Course Introduction
CS239
Advanced Operating Systems
April 1, 1996

CS239 - Advanced Operating Systems

Pre-requisites - CS111

Prof. Peter Reiher

E-mail: reiher@cs.ucla.edu

Grading

- Mid-term - 25%
- Paper - 25%
- Final exam - 50%

Tests

Probably closed book

Probably a combination of short answer/multiple choice and essay questions

Midterm in the 5th week

Paper

Paper describing some important recent research on some operating system topic

Around 12-15 pages

Reading Materials

No textbook

Papers assigned weekly

When possible, papers will be accessible via WWW

http://www.cs.ucla.edu/project-members/reiher/CS239_spring.html

Other papers will be put on reserve in archives

Office Hours

In BH3732J

Time to be decided based on student schedules

Generally available by appointment

Class Outline

- Review of basic OS concepts
- File systems
- IPC mechanisms
- OS security
- Distributed operating systems
- OS organizations
- New topics in operating systems

Review of OS Concepts

- Overview
- Basic OS organization
- Processes and scheduling
- Interrupts
- Synchronization
- Virtual memory
- Basic file systems

Overview

Purposes of OS -

- Clean virtual machine
- Architecture independence
- Resource sharing & utilization management
- Long term data storage
- Protection, security, accounting
- Real time support, parallelism, human interface

Changing Roles of the OS

- What the OS does depends on available hardware and software
- And on changing uses of machines
- And on changing expectations of users

OS Structural Issues

- Interactivity?
- Multiple users?
- Multiple concurrent processes?
- Multiple processors?
- Multiple machines?
- Virtual memory?
- Real time support?

Organization of an OS

Components include:

- OS kernel
- Interrupt vectors
- Bootstrap program
- Device drivers
- Various servers

Major Divisions of a Basic OS

- Program execution and process management
- File system management and device support
- Network and communications management
- User interface support

OS Organization Strategies

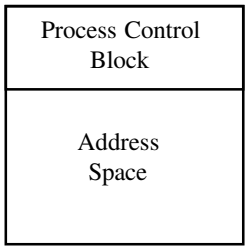
- Monolithic organization
- Layered organization
- Virtual machine systems
- Microkernel systems
- Object oriented systems

Processes and Scheduling

What is a process?

- Unit of processor scheduling
- Sequential activity
- Usually in control of private address space

Components of a Process



Process Control Blocks

Contains all information about process that the OS needs to know -

- Process name and type
- Process state image
- Scheduling info
- Address space location
- Process handling information
- Lots of other stuff

Process Address Space

- A physical or virtual segment or segments of memory
- Protected against access by other processes
- May be stored in core or on disk

  Or both

Process States

- Running
- Ready
- Blocked
- Swapped out

Typical Process Flow

- Ready process reaches head of queue and starts running
- Process runs till end of time slice
- Process is switched out and moved to end of queue
- Alternately, process blocks and is moved to end of queue
- Or process exits and is destroyed

Process Scheduling

How to handle needs of simultaneously active processes?

Need to provide -

- Fairness
- Effective utilization of resources
- Responsiveness

Every so often, switch from one process to another

When to Switch Processes?

- When time slice is used up
- When running process blocks for some reason
- When interrupt occurs
- When running process encounters a fault

Context Switching

Effectively changing the attention of the processor

Requires -

- Stopping running process cleanly
- Saving context of stopped process
- Loading context of next process
- Starting up next process
- Accounting and scheduling of administration

Scheduling Algorithms

- Round robin
- Preemptive priority
- Aging preemptive priority
- Shortest deadline first
- Shortest remaining time first
- Multilevel feedback

Interrupts

Hardware ability to change current path of execution unexpectedly

Generally based on either

- An internal processor event
- An external stimulus

OS must determine what to do when an interrupt occurs

Purpose of Interrupts

- To deal with high priority activities that cannot wait
- To handle real time constraints
- To maintain control over processor usage
- To recover from errors

Interrupts and Context Switches

- Interrupts usually lead to context switches
- An entirely different piece of code starts running

  For an entirely different purpose
- Old context may or may not be returned to

Kinds of Interrupts

- Trap instructions
- Software errors or exceptions
- Hardware failures
- I/O interrupts
- Timers
- Page faults

Handling an Interrupt

1. Save processor state
2. Consult interrupt vector table
3. Transfer control to interrupt handler
4. Interrupt handler does its work
5. Typically, control returns to OS quickly

Implication of Interrupts

- Asynchrony
- Effective parallelism
- Non-determinism
- Real time management possible

Synchronization

Real systems are non-deterministic

Due to concurrency

But we must hide much non-determinism from users

Synchronization provides determinism where important

Guarantees correct program execution regardless of interleaving

Sources of Non-Determinism

- Device timing
- Code execution timing
- Effects of multiple active processes
- User interactions

Important Synchronization Concepts

- Cobegin/coend
- Atomic actions
- Mutual exclusion
- Semaphores
- Deadlock

Cobegin/Coend

Basic synchronization construct

Statements bracketed by `cobegin`/`coend` execute concurrently

None begin before statements preceding `cobegin` finish

No statements after `coend` start until bracketed statements end

Cobegin/Coend Example

```
x = 0;
cobegin
x = x + 1;
y = x;
coend;
z = y + 1;
```

What value does `z` have?

Atomic Actions

A sequence of instructions that acts as a single, uninterruptable instruction

At any instant, all actions have been performed

Or none have

As seen from outside sequence

Mutual Exclusion

Ensures that no two processes are in critical section of code simultaneously

Can be used to provide atomic actions

Can be implemented in various ways

Semaphores

Comprehensive synchronization mechanism

Can provide mutual exclusion

Basically, a semaphore protects access to a shared resource; for example -

- Code segment
- Device
- File

Semaphore Operations

Three operations on semaphores

- Create and Initialize(s, n)
- P(s)
- V(s)

P Semaphore Operation

Either grants control of a resource protected by a semaphore

If resource not in use

Or blocks waiting for current controller to finish

If someone else is using it

Can't use resource till P operation succeeds

V Semaphore Operation

Releases control of protected resource

If any processes are waiting on semaphore,

Wakes exactly one of them

Waking looks the same as immediate return from P operation

Uses of Semaphores

Can provide many synchronization services

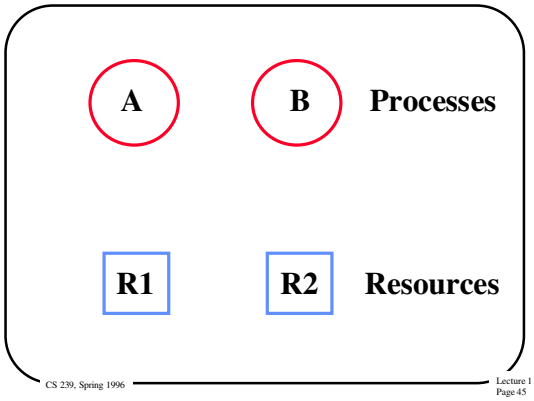
- Mutual exclusion
- Message rendezvous
- Coend
- Reader/writer on shared data

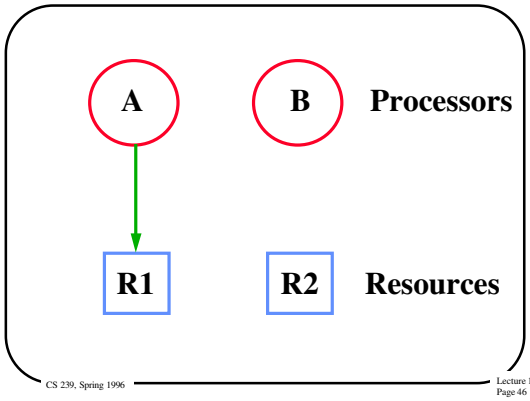
Deadlock

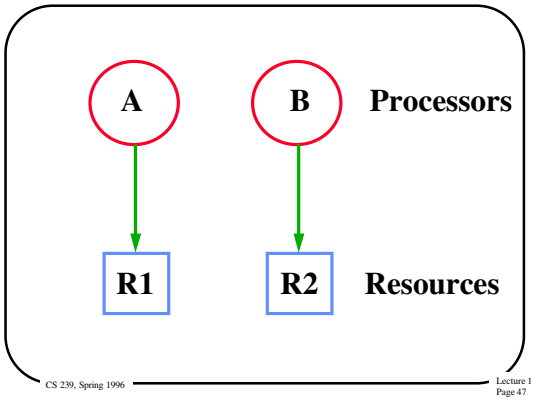
When a set of processes wait on resources locked by each other

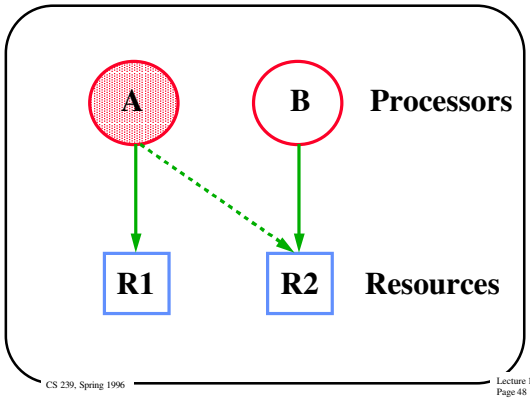
Can wait for

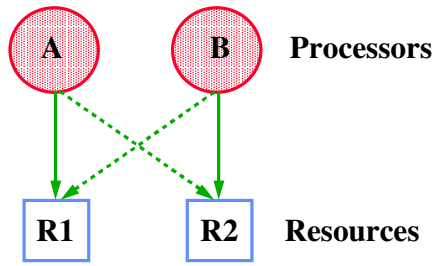
- critical code sections
- resources











Deadlock!

Dealing With the Deadlock Problem

- Ignore the problem
- Prevent deadlocks from occurring
- Detect deadlocks and recover from them

Ignoring Deadlocks

Often deadlocks are low probability

Cost of doing anything else may be too high

Resolve the few that occur by extra-system means

Users kill processes, reboot, etc.

Preventing Deadlocks

Not always possible

Often expensive, when possible

Generally overly conservative

Detecting and Solving Deadlocks

Not always possible

But possible for some important cases

Generally by detecting circularity in dependency graphs

Tradeoff - checking frequency vs. cost of mechanism

Memory Management

What if you don't have enough real memory to support processes' needs?

- Multiple processes needing more total memory than available
- Individual large processes

Process Swapping

Move processes in and out of main memory, as necessary

Those "out" of memory reside on disk

Allows multiple processes to be active at the same time

Even if there isn't enough memory for all

But swapping doesn't work for big processes

And it's slow

Virtual Memory

Allow processes to have large memory space

Divide main memory into *page frames* of fixed size

Divide processes' memory into *pages*

Same size as page frames

Only keep certain pages in main memory

Keep other pages on disk

Page Faults

What happens when a process asks for a page that isn't in main memory?

Generate a page fault that:

- Halts the process
- Generates a request to bring page from disk
- Swaps in another process

Page Faults, Continued

When page request from disk is ready

- Halt running process
- Choose where to put the page

  Possibly swapping a page out
- Clear the faulted process
- Start some process running

How Do You Do It?

Every address reference may generate page fault

Pages may sit anywhere in physical memory

Need hardware to translate virtual addresses to real addresses

Must be very fast

Page Tables

List of physical locations for virtual pages

Typically, each process has its own page table

Kept in core or in special hardware

But it's too slow to always consult page tables

Translation Lookaside Buffers

Cache of page table entries

Typically associative memory

i.e., special hardware

Ideally, all page requests satisfied here

Flushed on process switch

Typically LRU

Essential for sufficient speed

TLB and Hit Ratio

Hit ratio is the fundamental measure of TLB effectiveness

Hit ratio = fraction of memory accesses satisfied by TLB

Must be very, very high for virtual memory to be fast enough

Page Replacement Algorithms

When you get a page fault, what page gets thrown out?

- FIFO
- Optimal
- Least Frequently Used
- Least Recently Used
- Not Recently Used

Locality

Virtual memory can only work because of locality

Memory references tend to be clustered in a small number of pages

Over short periods of time

Most data access turns out to have good locality

But not all

Page Allocation Policy

How many page frames should a process get?

Too few and it faults all the time

Too many and other processes fault all the time

Processes with poor locality need more frames

Typical OS problem of resource management

Working Set Theory

WS(w,t) - at time **t**, set of pages accessed in the last **w** memory references

w is *window size;* should be a multiple of time slice

Regularly measure each process' WS

Increase or decrease allocation to match

Files and File Systems

Files are persistent collections of data

Typically organized into *file systems*

File systems provide generic way to access persistent data

File System Fundamentals

File systems should provide

- Data organization mechanisms
- Data naming mechanisms
- Data access primitives
- Synchronization
- Protection and security

Directories

Typically, file systems are hierarchically organized

Normal files are leaves in the tree

Non-leaves are called *directories*

Directories contain organizational information

And help file systems locate data

Directories are also files, of course

File Systems and Disks

Most typically, permanent file storage is on disk

File system must decide disk storage location for each data page

File system data page need not be the same size as VM page

Must be able to find data later

Complex issues to achieve good performance