Review of CS111
CS239
Advanced Operating Systems
1997

Review of OS Concepts

- Overview
- Basic OS architecture
- Processes, IPC, and scheduling
- Memory management
- Input/output
- File systems
- Distributed file system basics

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
- Memory management
- File system management and device support
- Network and communications management
- User interface support

OS Organization Strategies

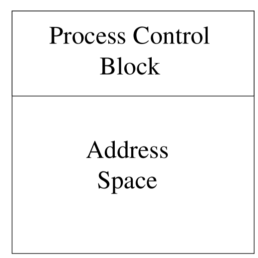
- Monolithic organization
- Layered organization
- Virtual machine systems
- Microkernel systems
- Object oriented systems

Processes and Scheduling

What is a process?

- Unit of processor scheduling
- Abstraction for unit of resource allocation
- Sequential activity
- Usually in control of private address space

Components of a Typical 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
- Killed

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
- Or process is interrupted

Interrupts

Hardware ability to change current path of execution unexpectedly

Generally based on either

- An internal processor event
- An external stimulus

Interrupts usually lead to context switches

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

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

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

Threads

- An abstraction for execution
- A process is composed of multiple threads
- All process threads share common resources
  - Heap, open files, IPC channels, etc.
- But each is an independent flow of execution through a program's code
- Especially useful for programs that inherently support multiple activities

Where Should Threads Live?

- Does the kernel know about threads or not?
- If it does,
  - Kernel can schedule threads
  - But kernel threads may be expensive (close to context switch)
- If it doesn't,
  - Kernel can't schedule threads
  - But user-space implementation is cheaper (no context switch)

Interprocess Communications

- IPC, for short
- Allows processes to interact
- But leads to many potential problems
  - Particularly synchronization and deadlock problems
- Similar problems are common internally in operating systems

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

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

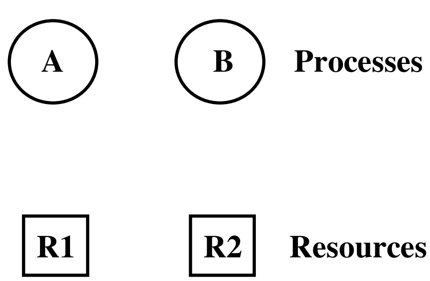
- Mutual exclusion
- Message rendezvous
- Coend
- Reader/writer on shared data

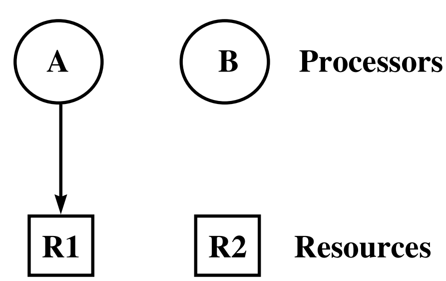
Deadlock

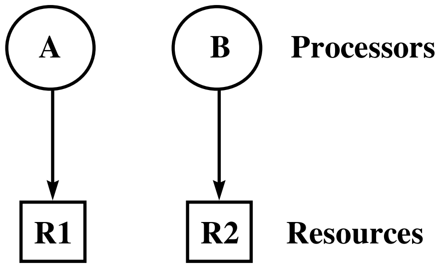
When a set of processes wait on resources locked by each other

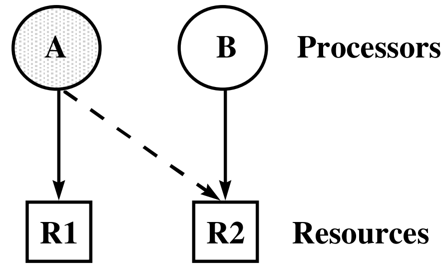
Can wait for

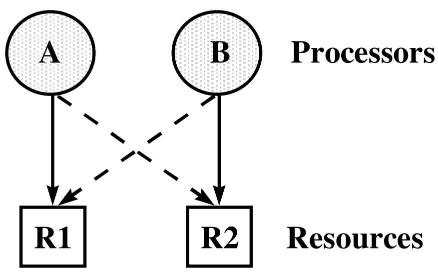
- critical code sections
- resources











Deadlock!

Dealing With the Deadlock Problem

- Ignore the problem
  – Good solution if probability very, very low
- Prevent deadlocks from occurring
  – Not always possible; many algorithms exist
- Detect deadlocks and recover from them
  – Common solution; many algorithms exist

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

Scheduling Algorithms

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

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

Dependent on hardware support (MMU)

Page Faults

When VM needs page not in main memory,

- Halt the process
- Generate a request to bring page from disk
- Swap in another process

When page request from disk is ready

- Halt running process
- Choose where to put the page
- Clear the faulted process
- Start some process running

How Do You Do It?

Any address reference may cause page fault

Pages may sit anywhere in physical memory

Need hardware to translate virtual addresses to real addresses

Must be very fast

Page tables and translation lookaside buffers

VM and Hit Ratio

Hit ratio measures VM effectiveness

Hit ratio = fraction of memory accesses satisfied by pages in memory

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

Translation lookaside buffers have their own hit ratio (which also must be very, very high)

Virtual Memory Policies

- Page fetch policy - when do you get a page and what pages do you get?
- Page placement policy - what page frame is chosen to hold particular pages?
- Page replacement policy - if all page frames are currently filled when a fetch occurs, which page is displaced?

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 is important to this issue

Input/Output

- Communicating outside the main memory of the computer is increasingly important
  - Pages from local disk
  - Input from mouse/keyboard
  - Windowing systems for screen display
  - Network communications
  - File systems/databases

Characteristics of I/O

- Highly variable
  - Disks aren't like networks aren't like screens
  - Even within a class, lots of variability
- Typically complex
  - Data formatting issues
  - Tricky synchronization
  - Performing well often challenging
- Operating system support usually required

Typical OS Approach to I/O

- Offer a few limited abstractions describing I/O
  - Easy for users to understand and work with
  - But not closely related to real devices
- Use device drivers to map from pleasant abstractions to unpleasant reality

Typical I/O Abstractions

- Files
- Data streams
- Block and character special devices
- I/O channels

Device Drivers

- Most common method of OS support for I/O
- Per-device type
  - Code highly specific to particular device
- Handles all the grungy details of that device

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

Buffer Caching in File Systems

- OS cannot afford typical disk seek and transfer times on every file access
- To speed up average access, keep some disk blocks in a cache in core
- Proper buffering can make a big performance difference

On-Disk File System Layout

- How files and file blocks are organized on disk has strong performance implications
- Various strategies exist to minimize the time spent reading from/writing to disk
- Which to use depends on relative disk/CPU/memory performance
  - And file access patterns

Standard Unix File Systems

- Use a per-file data structure (inode) to find a file's data blocks
  - Not necessarily contiguous
- Try to keep single file's blocks and those for "related" files close together
- Make writes atomic by changing the inode

Files, Directories, and Volumes

- Files are collections of data managed individually by the file system
- Directories are files containing pointers to other files
- Volumes are collections of directories and files
  - Typically contiguous subtrees
  - **Mounting** used to splice them into a single file namespace

Log Structured File Systems

- Assume a buffer satisfies almost all reads
- Then file system performance dominated by writes
- So group writes together into log, committing blocks of the log
  - Rather than individual files
- Finding file blocks and cleaning up the log complex

Using Parallelism to Speed Up File Systems

- Instead of reading from one disk at a time, read from n
  - Giving n-fold improvement in bandwidth
- RAID - Redundant Array of Independent Disks
- Also has good reliability characteristics
- Cost and scale problems sometimes limit effectiveness

Distributed File Systems

- Networking makes distributed file systems possible
- Allowing machines to perform file system operations on files they don't themselves store

Issues in Distributed File Systems

- Transparency
- Naming
- Performance
- Reliability
- Security

NFS

- Network File System
- Relatively simple distributed file system
- Essential idea - allow machines to mount remote volumes
- Strategy - largely transparent access to remote volumes
  - Stateless
  - Using underlying data transport to move bits

How NFS Works

- At mount point of remote volume, trap name translations
- Instead of going to disk, send request to remote site storing the volume
- Remote site fetches blocks from disk/buffer and sends them back
- Transparently give the blocks to the user