Scheme

Single Instance of each resource type

- Maintain wait-for graph
 - Nodes are processes
 - $P_i \rightarrow P_j$ if P_i is waiting for P_j .
- Periodically invoke an algorithm that searches for a cycle in the graph.
- An algorithm to detect a cycle in a graph requires an order of n^2 operations, where n is the number of vertices in the graph.

Several instances of a resource type

■ Data Structures

- *Available*: Vector of length m . If $\text{Available}[j] = k$, there are k instances of resource type R_j available.
- *Allocation*: $n \times m$ matrix. If $\text{Allocation}[i,j] = k$, then process P_i is currently allocated k instances of resource type R_j .
- *Request* : An $n \times m$ matrix indicates the current request of each process. If $\text{Request}[i,j] = k$, then process P_i is requesting k more instances of resource type R_j .

Deadlock Detection Algorithm

- Step 1: Let $Work$ and $Finish$ be vectors of length m and n , respectively. Initialize
 - $Work := Available$
 - For $i = 1, 2, \dots, n$, if $Allocation(i) \neq 0$, then $Finish[i] := false$, otherwise $Finish[i] := true$.
- Step 2: Find an index i such that both:
 - $Finish[i] = false$
 - $Request(i) \leq Work$
 - If no such i exists, go to step 4.

Deadlock Detection Algorithm

- Step 3: $Work := Work + Allocation(i)$
 - $Finish[i] := true$
 - go to step 2
- Step 4: If $Finish[i] = false$ for some i , $1 \leq i \leq n$, then the system is in a deadlock state. Moreover, if $Finish[i] = false$, then P_i is deadlocked.

Algorithm requires an order of $m \times (n^2)$ operations to detect whether the system is in a deadlocked state.

Example of Detection Algorithm

- 5 processes - P_0 - P_4 ; 3 resource types - $A(7)$ instances), $B(2$ instances), $C(6$ instances)
- Snapshot at time T_0 : $\langle P_0, P_2, P_3, P_1, P_4 \rangle$ will result in $Finish[i] = \text{true}$ for all i .

| | Allocation | | | Request | | | Available | | |
|----|------------|---|---|---------|---|---|-----------|---|---|
| | A | B | C | A | B | C | A | B | C |
| P0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| P1 | 2 | 0 | 0 | 2 | 0 | 2 | | | |
| P2 | 3 | 0 | 3 | 0 | 0 | 0 | | | |
| P3 | 2 | 1 | 1 | 1 | 0 | 0 | | | |
| P4 | 0 | 0 | 2 | 0 | 0 | 2 | | | |

Example (cont.)

- P2 requests an additional instance of type C.
- State of system
 - Can reclaim resources held by process P0, but insufficient resources to fulfill other processes' requests.
 - Deadlock exists, consisting of P_1, P_2, P_3 and P_4 .

| | Request | | |
|----|---------|---|---|
| | A | B | C |
| P0 | 0 | 0 | 0 |
| P1 | 2 | 0 | 2 |
| P2 | 0 | 0 | 1 |
| P3 | 1 | 0 | 0 |
| P4 | 0 | 0 | 2 |

Detection-Algorithm Use

- ❑ When, and how often to invoke depends on:
 - How often a deadlock is likely to occur?
 - How many processes will need to be rolled back?
 - ❑ One for each disjoint cycle
- ❑ How often --
 - Every time a request for allocation cannot be granted immediately
 - ❑ Allows us to detect set of deadlocked processes and process that “caused” deadlock. Extra overhead.
 - ❑ Every hour or whenever CPU utilization drops.
 - With arbitrary invocation there may be many cycles in the resource graph and we would not be able to tell which of the many deadlocked processes “caused” the deadlock.

Recovery from Deadlock: Process Termination

- Abort all deadlocked processes.
- Abort one process at a time until the deadlock cycle is eliminated.
- In which order should we choose to abort?
 - Priority of the process
 - How long the process has computed, and how much longer to completion.
 - Resources the process has used.
 - Resources process needs to complete.
 - How many processes will need to be terminated.
 - Is process interactive or batch?

Recovery from Deadlock: Resource Preemption

- Selecting a victim - minimize cost.
- Rollback
 - return to some safe state, restart process from that state.
- Starvation
 - same process may always be picked as victim; include number of rollback in cost factor.

Combined approach to deadlock handling

- Combine the three basic approaches
 - Prevention
 - Avoidance
 - Detection

allowing the use of the optimal approach for each class of resources in the system.
- Partition resources into hierarchically ordered classes.
 - Use most appropriate technique for handling deadlocks within each class.

ICS 143 - Principles of Operating Systems

Lectures 13-14 - Memory Management: Main Memory

Prof. Ardalan Amiri Sani

Prof. Nalini Venkatasubramanian

ardalan@uci.edu

nalini@ics.uci.edu

Outline

- Background
- Logical versus Physical Address Space
- Swapping
- Contiguous Allocation
- Paging
- Segmentation
- Segmentation with Paging

Background

- Program must be brought into memory and placed within a process for it to be executed.
- Input Queue - collection of processes on the disk that are waiting to be brought into memory for execution.
- User programs go through several steps before being executed.

Virtualizing Resources

- Physical Reality: Processes/Threads share the same hardware
 - Need to multiplex CPU (CPU Scheduling)
 - Need to multiplex use of Memory (Today)
- Why worry about memory multiplexing?
 - The complete working state of a process and/or kernel is defined by its data in memory (and registers)
 - Consequently, cannot just let different processes use the same memory
 - Probably don't want different processes to even have access to each other's memory (protection)

Important Aspects of Memory Multiplexing

- Controlled overlap:
 - Processes should not collide in physical memory
 - Conversely, would like the ability to share memory when desired (for communication)
- Protection:
 - Prevent access to private memory of other processes
 - Different pages of memory can be given special behavior (Read Only, Invisible to user programs, etc)
 - Kernel data protected from user programs
- Translation:
 - Ability to translate accesses from one address space (virtual) to a different one (physical)
 - When translation exists, process uses virtual addresses, physical memory uses physical addresses

Names and Binding

- Symbolic names → Logical names → Physical names
 - Symbolic Names: known in a context or path
 - file names, program names, printer/device names, user names
 - Logical Names: used to label a specific entity
 - inodes, job number, major/minor device numbers, process id (pid), uid, gid..
 - Physical Names: address of entity
 - inode address on disk or memory
 - entry point or variable address
 - PCB address

Binding of instructions and data to memory

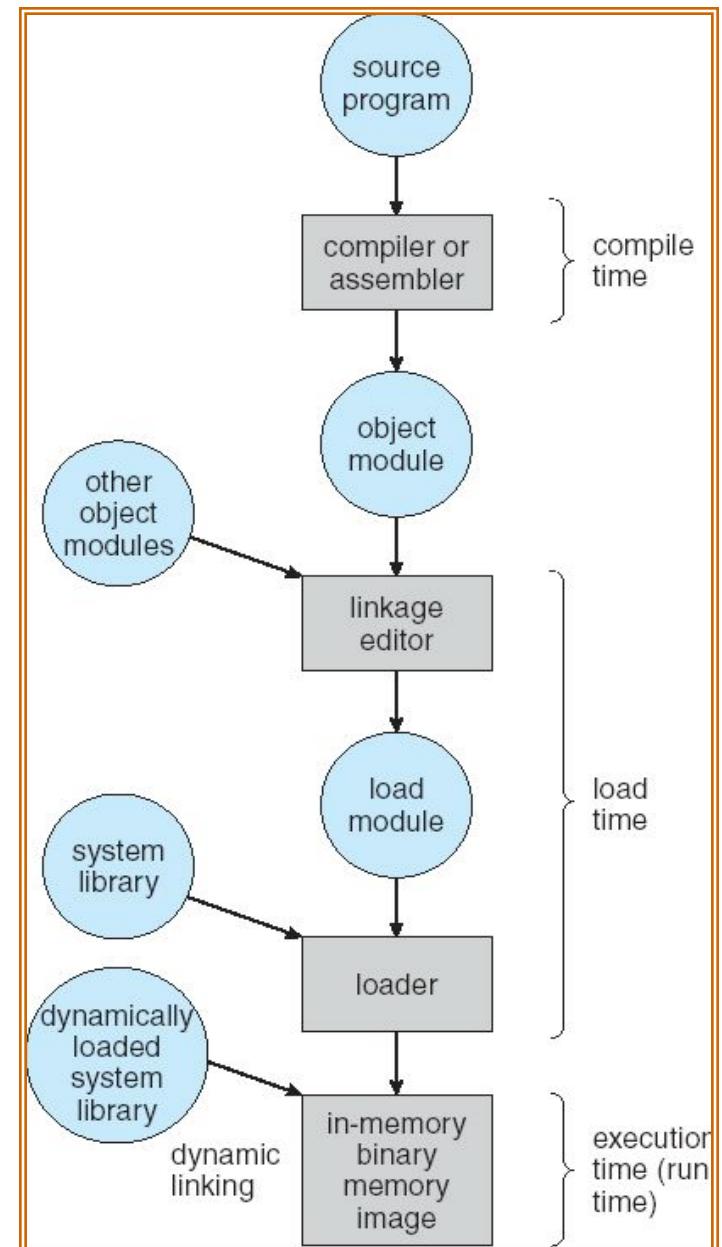
- ❑ Address binding of instructions and data to memory addresses can happen at three different stages.
 - Compile time:
 - ❑ If memory location is known a priori, absolute code can be generated; must recompile code if starting location changes.
 - Load time:
 - ❑ Must generate relocatable code if memory location is not known at compile time.
 - Execution time:
 - ❑ Binding delayed until runtime if the process can be moved during its execution from one memory segment to another.
Need hardware support for address maps (e.g. base and limit registers).

Binding time tradeoffs

- ❑ Early binding
 - ❑ compiler - produces efficient code
 - ❑ allows checking to be done early
 - ❑ allows estimates of running time and space
- ❑ Delayed binding
 - ❑ Linker, loader
 - ❑ produces efficient code, allows separate compilation
 - ❑ portability and sharing of object code
- ❑ Late binding
 - ❑ VM, dynamic linking/loading, overlaying, interpreting
 - ❑ code less efficient, checks done at runtime
 - ❑ flexible, allows dynamic reconfiguration

Multi-step Processing of a Program for Execution

- Preparation of a program for execution involves components at:
 - Compile time (i.e., “gcc”)
 - Link/Load time (unix “ld” does link)
 - Execution time (e.g. dynamic libs)
- Addresses can be bound to final values anywhere in this path
 - Depends on hardware support
 - Also depends on operating system
- Dynamic Libraries
 - Linking postponed until execution
 - Small piece of code, *stub*, used to locate appropriate memory-resident library routine
 - Stub replaces itself with the address of the routine, and executes routine



Dynamic Loading

- Routine is not loaded until it is called.
- Better memory-space utilization; unused routine is never loaded.
- Useful when large amounts of code are needed to handle infrequently occurring cases.
- No special support from the operating system is required; implemented through program design.

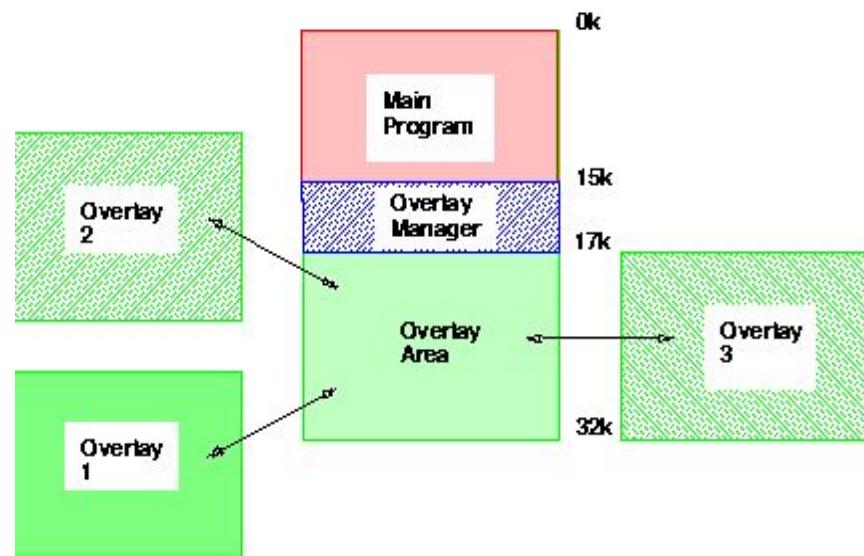
Dynamic Linking

- Linking postponed until execution time.
- Small piece of code, *stub*, used to locate the appropriate memory-resident library routine.
- Stub replaces itself with the address of the routine, and executes the routine.
- Operating system needed to check if routine is in processes' memory address.

Overlays

- Keep in memory only those instructions and data that are needed at any given time.
- Needed when process is larger than amount of memory allocated to it.
- Implemented by user, no special support from operating system; programming design of overlay structure is complex.

Overlaying



Logical vs. Physical Address Space

- The concept of a logical address space that is bound to a separate physical address space is central to proper memory management.
 - Logical Address: or virtual address - generated by CPU
 - Physical Address: address seen by memory unit.
- Logical and physical addresses are the same in compile time and load-time binding schemes
- Logical and physical addresses differ in execution-time address-binding scheme.

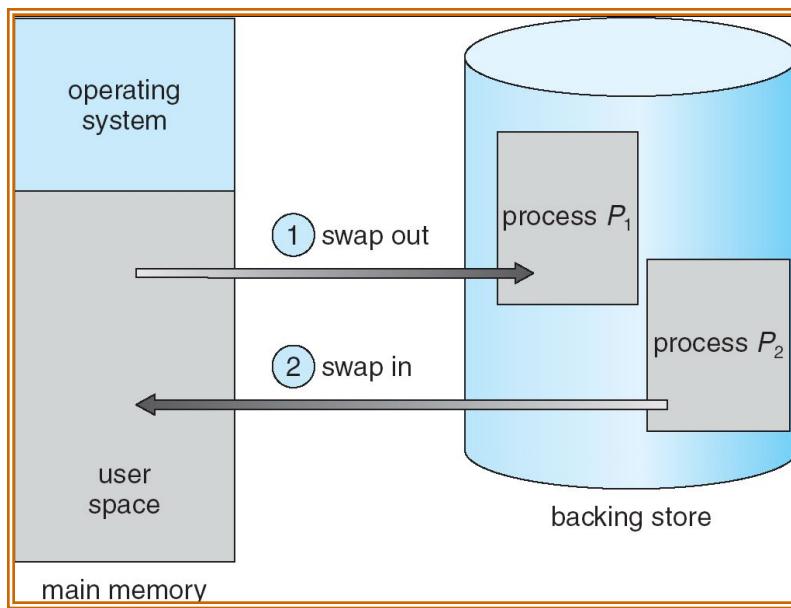
Memory Management Unit (MMU)

- Hardware device that maps virtual to physical address.
- In MMU scheme, the value in the relocation register is added to every address generated by a user process at the time it is sent to memory.
- The user program deals with logical addresses; it never sees the real physical address.

Swapping

- ❑ A process can be swapped temporarily out of memory to a backing store and then brought back into memory for continued execution.
 - ❑ Backing Store - fast disk large enough to accommodate copies of all memory images for all users; must provide direct access to these memory images.
 - ❑ Roll out, roll in - swapping variant used for priority based scheduling algorithms; lower priority process is swapped out, so higher priority process can be loaded and executed.
 - ❑ Major part of swap time is transfer time; total transfer time is directly proportional to the amount of memory swapped.
 - ❑ Modified versions of swapping are found on many systems, i.e. UNIX and Microsoft Windows.

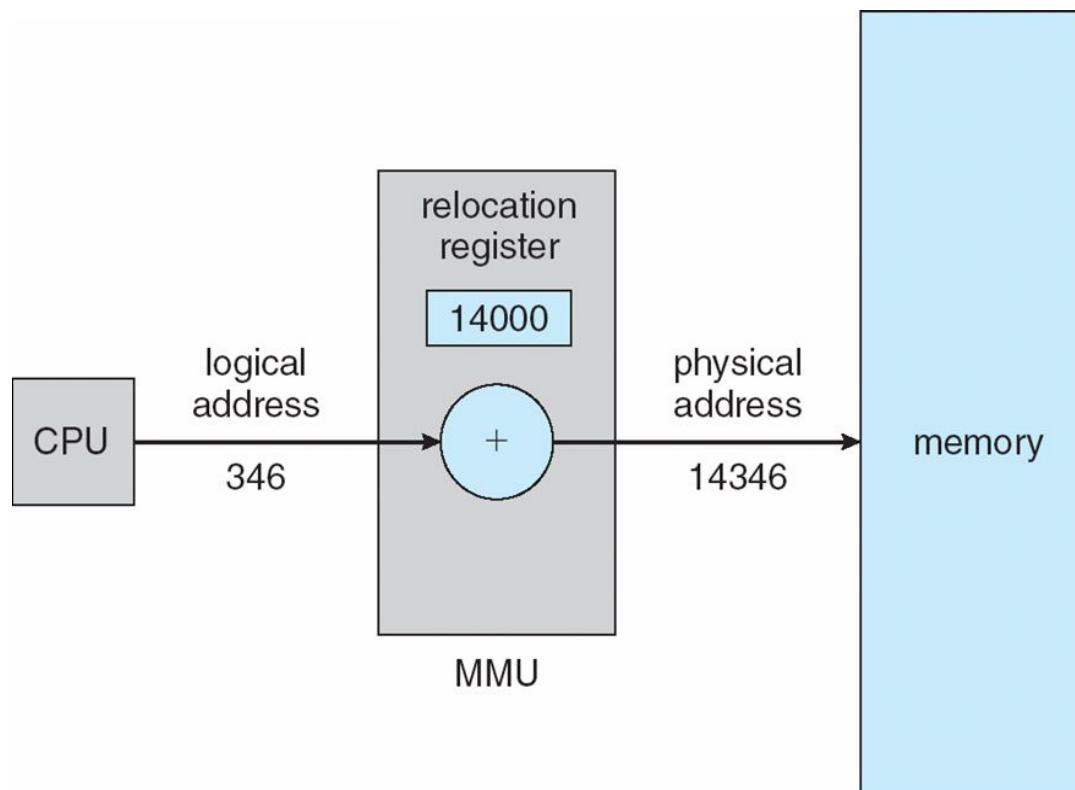
Schematic view of swapping



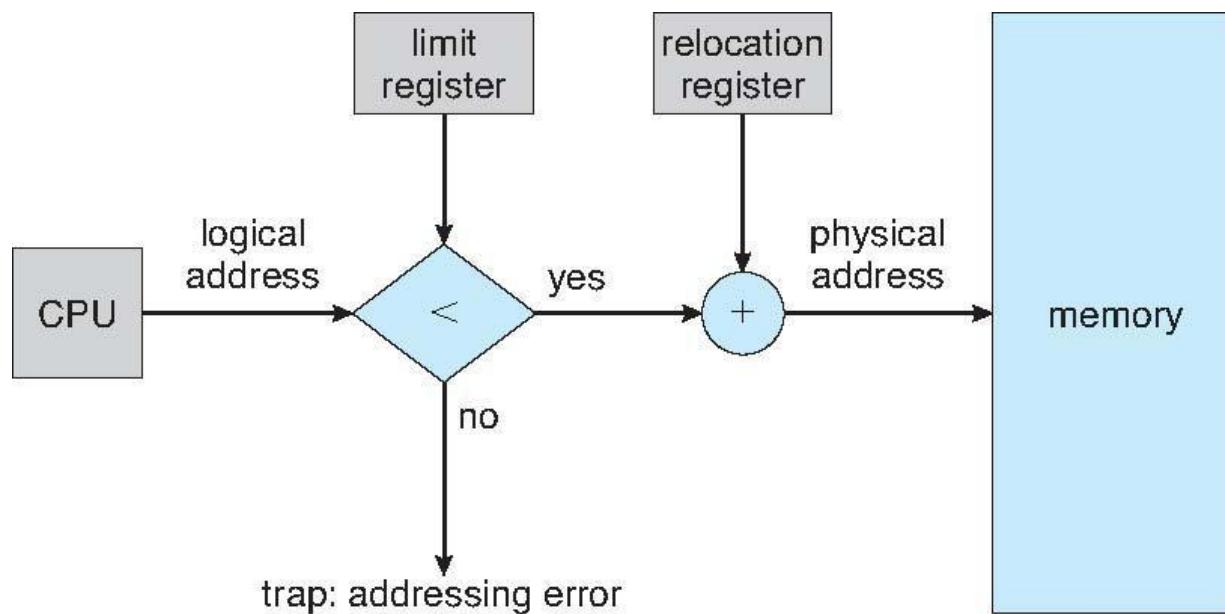
Contiguous Allocation

- Main memory usually into two partitions
 - Resident Operating System, usually held in low memory with interrupt vector.
 - User processes then held in high memory.
- Single partition allocation
 - Relocation register scheme used to protect user processes from each other, and from changing OS code and data.
 - Relocation register contains value of smallest physical address; limit register contains range of logical addresses - each logical address must be less than the limit register.

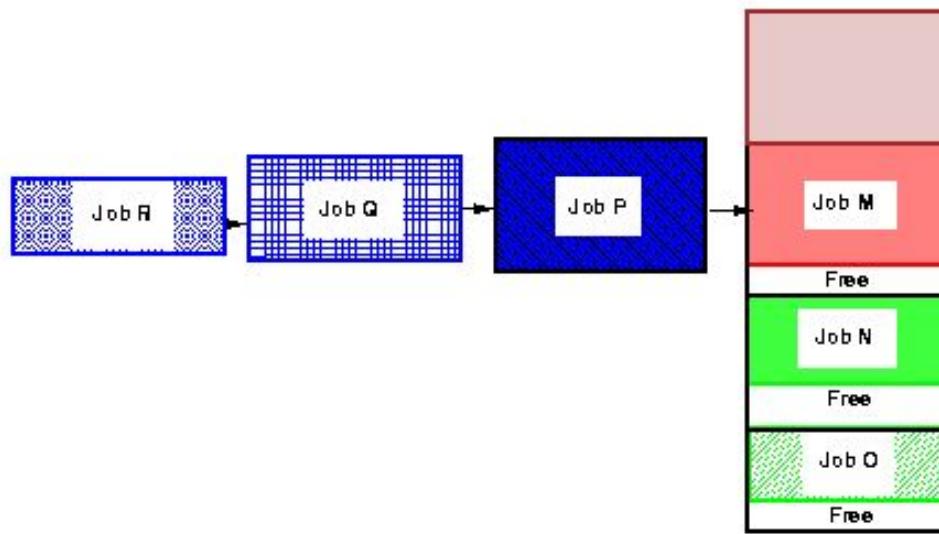
Relocation Register



Relocation and Limit Registers



Fixed partitions

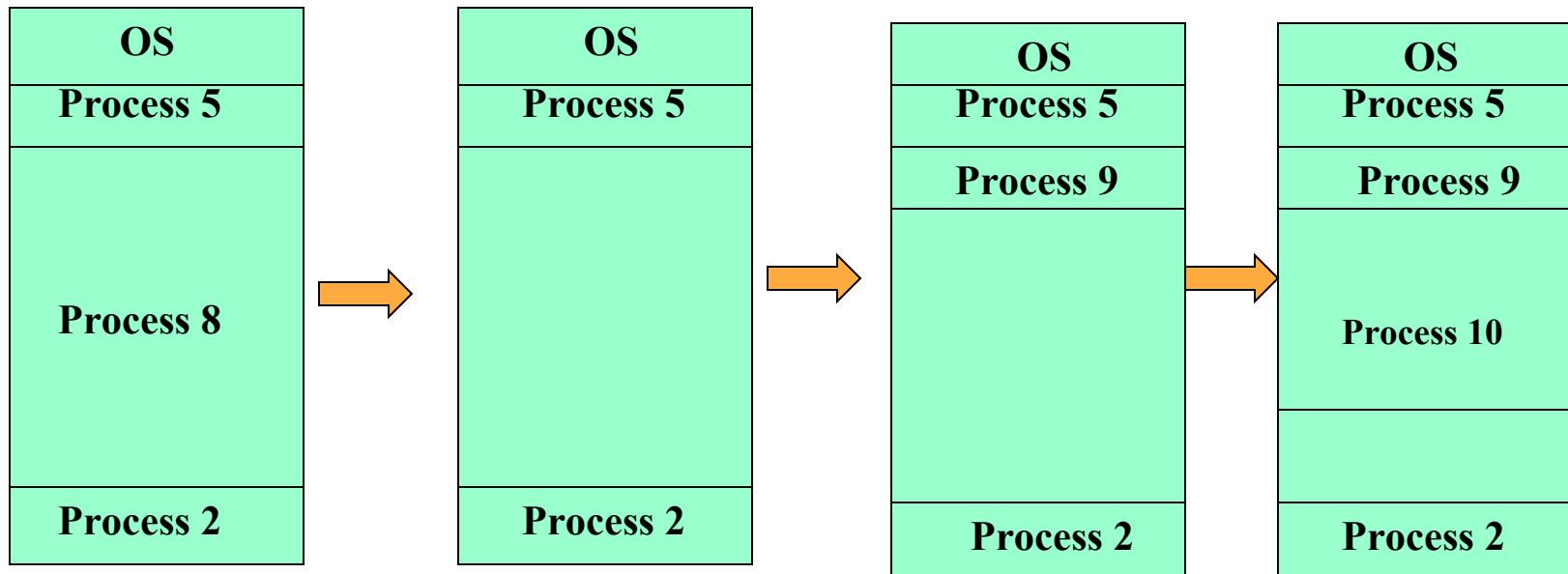


Contiguous Allocation (cont.)

■ Multiple partition Allocation

- Hole - block of available memory; holes of various sizes are scattered throughout memory.
- When a process arrives, it is allocated memory from a hole large enough to accommodate it.
- Operating system maintains information about
 - allocated partitions
 - free partitions (hole)

Contiguous Allocation example



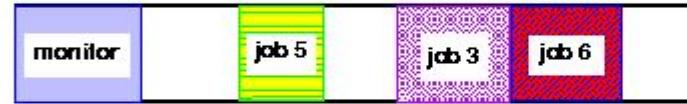
Dynamic Storage Allocation Problem

- ❑ How to satisfy a request of size n from a list of free holes.
 - First-fit
 - ❑ allocate the first hole that is big enough
 - Best-fit
 - ❑ Allocate the smallest hole that is big enough; must search entire list, unless ordered by size. Produces the smallest leftover hole.
 - Worst-fit
 - ❑ Allocate the largest hole; must also search entire list. Produces the largest leftover hole.
- ❑ First-fit and best-fit are better than worst-fit in terms of speed and storage utilization.

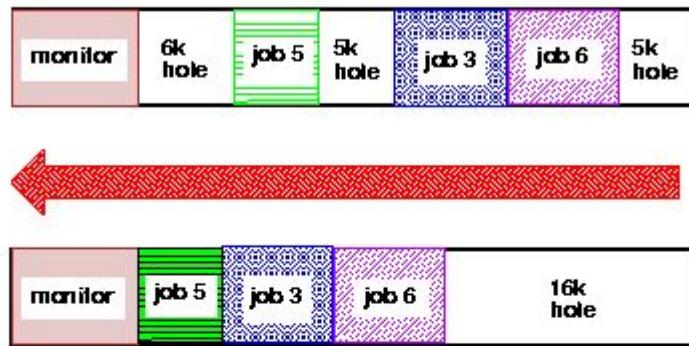
Fragmentation

- External fragmentation
 - total memory space exists to satisfy a request, but it is not contiguous.
- Internal fragmentation
 - allocated memory may be slightly larger than requested memory; this size difference is memory internal to a partition, but not being used.
- Reduce external fragmentation by compaction
 - Shuffle memory contents to place all free memory together in one large block
 - Compaction is possible only if relocation is dynamic, and is done at execution time.
 - I/O problem - (1) latch job in memory while it is in I/O (2) Do I/O only into OS buffers.

Fragmentation example



Compaction



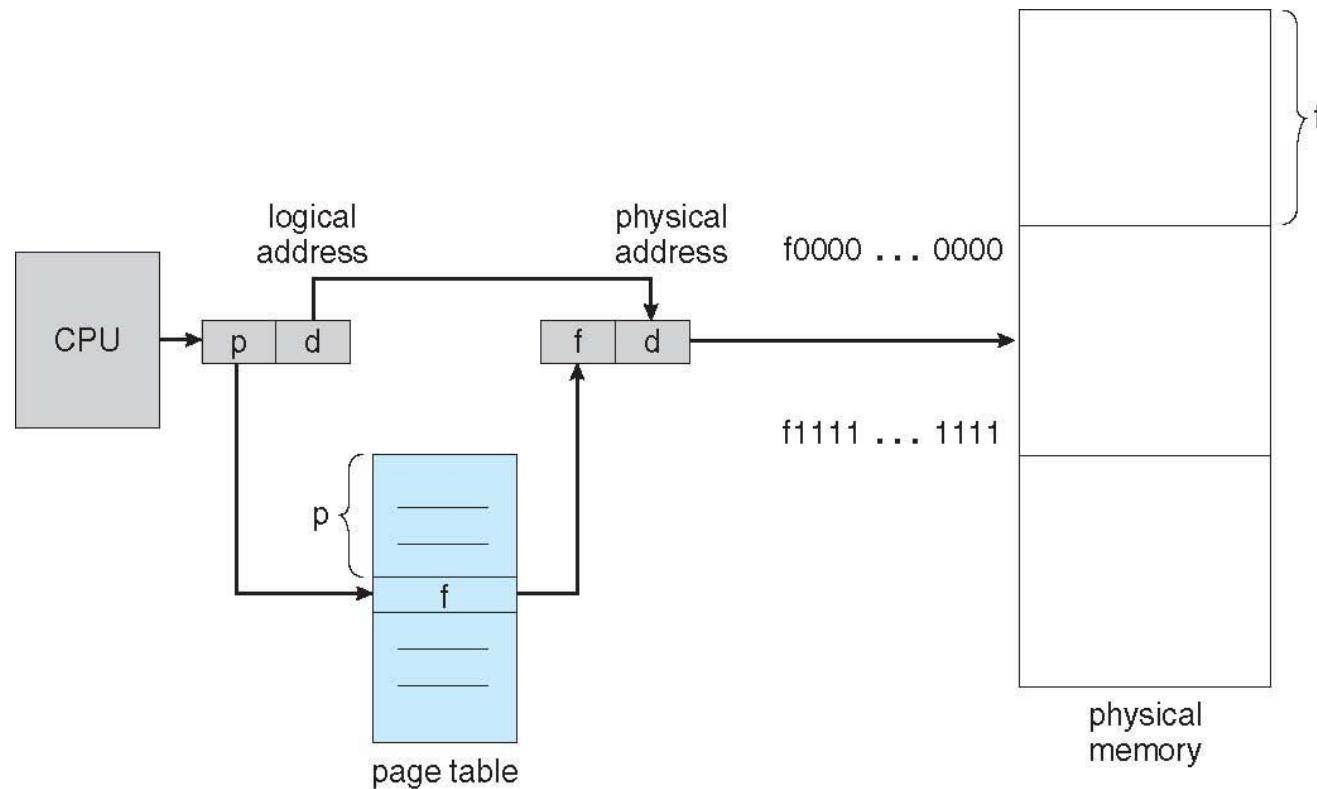
Paging

- Logical address space of a process can be non-contiguous;
 - process is allocated physical memory wherever the latter is available.
 - Divide physical memory into fixed size blocks called ***frames***
 - size is power of 2, 512 bytes - 8K
 - Divide logical memory into same size blocks called ***pages***.
 - Keep track of all free frames.
 - To run a program of size n pages, find n free frames and load program.
 - Set up a page table to translate logical to physical addresses.
 - Note:: Internal Fragmentation possible!!

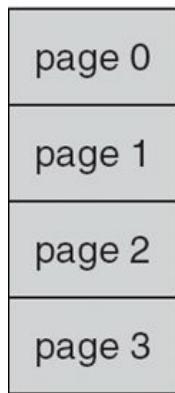
Address Translation Scheme

- Address generated by CPU is divided into:
 - Page number(p)
 - used as an index into page table which contains base address of each page in physical memory.
 - Page offset(d)
 - combined with base address to define the physical memory address that is sent to the memory unit.

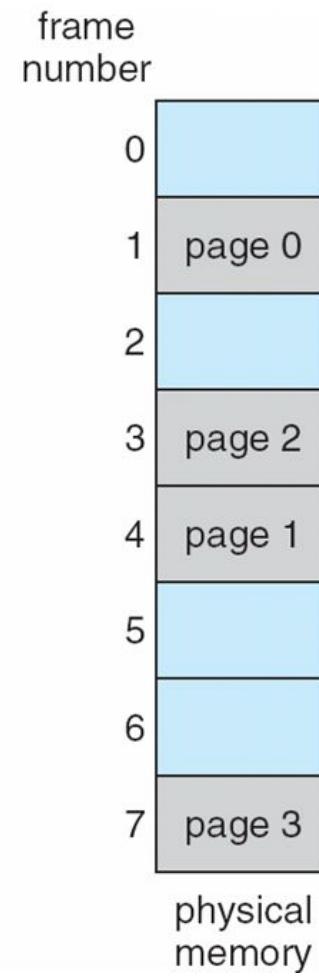
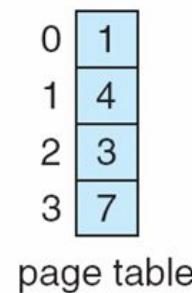
Address Translation Architecture



Example of Paging



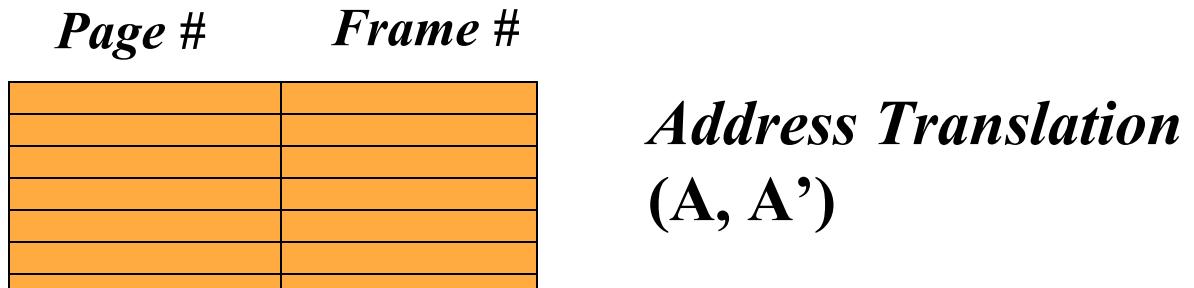
logical
memory



Page Table Implementation

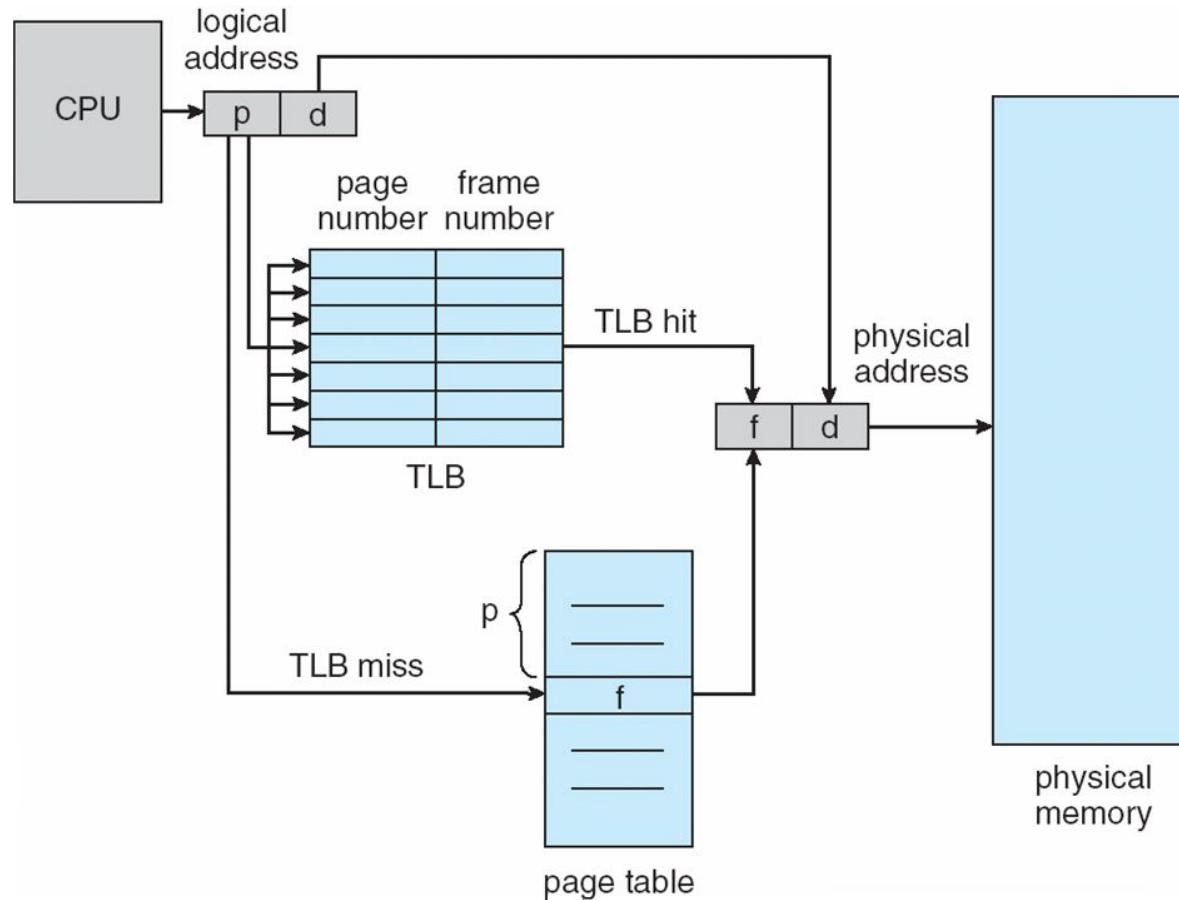
- Page table is kept in main memory
 - Page-table base register (PTBR) points to the page table.
 - Page-table length register (PTLR) indicates the size of page table.
- Every data/instruction access requires 2 memory accesses.
 - One for page table, one for data/instruction
 - Two-memory access problem solved by use of special fast-lookup hardware cache (i.e. cache page table in registers)
 - associative registers or translation look-aside buffers (TLBs)

Translation Lookaside Buffer (TLB) (aka Associative Registers)



- If A is in TLB, get frame #
- Otherwise, need to go to page table for frame#
 - requires additional memory reference
 - Page Hit ratio - percentage of time page is found in TLB.

Paging hardware with TLB



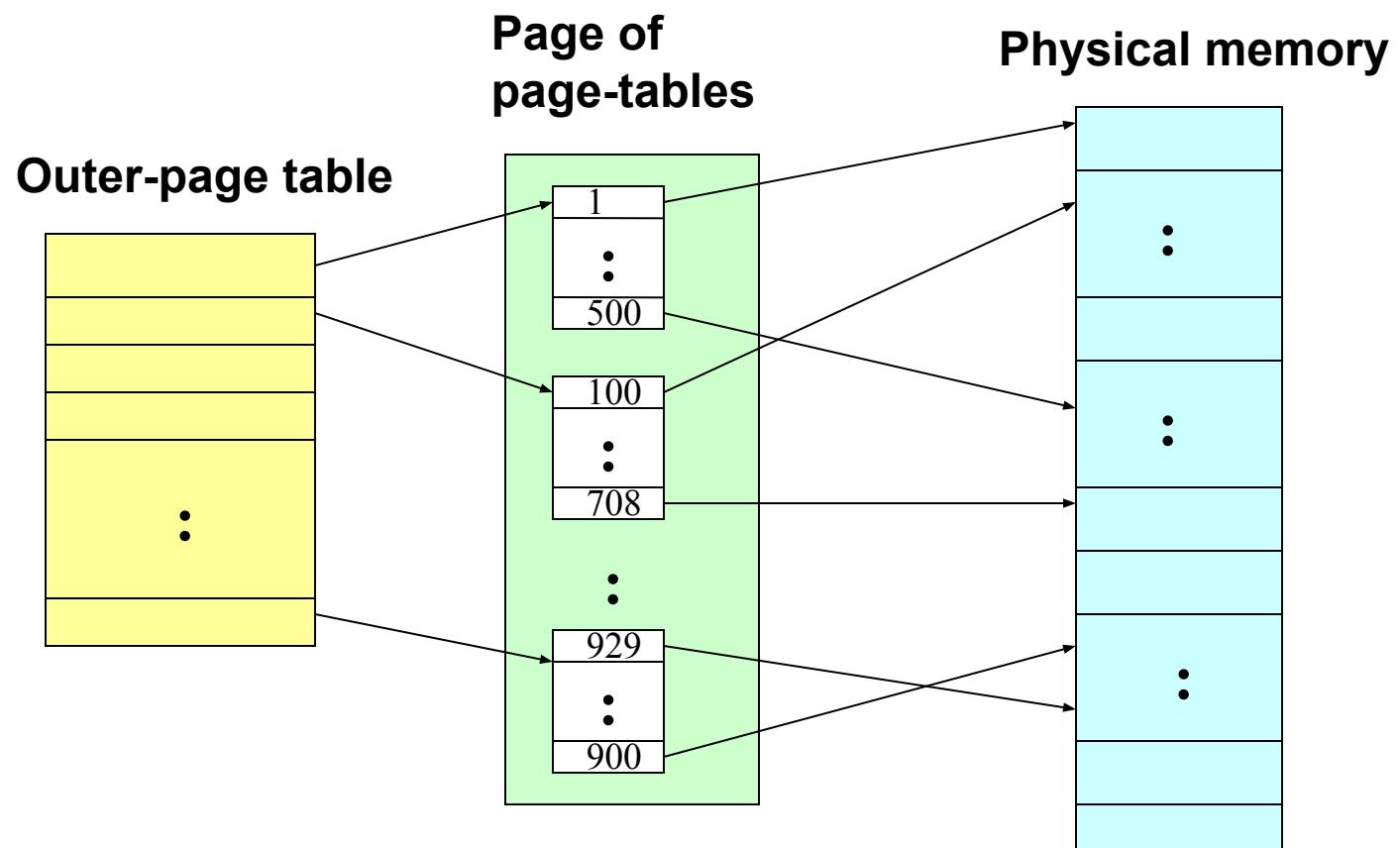
Effective Access time

- TLB lookup time = ε time unit
- Assume Memory cycle time = 1 microsecond
- Hit ratio = α
- Effective access time (EAT)
 - $EAT = (1 + \varepsilon) \alpha + (2 + \varepsilon) (1 - \alpha)$
 - $EAT = 2 + \varepsilon - \alpha$

Memory Protection

- Implemented by associating protection bits with each frame.
- Valid/invalid bit attached to each entry in page table.
 - Valid: indicates that the associated page is in the process' logical address space.
 - Invalid: indicates that the page is not in the process' logical address space.

Two Level Page Table Scheme

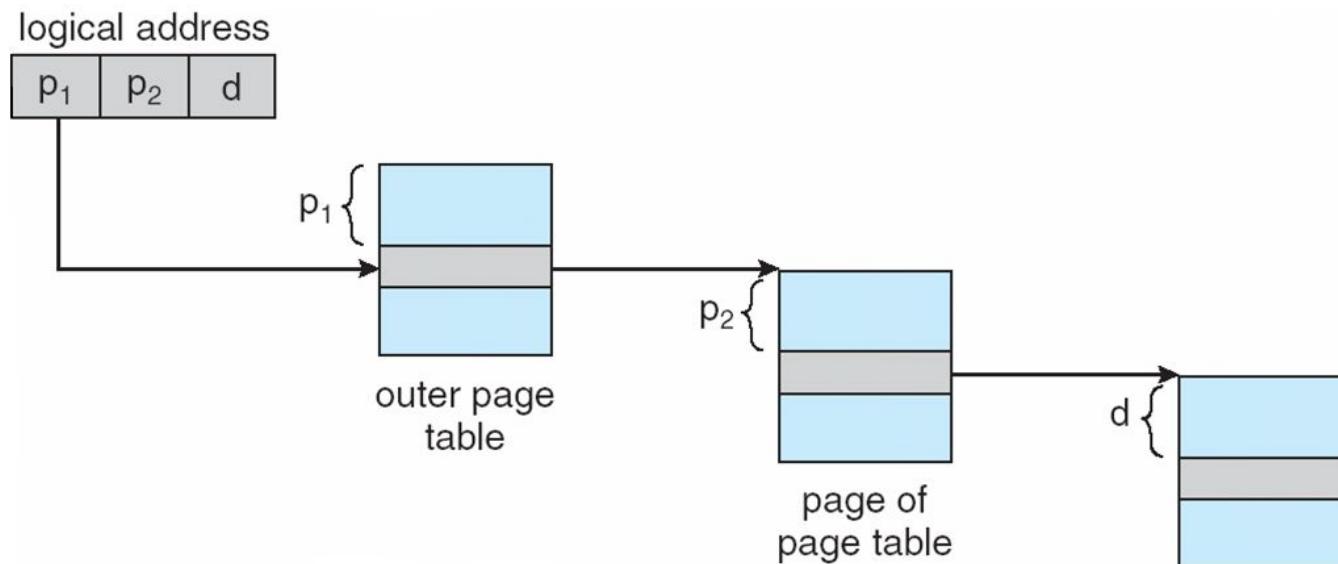


Two Level Paging Example

- A logical address (32bit machine, 4K page size) is divided into
 - a page number consisting of 20 bits, a page offset consisting of 12 bits
- Since the page table is paged, the page number consists of
 - a 10-bit page number, a 10-bit page offset
- Thus, a logical address is organized as (p_1, p_2, d) where
 - p_1 is an index into the outer page table
 - p_2 is the displacement within the page of the outer page table

| Page number | | Page offset |
|-------------|-------|-------------|
| p_1 | p_2 | d |

Two Level Paging Example



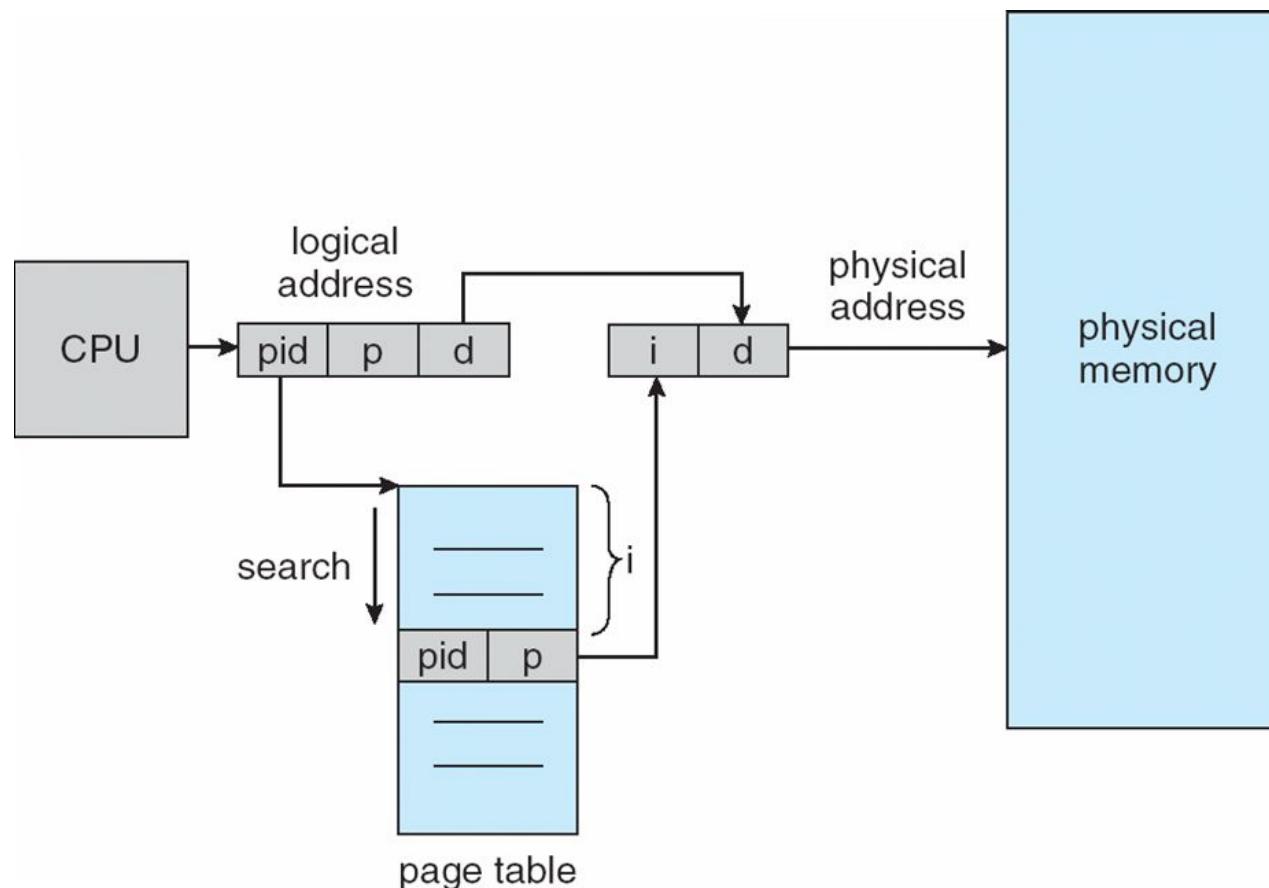
Multilevel paging

- Each level is a separate table in memory
 - converting a logical address to a physical one may take 4 or more memory accesses.
 - Caching can help performance remain reasonable.
 - Assume cache hit rate is 98%, memory access time is quintupled (100 vs. 500 nanoseconds), cache lookup time is 20 nanoseconds
 - Effective Access time = $0.98 * 120 + .02 * 520 = 128 \text{ ns}$
 - This is only a 28% slowdown in memory access time...

Inverted Page Table

- One entry for each real page of memory
 - Entry consists of virtual address of page in real memory with information about process that owns page.
 - Decreases memory needed to store page table
 - Increases time to search table when a page reference occurs
 - table sorted by physical address, lookup by virtual address
 - Use hash table to limit search to one (maybe few) page-table entries.

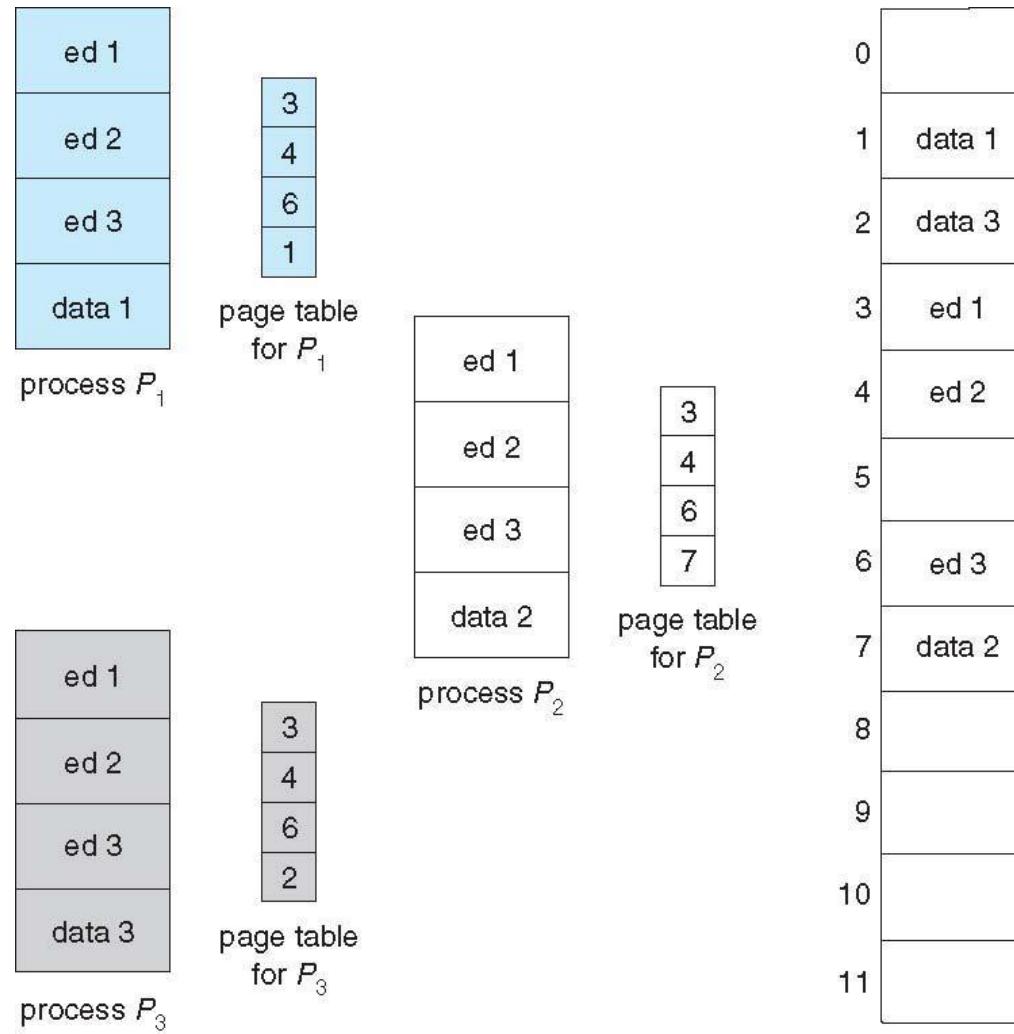
Inverted Page Table



Shared pages

- Code and data can be shared among processes
 - Reentrant (non self-modifying) code can be shared.
 - Map them into pages with common page frame mappings
 - Single copy of read-only code - compilers, editors etc..
- Shared code must appear in the same location in the logical address space of all processes
- Private code and data
 - Each process keeps a separate copy of code and data
 - Pages for private code and data can appear anywhere in logical address space.

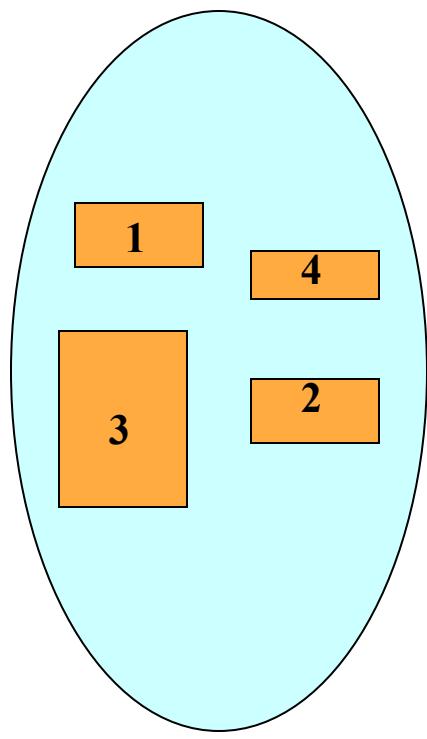
Shared Pages



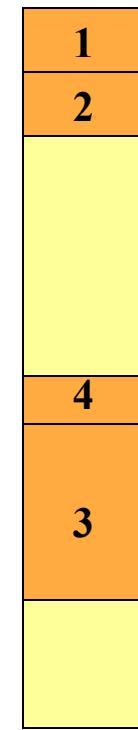
Segmentation

- Memory Management Scheme that supports user view of memory.
- A program is a collection of segments.
- A segment is a logical unit such as
 - main program, procedure, function
 - local variables, global variables, common block
 - stack, symbol table, arrays
- Protect each entity independently
- Allow each segment to grow independently
- Share each segment independently

Logical view of segmentation



User Space



Physical Memory

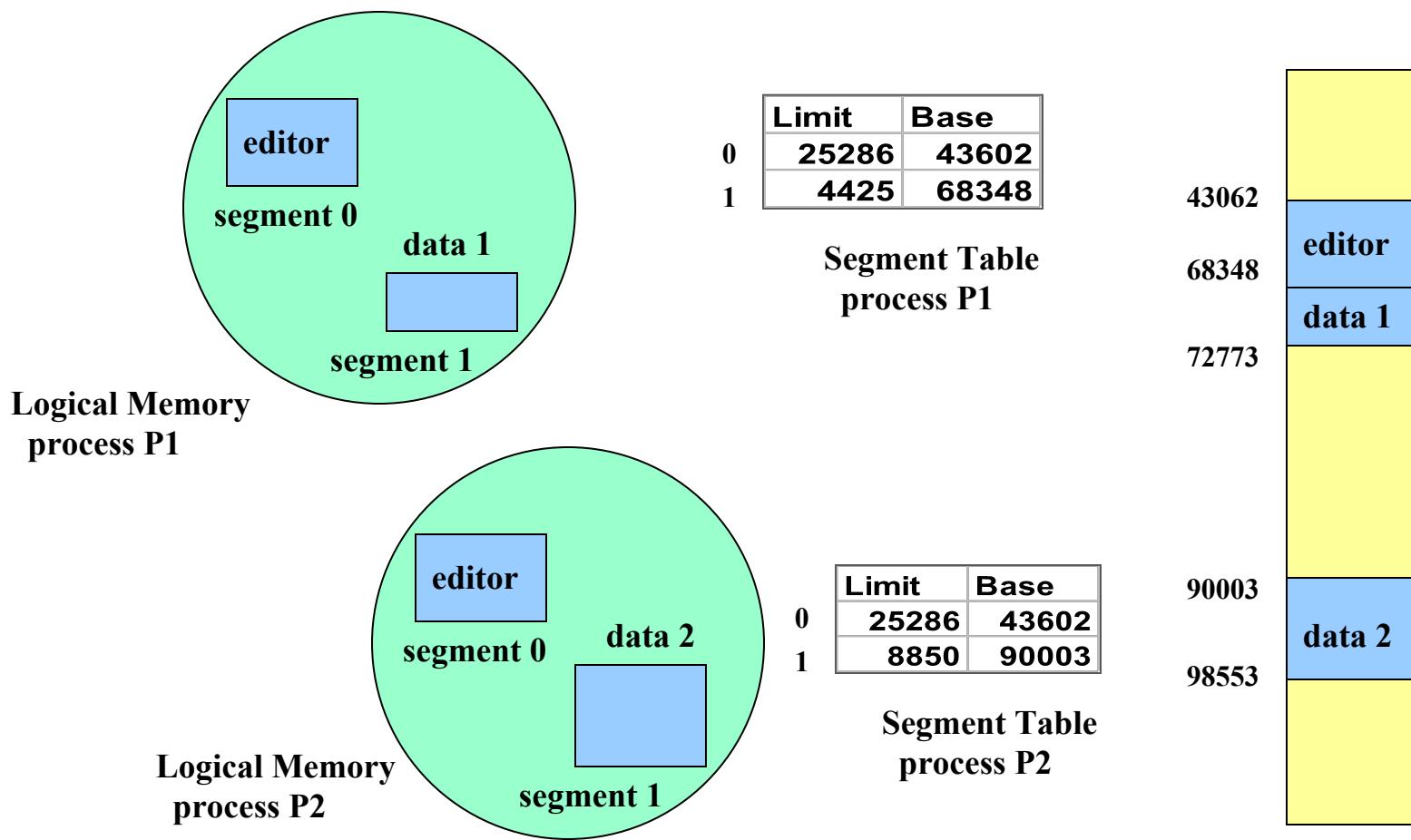
Segmentation Architecture

- Logical address consists of a two tuple
 <segment-number, offset>
- Segment Table
 - Maps two-dimensional user-defined addresses into one-dimensional physical addresses. Each table entry has
 - Base - contains the starting physical address where the segments reside in memory.
 - Limit - specifies the length of the segment.
 - *Segment-table base register* (STBR) points to the segment table's location in memory.
 - *Segment-table length register* (STLR) indicates the number of segments used by a program; segment number is legal if $s < \text{STLR}$.

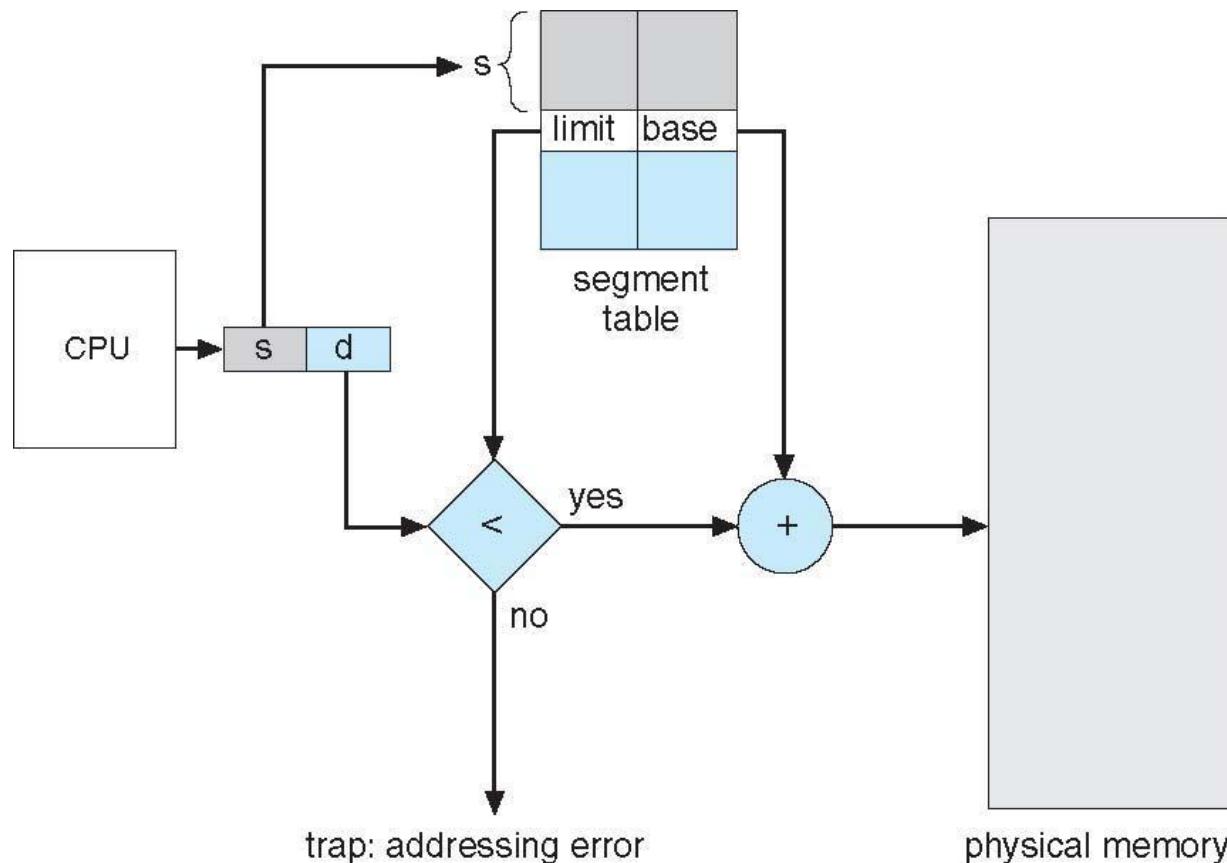
Segmentation Architecture (cont.)

- Relocation is dynamic - by segment table
- Sharing
 - Code sharing occurs at the segment level.
 - Shared segments must have same segment number.
- Allocation - dynamic storage allocation problem
 - use best fit/first fit, may cause external fragmentation.
- Protection
 - protection bits associated with segments
 - read/write/execute privileges
 - array in a separate segment - hardware can check for illegal array indexes.

Shared segments



Segmentation hardware



Segmented Paged Memory

- Segment-table entry contains not the base address of the segment, but the base address of a page table for this segment.
 - Overcomes external fragmentation problem of segmented memory.
 - Paging also makes allocation simpler; time to search for a suitable segment (using best-fit etc.) reduced.
 - Introduces some internal fragmentation and table space overhead.
- Multics - single level page table
- IBM OS/2 - OS on top of Intel 386
 - uses a two level paging scheme

MULTICS address translation scheme

